KIT SCIENTIFIC REPORTS 7598

3rd Many-core Applications Research Community (MARC) Symposium

Diana Göhringer
Michael Hübner
Jürgen Becker
(Hrsg.)

Scientific Publishing

Diana Göhringer, Michael Hübner, Jürgen Becker

3rd Many-core Applications Research Community (MARC) Symposium

Karlsruhe Institute of Technology

KIT SCIENTIFIC REPORTS 7598

3rd Many-core Applications Research Community (MARC) Symposium

Diana Göhringer
Michael Hübner
Jürgen Becker
(Hrsg.)

Scientific
Publishing

Report-Nr. KIT-SR 7598

Impressum

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe
www.ksp.kit.edu

KIT – Universität des Landes Baden-Württemberg und nationales
Forschungszentrum in der Helmholtz-Gemeinschaft

KIT Scientific Publishing 2011
Print on Demand

ISSN 1869-9669
ISBN 978-3-86644-717-2

Preface

The 3rd Many-core Applications Research Community (MARC) Symposium was held at Fraunhofer Institute of Optronics, System Technologies and Image Exploitation (Fraunhofer IOSB) in Ettlingen, Germany on July 5-6, 2011.

For the first time in the history of this symposium, a peer review for the 26 submissions had been organized which led to 12 accepted papers for oral presentation. Since the submitted paper were of high quality, the symposium organizer decided to accept the remaining 14 paper as poster presentation. This proceeding therefore contains all submitted papers, which were updated according to the reviewer feedback.

The symposium had a high registration rate with 64 attendees from Germany, the Netherlands, Switzerland, USA, Spain, Italy, Greece, Cyprus, Israel, New Zealand and Korea.

After a welcome and introduction note from Diana Göhringer from Fraunhofer IOSB and Ulrich Hoffmann from Intel Labs Braunschweig, the symposium started with the technical presentations. We kindly want to thank all authors for presenting their work so well and with high quality. Furthermore, we want to thank all attendees for the constructive discussions during the symposium. All this led to a very fruitful and well received event.

A special thank is dedicated to Prof. Maurus Tacke, director of the Fraunhofer IOSB, who enabled the event and gave an introduction into the research and development activities of the institute. We also want to thank the keynote speaker, Intel Fellow Jim Held, who presented a talk about *"The SCC MARC objectives and opportunities"*. Werner Haas from Intel Labs Braunschweig presented a tutorial with the important topic "*Shared Memory on SCC - pitfalls on the way to data consistency*". Michael Riepen from Intel Labs Braunschweig presented the concluding tutorial, the *"SCC Working Environment"*.

We kindly want to thank the Intel Research Group in Braunschweig for their excellent and kind support of the symposium. Furthermore, we want to thank the Karlsruher Institute of Technology (KIT) and the MSC Vertriebs GmbH und Gleichmann & Co. Electronics GmbH for their great support.

Last but not least, we want to thank all colleagues at Fraunhofer IOSB who helped with the management, catering, demonstration, web-page and the daily errands which ensured a smooth and professional organization of the MARC symposium.

Co-Organizer of the 3rd MARC Symposium

Diana Göhringer, Fraunhofer IOSB, Ettlingen Germany

Ulrich Hoffmann, Intel Labs Braunschweig, Germany

Michael Hübner, Karlsruhe Institute of Technology (KIT), Germany

Jürgen Becker, Karlsruhe Institute of Technology (KIT), Germany

Content

Poster Session

Application-Level Automatic Performance Tuning on the Single-Chip Cloud Computer

Victor Pankratius
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
pankratius@kit.edu, www.victorpankratius.com

Sven Blaese
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
sven.blaese@student.kit.edu

***Abstract*—Improving application performance is the main motivation to switch to manycore hardware and parallelize all kinds of software. Intel's Single-chip Cloud Computer (SCC) offers a testbed with 48 general-purpose cores on one chip as a first step towards chips with even more cores. However, a current difficulty is that programmers are responsible for controlling a variety of performance-affecting parameters that are interdependent and that are influenced by both hardware and software. Thus, tuning parallel applications on the SCC is far from trivial, and manual approaches are tedious. To address these problems, this paper is the first to introduce an application-level automatic performance tuning approach on the SCC. Programmers identify performance-impacting parameters within their software applications and let an external auto-tuner search for the best configuration. We discuss and evaluate several tuning algorithms, including HyDES, our own approach that combines Differential Evolution and Nelder-Mead Simplex methods in a novel way. First results on parallel compression and other applications show that HyDES can significantly boost performance. In addition, our measurements reveal that good parameter configurations can be non-intuitive; in certain scenarios, existing programs such as MPIBZIP do not even allow users to set the optimal performance parameter configuration from the command line. Our auto-tuning approach greatly simplifies the application performance optimization process. It is easy to use and has the potential to become a standard approach for large number of SCC programmers.**

I. Introduction

Multicore and manycore chips are here to stay, so programmers need to develop parallel software to exploit the hardware potential. Average programmers with little experience in parallel programming now face a spectrum of additional difficulties, such as writing programs that are correct and at the same time perform well on all available parallel platforms. However, already on one single platform, parallel application performance tuning is far from trivial. A key reason is that parallel applications typically depend on a variety of inter-related software parameters, hardware parameters, and input parameters that influence performance in ways that can be too complex to model with acceptable effort. As a consequence, many programmers resort to manual trial-and-error when it comes to tuning (e.g., to determine which degree of parallelism is leads to the best compute performance, which data partitioning has optimal cache exploitation, etc.). This tedious process is typically repeated every time when a program is ported to a new platform.

Intel's experimental Single-chip Cloud Computer (SCC) offers a glimpse into the future on how it would be like to program a manycore computer with 48 general-purpose cores. With an increasing number of cores, hardware needs to adapt and make tradeoffs, which leads to solutions that differ from today's mainstream. This is why the SCC's architecture offers programmers even more freedom than many other multicore architectures, which has significant consequences for programmers. More control in software is good to squeeze out the last percent of performance, but the downside is that application tuning becomes even more difficult than it is already on current shared-memory multicore platforms. For example, every SCC core can boot an own operating system image (where each image might be optimized differently) and communicate via message passing with other cores, just as in a regular distributed system. Communication performance depends, among others, on which cores communicate and how the software assigns work to cores. Inexperienced programmers might be overwhelmed by such details and fail to achieve good performance, while experts might have to invest large amounts of time in performance optimization instead of moving on with other features. Software engineers clearly need automation support for performance optimization to be more productive on the SCC.

We tackle these problems in this paper, which is the first paper to present an application-level automatic performance tuning approach on the SCC. In particular, we make the following novel contributions. We introduce an SCC auto-tuner that automatically searches for good performance configurations for every tunable SCC application. To create tunable applications, we exploit programmer's knowledge and let programmers define application performance parameters that the auto-tuner should configure. For example, parameters may include the maximum number of processes to generate, block sizes for data structures, number of pipeline stages to create, and choices of predefined algorithm variants for a particular program task. Our technique leverages parallelism on higher abstraction levels and complements other performance optimizations (e.g. compiler optimizations). We introduce HyDES, a novel search algorithm based on Nelder-Mead Simplex and Differential Evolution methods, which has been specifically developed and tested for the SCC. We experimentally compare our approach with others using standard benchmarks and show that our approach leads to significantly better performance. Our SCC auto-tuner is able to improve the performance even for parallel benchmark programs that are already optimized with other techniques. Case studies reveal that well-performing configurations can be non-intuitive.

The paper is organized as follows. Section II details the problem specification and our application context. Section III introduces auto-tuning strategies for parallel applications. Section IV presents experimental evaluations on the SCC. Section V contrasts related work. Section VI provides a conclusion.

II. Problem Specification and Context

Our optimization seeks to minimize the application run-time

$$f(\vec{x}), \text{ where } \vec{x} = (x_1, \ldots, x_N)$$

is a vector of tunable application parameters. Each parameter x_i has values out of a predefined valid range $[x_{il}, x_{ih}] \subset \mathbb{R}$.

We assume that developers write programs in such a way that the prograns are configurable, i.e., $\vec{x}$ is configurable from the command line or the program communicates the addresses of configurable variables to the auto-tuner. For now, we assume for simplification that program inputs (e.g., files used in computations) are additional implicit parameters; we treat program tuning with different inputs as own optimization problems. The function f depends on the specific properties of each program and is typically unknown or difficult to model.

The purpose of our auto-tuner is to empirically find good configurations of $\vec{x}$ that minimize the function f, which in our examples corresponds to finding the lowest parallel application run-time. The complete search space consists of the cartesian product of all parameter domains. i.e., $dom(x_1) \times dom(x_2) \times, \ldots, \times dom(x_N)$. Trying out the entire search space is obviously impractical for most application scenarios. Our auto-tuner thus works iteratively to explore the search space: It generates values for parameters using the algorithms described in the next section, executes the programs, measures performance, and uses the feedback to find more promising parameter values. The process stops after a defined number of iterations or if other termination criteria are satisfied (e.g., run-time is below a certain threshold).

A. Homogeneous and Heterogeneous Tuning

For each auto-tuning algorithm, we developed and implemented two versions that distribute the tuning process among SCC cores in different ways.

In *homogeneous tuning* (Figure 1), there is a master core that runs the tuning algorithm, computes the parameter values of $\vec{x}$, and assigns the same $\vec{x}$ to all worker cores. The worker cores then process a program's tasks (which are all split up equally) in parallel. Considering parallel compression with BZIP as an example, all cores would compress blocks of a file using the same file block size. Updates to the block size are performed by the master between complete program runs.

In *heterogeneous tuning* (Figure 2), every core is its own master and computes its own $\vec{x}$. When a core is done, it

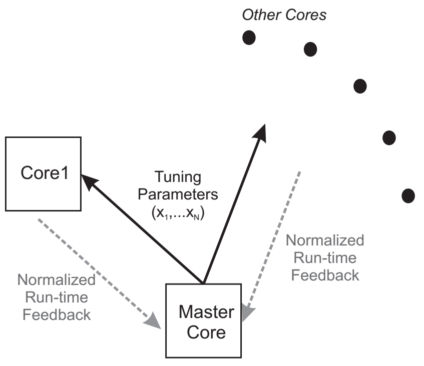

Other Cores
Core1
Tuning Parameters $(x_1, \ldots x_N)$
Normalized Run-time Feedback
Normalized Run-time Feedback
Master Core

Fig. 1. Principles of homogeneous tuning on SCC cores.

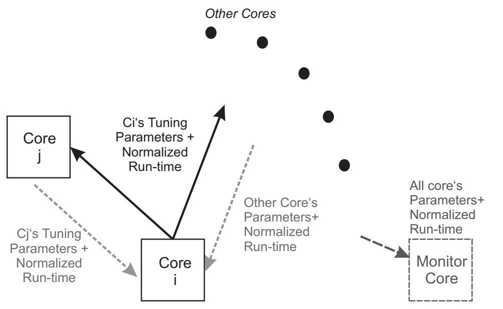

Other Cores
Core j
Ci's Tuning Parameters + Normalized Run-time
Cj's Tuning Parameters + Normalized Run-time
Core i
Other Core's Parameters+ Normalized Run-time
All core's Parameters+ Normalized Run-time
Monitor Core

Fig. 2. Principles of heterogeneous tuning on SCC cores.

broadcasts $\vec{x}$ and the run-time feedback (normalized w.r.t amount of work) to all other cores. When finished, the cores may adapt the new parameters if they are better than their own. A monitor core collects global statistics about the best parameter configuration so far. This approach parallelizes and exchanges information about the optimization process itself and may converge faster. However, it might not be applicable to all sort of programs, e.g., when differing parameter values are not allowed for different workers. This is why we have two versions of tuning, so we can resort to homogeneous tuning if necessary. Heterogeneous tuning for parallel compression, for example, has different cores working on different file blocks, but each core uses different file block sizes; block sizes are adapted while the application is running when new work is assigned to cores.

III. Auto-Tuning Strategies on the SCC

A. ***Random Tuning – A Comparison Baseline***

This strategy serves as a comparison baseline for other tuning algorithms, including ones from the literature. A good tuning algorithm should be able to beat Random on the same number of iterations to justify the implementation effort.

1) Homogeneous tuning: In each iteration, we generate a random vector $\vec{x}$ with elements $x_i \sim U(x_{il}, x_{ih})$ with uniform random distribution. The same vector is used by every worker core. The master keeps the vector that leads to the best application run-time after every tuning iteration.

2) Heterogeneous tuning: Each worker core configures its random vector on its own and logs the best configuration.

When a core completes its work, it sends the best configuration so far to the monitor core. The monitor core keeps the best configuration found by any worker core.

B. Nelder-Mead Simplex

The principle of this algorithm is well-known in the literature [1], [2], [7], which is why we implement it for comparison.

1) Homogeneous tuning: The algorithm has an elaborate scheme with several case differentiations whose details are beyond the scope of this paper, so we briefly sketch the ideas. A simplex $s \in (\mathbb{R}^N)^{N+1}$ is the simplest polygon in N dimensions (e.g., a triangle for $N = 2$). For an N-dimensional search space, the algorithm initializes $N + 1$ vertices with random values. Then, each iteration evaluates the points of the current simplex and moves it to more promising locations. There are several rules to move simplex points, such as reflection, expansion, contraction, and reduction, which are described in [1], [2]. For these rules, our implementation uses the common values of $\alpha = 1$, $\beta = 0.5$, $\gamma = 2$.

2) Heterogeneous tuning: Each worker core runs the entire algorithm on its own and generates a random initial simplex. After a core finishes the optimization, it communicates the best result so far to the monitor core that gathers the best configuration from all cores.

C. Differential Evolution

Literature shows that Nelder-Mead is not powerful enough when it comes to escaping global minima, as it might get trapped in local minima [3]. Differential Evolution is a more promising heuristic that compensates this shortcoming. It works even on non-linear and non-differentiable functions, and it is less computationally intensive than other methods that have the same goal [3]. In each iteration, Differential Evolution works on a population of individuals (i.e., parameter configurations or vectors, respectively), evaluates the best individuals, and replaces individuals in new population generations by using several operators as described next.

1) Homogeneous tuning: Our approach is based on [3] and starts with a population of n_{pop} configurations, where each individual has values randomly chosen from each parameter range. The population size n_{pop} remains constant during the entire optimization process. A mutation operator creates new configurations by taking the difference of two random population vectors, weighing it by a factor $F = \sqrt{\frac{1}{n_{pop}} - \frac{\alpha}{2n_{pop}}}$ [4], and adding the result to a third randomly chosen population vector. To increase diversity, the mutated vector is mixed with yet another randomly chosen population vector v_{target} by taking an element from the mutated vector with probability $\alpha = 0.7$, and from v_{target} with $1 - \alpha$, resulting in the vector v_{trial}. The program to optimize is run with the configuration of v_{target} and v_{trial} on all cores, and the better-performing configuration is kept in the new population, which is used as a starting point in future iterations.

2) Heterogeneous tuning: One or more individuals are represented by one core. Each core works with a different configuration of individuals. After each individual has been evaluated, each core selects the best individual so far and propagates the configuration to all other cores (using `allgather` from `RCCE_comm`). Then, each core generates a new individual as described for homogeneous tuning, and the process restarts.

D. HyDES: Hybrid Differential Evolution Simplex

Our new approach combines the best of Nelder-Mead and Differential Evolution methods. As shown later, it is effective for application-level auto-tuning on the SCC.

1) Homogeneous tuning: As in Differential Evolution, we start with a population of n_{pop} vectors (i.e, configurations). One randomly chosen vector $\vec{p} = (p_1, \ldots, p_N)$ out of this population is used to generate a simplex. We employ the following schema to generate N additional simplex points and ensure that the simplex stretches appropriately in the search space: For every dimension i, we compute $d_i = (max_i - min_i)/4$. If $max_i - p_i \leq p_i - min_i$ then we multiply d_i by -1 to achieve a larger stretching of the simplex. Then, we generate new vectors by adding d_1 to p_1, d_2 to p_2, and so on. So from p, each new simplex vector has a distance of d_i in dimension i. In case that a simplex point is above any max_i or below any min_i of a dimension i, we correct the value in this dimension by moving the vector by a random value into the valid search space. This random displacement is computed using a normal distribution that has its mean at the violated boundary and the standard deviation $(max_i - min_i)/20$. With the resulting simplex, we perform optimization as described by Nelder-Mead, which stops when the maximum distance between any simplex vectors is less than a predefined ϵ. The result replaces the initially selected individual p. Then, Differential Evolution continues with the new population in which the new result is used to generate other vectors as in Section III-C. Nelder-Mead is applied on future populations with 5% probability, otherwise Differential Evolution continues as described.

2) Heterogeneous tuning: Each core is responsible for one individual and initialized by Differential Evolution. Each core communicates its individual to all other cores. As an extension to Figure 2, the monitor core also acts as a master to dictate to all other cores when to apply Nelder-Mead and when to apply Differential Evolution. When Nelder-Mead is applied, the master randomly choses one core to monitor. When that core finishes its Nelder-Mead optimization, the optimization process in all other cores is stopped as well. All cores replace their old individual by the new individual and broadcast their new result to all cores.

IV. Experimental Evaluation on the SCC

A. Setup

We employ the latest available Intel SCC with 48 cores (16KB L1, 256KB L2 cache per core, 16KB message passing buffer per tile). Cores are clocked at 533 MHz, routers at 800 MHz, and memory at 800 MHz. Each core runs an own

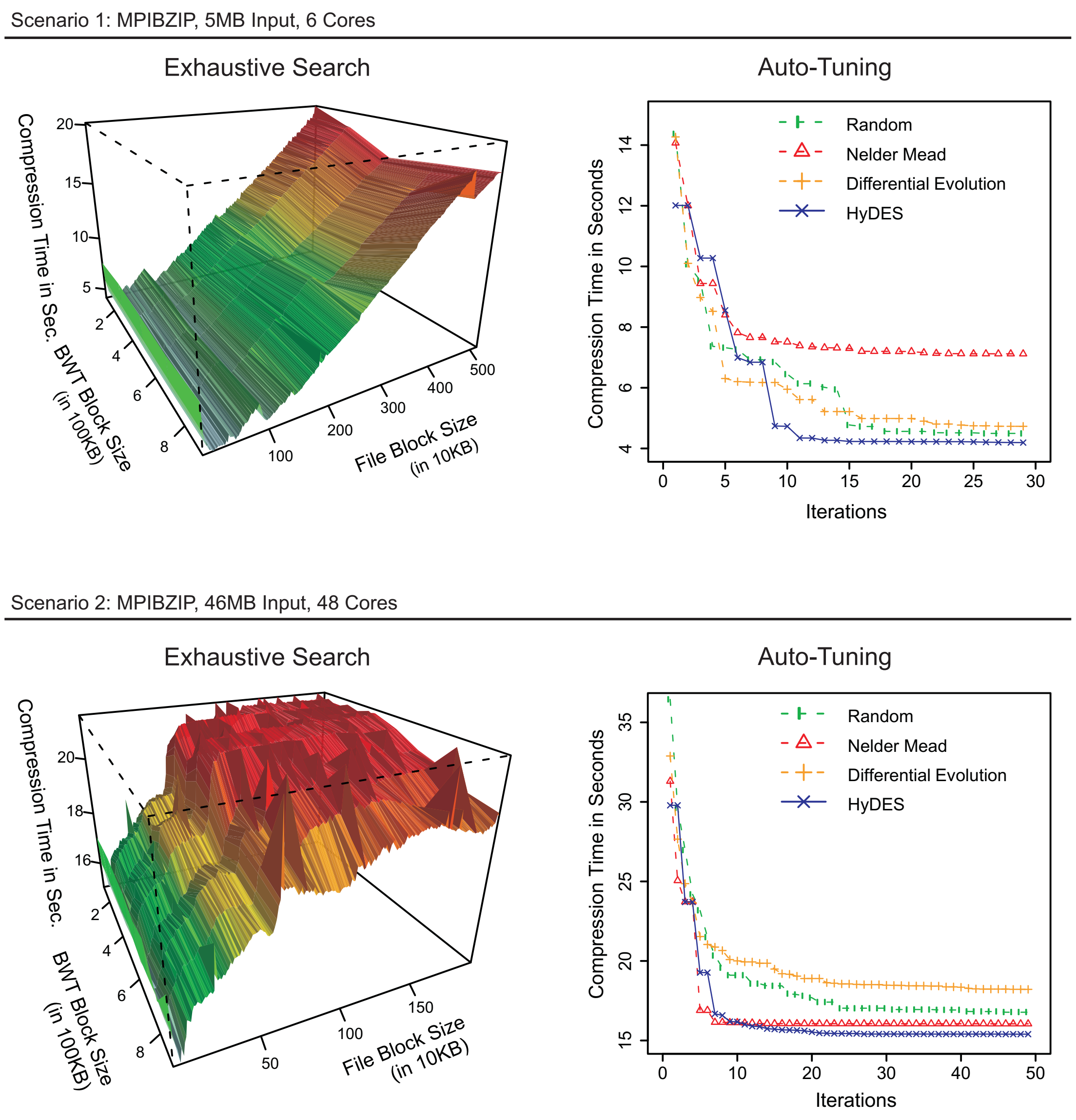


Fig. 3. Comparison of parallel compression tuning results with homogeneous tuning. Lower values are better.

image of Linux kernel 2.6.16. The compilers used are icc 8.1.038 for XHPL benchmarks (with MKL 8.1.1.004) and gcc 3.4.5 for MPIBZIP (with RCKMPI and RCCE). Due to space limitations, however, we can present just an excerpt of all evaluation results.

B. Auto-Tuning SCC Applications

To test our approach on a real application, we made the MPIBZIP compression program configurable for two important parameters (Burrows-Wheeler-Transform (BWT) block size and file block size). Figure 3 shows results with homogeneous tuning which illustrate that our HyDES approach finds the best performance configurations already after 10 program executions. The exhaustive search in the 3D graphs shows that the best performance configuration is non-intuitive; for example, with 6 cores (1 master and 5 slave cores), distributing a 5 MB file intuitively as 1 MB chunks per core is worse than the optimum of 170 KB-sized chunks per core (also found by the auto-tuner). When using a 46 MB input file and all 48 cores, the optimum chunk size per core is 50 KB. Interestingly, MPIBZIP does not even allow users to set such a low value from the command line, so auto-tuning not only paid off, but also pointed to software improvements.

In practice, parameter tuning can be done for clusters of inputs (e.g., ranges of file sizes) when the program is customized to a platform; this yields good parameters that are representative for a class of inputs, so tuning won't have to be repeated for every single input.

C. Stress-Test Evaluations

In addition to real applications, we used common stress-test functions from the optimization literature [5], [6] to evaluate the effectiveness of the aforementioned tuning algorithms in difficult scenarios.

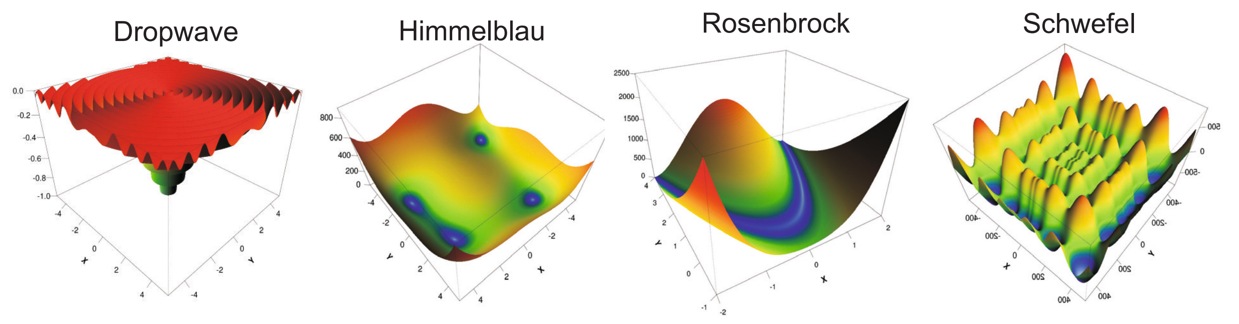


Fig. 4. Common benchmark functions [5], [6] used for optimization stress-testing.

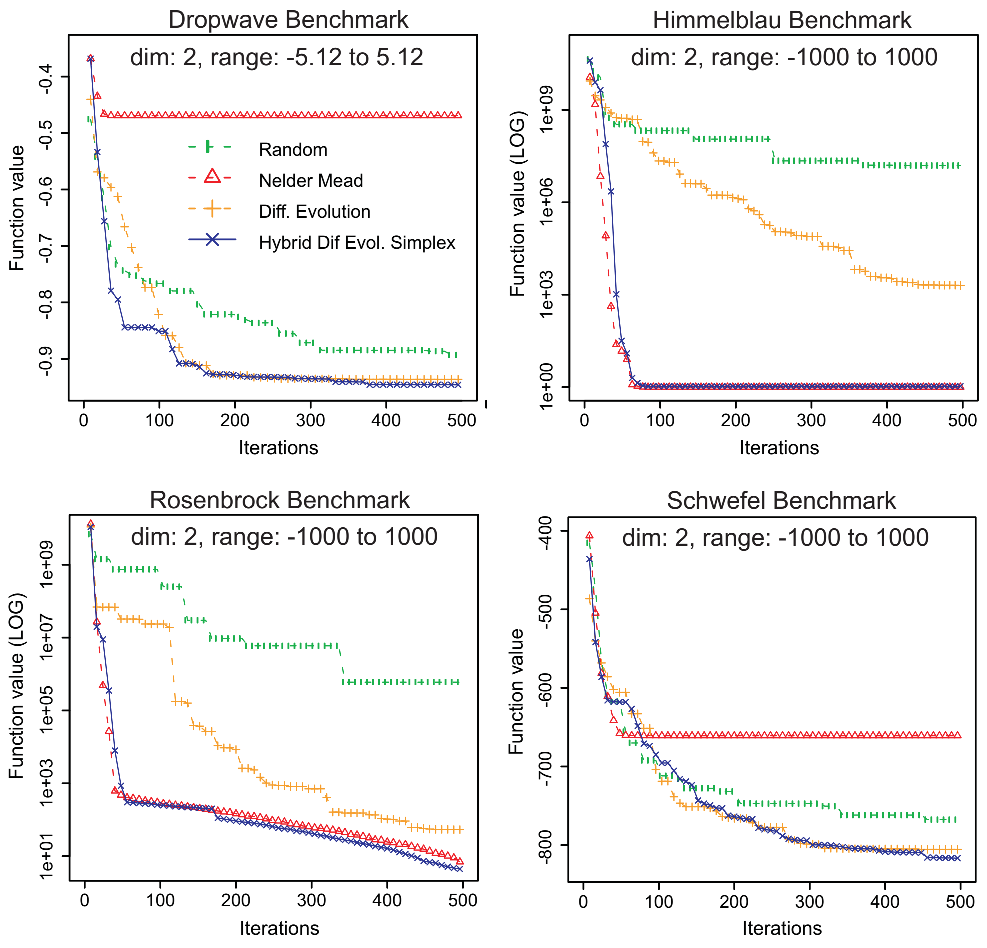


Fig. 5. Tuning results with homogeneous tuning on common optimization stress-test benchmark functions (see Fig. 4). Lower values are better.

Figure 4 exemplifies the shapes of these multidimensional functions for three dimensions. We implemented parameterized SCC test programs whose run-time performance behaves according to these functions.

Figures 5 and 6 show experimental evaluations of the function benchmarks with homogeneous and heterogeneous tuning. It is remarkable that our HyDES approach beats all other approaches in all scenarios in finding better performance configurations, while at the same time the speed of convergence (measured by number of iterations) is good as well.

The homogeneous scenarios in Figure 5 use 20 individuals for evolution. The heterogeneous scenarios in Figure 6 employ a pre-specified number of cores (10, 15, and 48 cores). With heterogeneous tuning, convergence is faster (i.e., fewer iterations are needed) even on a larger problem (4D Rosenbrock). This can be explained by the fact that the heterogeneous tuning approach parallelized the tuning process, as explained earlier.

Note that HyDES combines the best of both worlds: It avoids problems that single Nelder-Mead or Differential Evolution would run into. Sometimes Differential Evolution is

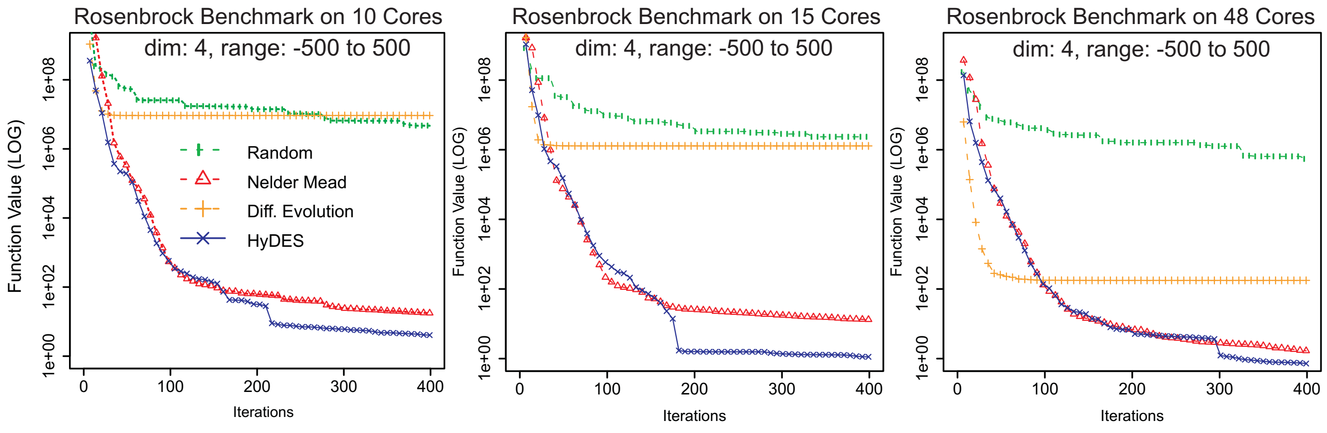


Fig. 6. Rosenbrock benchmark with heterogeneous tuning. Lower values are better.

better than Nelder-Mead or vice-versa, but HyDES always ends up better than any of the latter.

D. *Auto-Tuning a Common Benchmark*

Finally, we also tuned the XHPL (Linpack) benchmark in 11 dimensions (i.e., with 11 tuning parameters). Due its application nature, we employed homogeneous tuning. The results are shown in Figure 7.

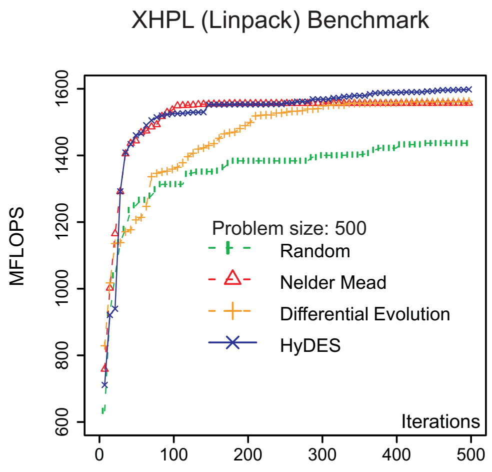


Fig. 7. Auto-Tuning the XHPL (Linpack) benchmark with homogeneous tuning (higher values are better).

The chart demonstrates the superior MFLOPS achievement of our HyDES approach. It also visualizes that systematic tuning is significantly better than using random performance configurations.

V. Related Work

Related work on auto-tuning on the SCC is scarce, but this is because the SCC has appeared just recently and still is an experimental system. It is conceivable to adapt other auto-tuners that work on clusters (e.g., [7]) to the SCC. However, we already include a comparison with [7], as that particular tuner is based on Nelder-Mead optimization. Other tuners that use new optimization approaches in the future can be compared via our random tuning baseline. Most of the literature on optimization [1], [2], [3], [4], [5] focuses on specific classes of algorithms that are inappropriate to use in our SCC context. For example, many algorithms either don't fit to our problem or assume that we can take arbitrarily many samples (potentially infinitely many), which translates to running a program infinitely often. Hybrid algorithms, such as our HyDES, received little attention so far.

VI. Conclusion

Automatic performance tuning makes performance engineering on the SCC less tedious for software developers. As a new aspect, our approach targets general SCC applications, not just scientific numerical applications. The results show for compression and various stress-test applications that our novel tuning approach systematically leads to better parallel performance. Auto-tuning found non-intuitive performance configurations and revealed that existing applications, such as MPIBZIP, didn't even allow users to manually set the best-performing configurations from the command line. Our technique has the potential to become standard for the SCC, as it makes application development and tuning easier.

Remarks and Acknowledgments. The first author is the principal investigator in the "Software Engineering for SCC Parallel Programs" project in collaboration with Intel. We thank Intel for providing us with access to the SCC.

References

[1] R. R. Barton and J. S. Ivey, Jr., "Modifications of the nelder-mead simplex method for stochastic simulation response optimization," in *Proc. IEEE WSC '91*, 1991.
[2] J. A. Nelder and R. Mead, "A simplex method for function minimization" *Computer Journal*, vol. 7, 1965.
[3] R. Storn and K. Price, "Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces" *J. of Global Optimization*, vol. 11, 1997.
[4] D. Zaharie, "Critical values for control parameters of differential evolution algorithm" in *Proc. 8th MENDEL intl. conf. on soft computing*, 2002.
[5] F. Neri and V. Tirronen, "Recent advances in differential evolution: a survey and experimental analysis" *Artificial Int. Rev.*, vol. 33, 2010.
[6] L. Schoeman and A. P. Engelbrecht, "Containing particles inside niches when optimizing multimodal functions" in *Proc. SAICSIT '05*, 2005.
[7] C. Tapus et al. "Active Harmony: Towards Automated Performance Tuning", in *Proc. SC'02*, 2002

SCC Thermal Sensor Characterization and Calibration

Andrea Bartolini, MohammadSadegh Sadri, Francesco Beneventi, Matteo Cacciari, Andrea Tilli, Luca Benini
Email: a.bartolini,mohammadsadegh.sadr2,francesco.beneventi,matteo.cacciari,andrea.tilli,luca.benini@unibo.it
University of Bologna, DEIS
via Risorgimento 2
40136 Bologna, Italy

***Abstract*—Many-cores systems on chip provide the highest performance scaling potential due to the massive parallelism, but they suffer from thermal issues due to the high power densities. Furthermore, workload and process variation requires performance run-time adaptation based on feedback controllers, as open-loop control is not sufficiently robust and accurate.**

The Single-Chip Cloud Computer (SCC) is an experimental processor created by Intel Labs and, as most multi-core prototypes, it integrates thermal sensors to track the thermal behavior of the die. Unfortunately these sensors are not calibrated, preventing the development of thermal management solutions. In this paper we first extensively characterize the SCC thermal sensors and propose a system level technique to calibrate them. Our approach is based on the combination of stress patterns and least-square fitting to extract the thermal sensor characterization directly from the SCC device. Compared to other strategies this method requires only the knowledge of the ambient temperature under the minimum chip power consumption.

I. Introduction

Upcoming many-cores platforms stress the limits of Moore's law. High performance translates in high power densities, that, combined with high spatial parallelism and workload variations, produces non-uniform power dissipation that translates in not-uniform silicon die thermal map. This leads to degradation, acceleration of chip aging and increase in cooling costs. To help designers in studying and counteracting these looming threats, leading silicon manufacturers have started to deliver the many-core prototypes to push researchers toward creating solutions anticipating next-generation product challenges.

As the number of cores integrated on single die increases, an increasing number of accurate thermal sensors is required to reliably maximize performance under a thermal envelope[1], [2]. Unfortunately, due to process variations[3], [4], sensors differ from the nominal ones[5] and thus need to be carefully calibrated before being used. This process requires complex and expensive infrastructures usually not available to the end-users[6].

The Single-Chip Cloud Computer (SCC) experimental processor [7] is a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research. It has 24 dual-core tiles arranged in a 6x4 mesh. Each core is a P54C core. The entire system is controlled by a board management microcontroller (BMC) that initializes and shuts down critical system functions. It is commonly connected by PCI-Express cable to a PC acting as a Management Console (MCPC). Each tile integrates two thermal sensors based on ring oscillators, one positioned in proximity of the router and the other positioned close to the bottom core L1 cache. The BMC includes a power sensor capable of measuring the full SCC chip power consumption.

Unfortunately the built-in thermal sensors are not characterized and thus provide as output only a counter value proportional to the temperature. Due to process variation each sensor under the same operating conditions has different offset and temperature sensibility thus it is hard to use these uncalibrated sensors to drive advanced closed-loop thermal-control policies. Moreover thermal sensors output is also affected by noise, adding another degree of complexity on the usage of them.

In this paper we present a technique to overcome these limitations, based on a combination of stress patterns and least square fitting to extract the thermal sensor characterization directly from the SCC device. This method requires the initial knowledge of only the ambient temperature under the minimum chip power consumption.

The rest of paper is organized as follows: Section II describes the performance and behavior of the SCC thermal sensors. Section III describes the thermal sensor calibration technique. Section IV shows the results and performance of the calibration procedure.

II. Thermal Sensors Characteristics

As highlighted above, the SCC integrates in each tile two built-in thermal sensors. The first thermal sensor is placed close to the router and the second one is placed near the L1 cache of the bottom core. Each thermal sensor is composed of two ring oscillators and the sensor output (TS) is the difference of the two oscillators clock counts over a specific time window t_W. The difference is proportional to the local die temperature(T) [8]. For each tile, the time window (t_W) can be programmed through a per-tile control register[1][9] as a number of tile clock cycle(NCC) $t_W[s] = NCC/f_{TILE}$. Thus it needs to be updated each time the tile frequency changes.

For both the ring oscillators of the thermal sensor we can write:

$$f_1(T) = a_1 + b_1 T, f_2(T) = a_2 + b_2 T \quad (1)$$

[1]CRB Sensor Register (rw).

where f_1 and f_2 are the frequencies of the two ring oscillators and a_1, a_2, b_1 and b_2 are oscillator characteristic parameters.

The counter of each ring oscillator counts a number of cycles:

$$C_1 = f_1(T) \cdot t_W, C_2 = f_2(T) \cdot t_W \qquad (2)$$

The output of each thermal sensor (TS) is

$$TS = C_2 - C_1 = [(a_2 - a_1) + (b_2 - b_1) \cdot T] \cdot t_W = (A + B \cdot T) t_W \qquad (3)$$

where $B < 0$ (TS decreases with the temperature rising) and $A > 0$ (TS is always positive and in the thousands) are different for each sensors. By knowing A_i, B_i for each thermal sensors (i) we can now relate the TS_i with the absolute temperature (T_i) as $T_i = (\frac{TS_i}{t_W} - A_i)/B_i$. The A_i, B_i calibration will be described in the next Section.

Now we focus on set of tests aimed to highlight time window selection issues and its relation with sensibilities. To restrict the source of variation on thermal sensors output we constrained ourself to frequency changes without any change in voltage supply[2].

We first run a power virus[3] in each core while setting different time windows (t_w). For all these configurations we read the thermal sensors values (TS). As the time window increases the output sensor value grows, and this can lead to a counter overflow, as shown in figure 1.

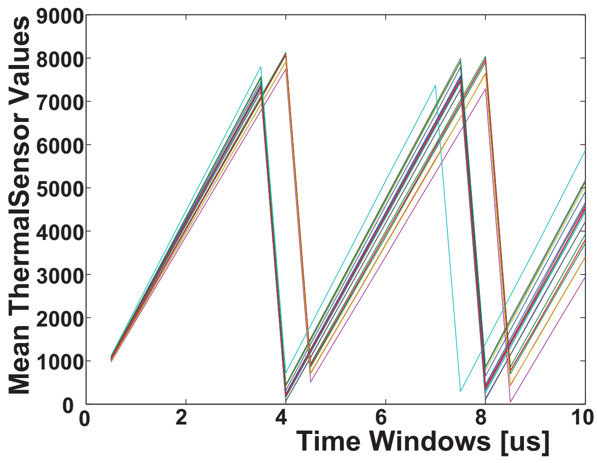


Fig. 1. Sensor output at different t_W (TS/t_W), frequency 533 MHz

The overflow position depends on the ring oscillator frequency thus it depends on the temperature itself. As the frequency increases, die temperature increases and overflows happen at different place. This suggests to set the time window in the safe region, far from overflow points (2,5,10 μs). Eq.4 needs to be updated accordingly.

We perform a second test with the goal of characterizing the impact of the time window on the thermal sensor sensitivity. Intuitively the sensibility should increase as the integrating time of the ring oscillator (t_W) increases. To quantify it we apply after a few seconds a power virus to all the cores and we track the sensor values during the transient.

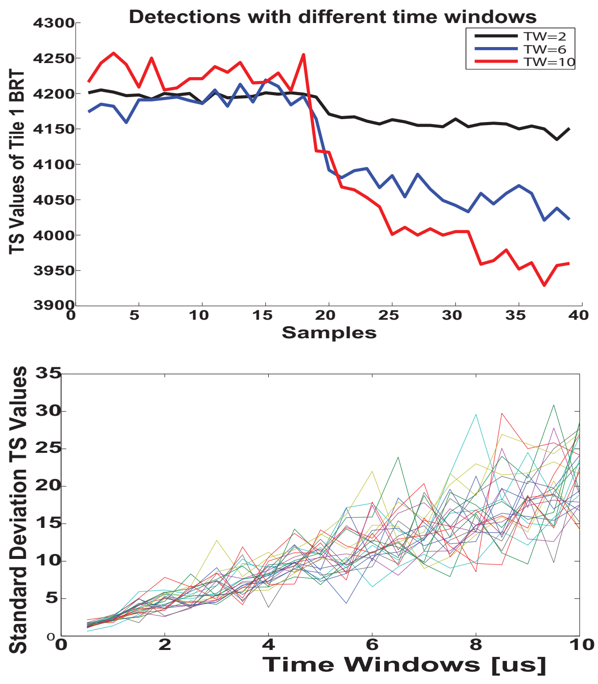


Fig. 2. Sensor sensibility at $t_W = 2, 6, 10 \mu s$ and standard deviation at different time windows

From Fig.2a it is clear that sensor values decrease with the temperature (B<0) and start counting from a positive (A>0) value. From the same figure we can also notice that the TS sensibility increases significantly with the time windows without impairing the dynamical sensor accuracy. Thus we decided to use the 10μs value for all the future tests. Fig. 2b instead shows the standard deviation of each thermal sensor when the thermal transient is ended at different time windows. We can notice that the noise increases along the integrating time of the ring oscillator suggesting the presence of colored noise. Even if the noise amplitude is greater at higher t_W, the Signal to Noise Ratio increases at higher time window.

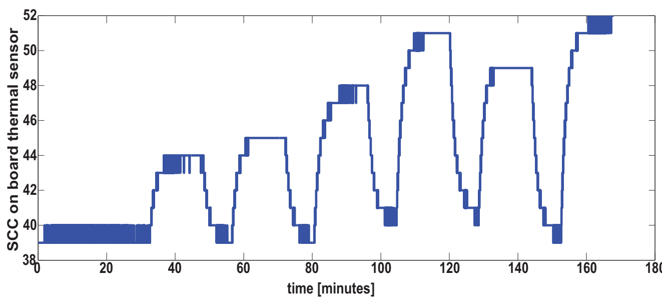


Fig. 3. SCC board thermal sensor output under different SCC chip load.

The SCC platform is provided with a thermal sensor posi-

[2]This applies throughout the paper.

[3]*cpuburn* power virus by Robert Redelmeier: it takes advantage of the internal architecture to maximize the CPU power consumption.

tioned on the motherboard. In Fig.3 we show the changes in its value during a series of SCC stress tests. We can notice that the ambient around SCC is exposed to a significant temperature drift due to the heat dissipated by the SCC chip. This gives us a lower bound for the real thermal cycle happening inside the SCC die. But how to extract the absolute temperature values for each core?

III. Thermal Sensors Calibration methods

In this section we present a technique to link the thermal sensor values with absolute temperature. [8] suggests to use off-chip temperature measurements at references under minimum activity (T_{AMB_MIN}) and maximum activity (T_{AMB_MAX}), and then probe for all the thermal sensors the output value (TS_i) under the two different reference cases (TS_{i_MIN}, TS_{i_MAX}). Then we can characterize each sensor and find the A_{REF_i} and B_{REF_i} for each core by linearly interpolating these two points as described in Eq.4.

$$\begin{cases} A_{REF_i} = TS_{i\_MIN} - TS_{i\_MAX} \cdot \frac{TS_{i\_MAX} - TS_{i\_MIN}}{T_{AMB\_MAX} - T_{AMB\_MIN}} \\ B_{REF_i} = \frac{TS_{i\_MAX} - TS_{i\_MIN}}{T_{AMB\_MAX} - T_{AMB\_MIN}} \end{cases} \tag{4}$$

This approach forces all sensors temperature to be equal to each other both under minimum and maximum SCC utilization. Whereas this approximation is reasonable at lower ambient temperature, it leads to severe approximation errors at maximum utilization since it does not consider spatial temperature gradients caused by the not-uniform thermal dissipation.

To account for this effect we develop a new characterization method that takes advantage of the linear relation between temperature and power (in steady-state condition) and discovers it by linear regression. We first evaluate the relation between SCC power and thermal sensor output while executing different benchmarks and while running all the tiles together at different frequency levels. Fig.4 shows the results, we can recognize that sensor values are linear with frequency regardless of frequencies below 166MHz where a saturation effect is present. This it may be caused by the cooling system. From the same picture we see that power scales linearly with the frequency[4].

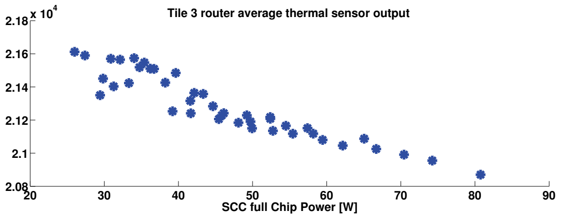


Fig. 4. SCC thermal sensors output vs. full chip power consumption.

Avoiding the saturation region, we can assume the steady temperature to be linear with the core power, $T = K \cdot P$. Thus thermal sensors output can be written as $TS_i = [B_i \cdot K_i \cdot P + A_i] \cdot t_W$. This formula holds when all the cores are stimulated and configured homogeneously. Now we generate a cloud of tuples by stressing all the cores together at different f_{tile}, workloads. With this data we implement a least square algorithm to find out A and $B \cdot K$ for each sensor.

Now we generate a cloud of tuples by stressing all the cores of SCC with different f_{tile}, workloads and t_W. This values are obtained by stressing all the cores With this data we implement a least square algorithm to find out A_i and $B_i \cdot K_i$ for each sensor. Having the A_i, we can estimate B_i by only using one reference point for each thermal sensor (T_{AMB_MIN}, TS_{i_MIN}). We obtain it by reading each thermal sensor and the internal environment temperature using the sensor on the board value at the lowest frequency (100Mhz). We assume that at this configuration, the internal chip temperature is equal to the one we read on the board. We obtain B with the following formula $B_i = \left(\frac{TS_{i\_MIN}}{t_W} - A_i\right) / T_{AMB\_MIN}$.

IV. Experimental Results

In this section we discuss the performance of the presented calibration method. In section IV-A we describe our data collection infrastructure. In section IV-B we show the performance of our sensor calibration technique whereas in section IV-C we make a comparison between characterization results with a Hotspot [10] model of SCC.

A. XTS Framework

To obtain the thermal sensors values we have modified the SCC standard linux image to access at each schedule tick the tile thermal sensors and the per core performance counters. Inside the kernel we implement a double buffer system to save the sensors entries generated for each core at the speed of the scheduler kernel tick. In user space we implement an application that synchronizes through a kernel driver with the internal double buffer system and flushes it as soon one is full. Thanks to this, we can trace sensors and performance counters with a sampling time of 10ms to 1ms[5].

B. Sensors Calibration Results

We first use the A, B calibrated sensors in comparison with the A_{REF_i}, B_{REF_i} calibration described by Eq.4 to evaluate the performance of proposed solution. We used for both of the two calibration methods the same minimum reference point ($T_{AMB\_MIN} = 33^oC$) thus at lower power values they perform similarly. We used as T_{AMB_MAX} the average of the sensors temperature obtained with our approach under full load at 533Mhz. At minimum chip utilization, both calibrations return same temperature distribution. Since they use same reference point at full load, we can appreciate the main difference between the two approaches.

Fig.5 shows the temperature map of SCC under full load with our approach whereas Fig.6 shows the difference with the reference calibration (REF). From both plots we can notice that our calibration is capable of accounting for more realistic thermal gradients.

[4]As early introduced we do not change the voltage supply to avoid variation in the ring oscillator dependency with temperature.

[5]Accordingly to the linux kernel internal configuration and current tile frequency.

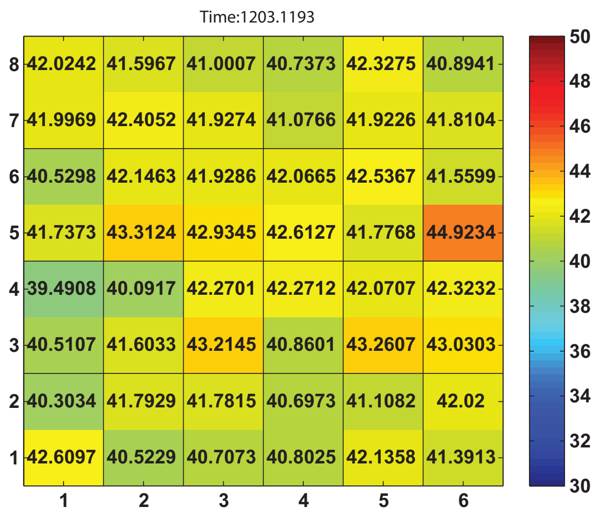


Fig. 5. Distribution of temperature on chip surface under full load at 533Mhz.

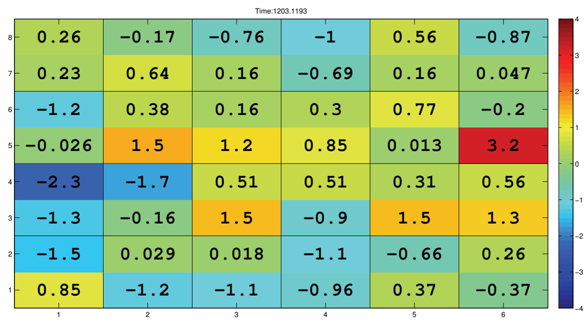


Fig. 6. Difference between our calibration and the reference one under full load at 533Mhz.

Secondly, we use our calibrated thermal sensors to analyze the thermal transient when passing from idle to full chip high-load. Fig. 7 shows the thermal transient for both the sensors of a center tile. All temperature values are represented in blue whereas average signal is in red. In this figure, we can notice that the thermal transient is characterized by different time constants, indeed it looks like to be the response of a system composed by multiple poles with a very long settling tail, probably it is caused by a "partial" pole-zero cancellation.

From the same plot we can see that when we average the noise on the thermal sensor output we loose the information about the fast dynamics in the system thermal response. This suggests that thermal aware dynamic solutions that want to tackle this time scale, need to embed advanced techniques to be tolerant and resilient to sensor input noise.

C. Hotspot Model Comparison

To stress the performance of our calibration results we have developed a thermal model for SCC using HotSpot[10] thermal simulator. This model is capable of predicting SCC chip temperature values at different on-die locations according

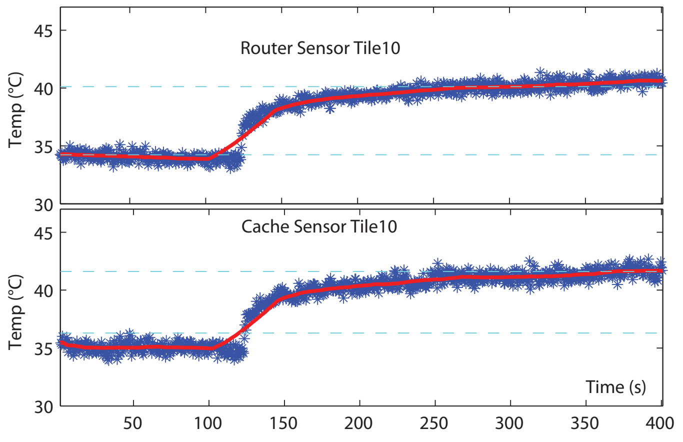


Fig. 7. Step response Zoom

to input power to the chip. Fig 8 shows the results of comparison between HotSpot and measured results when imposing different stress patterns to SCC. These patterns are obtained by running a Power Virus on some of the cores and leaving the other cores idle. Our stress tests cover these 3 cases: (a) Chip is in Idle state; (b) Half of the chip is in full load and half is idle; (c) All of the cores are in full load. The surface in Fig. 8 is the output temperature of the thermal model and bars represent the calibrated sensors outputs with a tolerance range ($\pm 1C$). Fig. 8 d,e,f report the sensors outcome compared to the modeled one. The ticks show when the model output is within the sensor error tolerance whereas the arrows represent when the model output is lower or higher. From the first set of results we notice that the calibrated thermal sensors measurements captures the thermal trends highlighted by the hotspot model. Indeed at full load the center cores show higher temperature than the external ones, whereas in idle the thermal map is more uniform. Regarding the sensor values that have more significant mismatch w.r.t. model predictions, we can see different behaviors. One sensor is constantly outliers, suggesting the presence of a HW fault or low performance in the thermal characterization. The corner cases a,c show good matching, with a percentage of outliers below 18%.

V. Conclusions

In this paper we have presented a study of the SCC thermal sensors performance and behavior. Our results highlight the presence of overflows on the output values and significant spatial and temporal noise. To manage these non uniformities in uncalibrated sensors we developed a novel procedure to extract the relationship between thermal sensors output and absolute temperature using data-regression. We then used our calibration to highlight the thermal performance of SCC. We compared it with Hotspot thermal simulation of a floorplan similar to SCC and we highlighted rooms for improvement in the presented thermal calibration techniques. We are currently working to improve the presented technique by gathering more data on different SCC HW platforms.

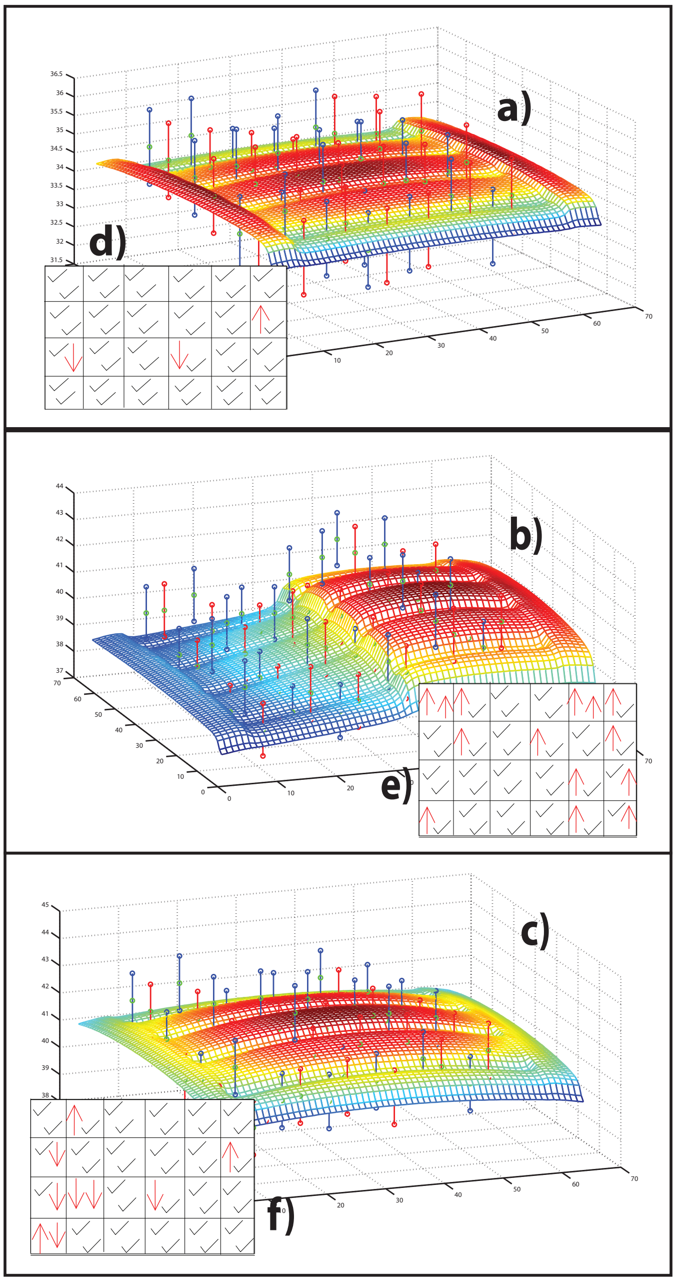


Fig. 8. Comparison between HotSpot measured temperatures, From left to right: chip full load, half chip full load, and chip idle

VI. Acknowledgements

This work was supported, in parts, by Intel Corp., Intel Labs Braunschweig and the EU FP7 Projects Pro3D (GA n. 248776) and Therminator (GA n. 248603).

References

[1] Bartolini, A., Cacciari, M., Tilli, A., Benini, L., "A distributed and self-calibrating model-predictive controller for energy and thermal management of high-performance multicores." Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011 , vol., no., pp.1-6, 14-18 March 2011

[2] Shervin Sharifi and Tajana Simunic Rosing. "Accurate direct and indirect on-chip temperature sensing for efficient dynamic thermal management." Trans. Comp.-Aided Des. Integ. Cir. Sys. 29, 10 (October 2010), 1586-1599. DOI=10.1109/TCAD.2010.2061310 http://dx.doi.org/10.1109/TCAD.2010.2061310

[3] Eric Humenay, David Tarjan, and Kevin Skadron. "Impact of process variations on multicore performance symmetry." In Proceedings of the conference on Design, automation and test in Europe (DATE '07). EDA Consortium, San Jose, CA, USA, 1653-1658.

[4] Paterna, F., Acquaviva, A., Caprara, A., Papariello, F., Desoli, G., Benini, L., "An efficient on-line task allocation algorithm for QoS and energy efficiency in multicore multimedia platforms." Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011 , vol., no., pp.1-6, 14-18 March 2011

[5] Spandana Remarsu and Sandip Kundu. "On process variation tolerant low cost thermal sensor design in 32nm CMOS technology." In Proceedings of the 19th ACM Great Lakes symposium on VLSI (GLSVLSI '09). ACM, New York, NY, USA, 487-492. DOI=10.1145/1531542.1531653 http://doi.acm.org/10.1145/1531542.1531653

[6] IBM "Calibrating the Thermal Assist Unit in the IBM25PPC750L Processors" PowerPC Embedded Processors Application Note. October 6, 2001.

[7] Howard, J.; and others, "48-Core IA-32 message-passing processor with DVFS in 45nm CMOS", Solid-State Circuits Conference (ISSCC), 2010

[8] Intel Labs "Using the Sensor Registers", Revision 1.1, available at: http://communities.intel.com/community/marc

[9] Intel Labs "SCC External Architecture Specification (EAS)", Revision 1.1, available at: http://communities.intel.com/community/marc

[10] Wei Huang, Stan, M.R., Skadron, K., "Parameterized physical compact thermal modeling." Components and Packaging Technologies, IEEE Transactions on , vol.28, no.4, pp. 615- 622, Dec. 2005 doi: 10.1109/TCAPT.2005.859737

Efficient Memory Copy Operations on the 48-core Intel SCC Processor

Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck and Chris R. Jesshope
Informatics Institute, University of Amsterdam
Sciencepark 904, 1098 XH Amsterdam, The Netherlands

Abstract—**The Single-chip Cloud Computer (SCC) is a 48-core experimental processor created by Intel Labs targeting the many-core research community. It has hardware support for sending short messages between cores, while large messages have to go through off-chip shared memory. However, memory copy operations on this chip are expensive and inefficient. In this paper we provide insight in the SCC's memory architecture and describe and evaluate a few memory copy methods. We propose a novel method, unique to the SCC, which we believe achieves the maximum possible throughput for a single core on this chip. In order to efficiently implement this approach we introduce dedicated cores that run a memory copy service which can be used asynchronously by other cores.**

I. Introduction

The Single-chip Cloud Computer (SCC) experimental processor [1] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. It provides an on-chip message passing network, a non cache-coherent off-chip shared memory and dynamic frequency and voltage scaling. We are investigating possible implementations on this platform of SVP [2], a hierarchical concurrent execution model, and S-Net [3], an asynchronous stream processing coordination language. The dataflow-style execution properties of both models would provide us with a handle for adaptive power management.

As the SCC effectively is an on-chip distributed system, we can already run the two available distributed implementations [4], [5] of the models without any modification. As these are based on more coarse grained communication primitives such as TCP/IP sockets and MPI, we plan to rewrite them to optimally use the hardware messaging support on the SCC. However, the on-chip message passing buffers are only efficient for relatively small messages; up to 8KB using RCCE [6], or 128KB using the pipelined iRCCE [7] approach. This is sufficient for many message passing programs that only need to communicate small updates on every iteration, but in SVP and S-Net we potentially move a lot more data around between cores. Therefore we have investigated efficient ways to copy large pieces of data on the SCC.

In this paper we make an analysis of several approaches to implement efficient memory copy operations between cores on the SCC. We do this by first giving an overview of its relevant hardware features and performance properties (Section II), then we discuss existing communication libraries and several approaches using different memory access methods in Section III. We present a novel method to copy memory, unique to the SCC, in Section IV. We believe that this method achieves the highest possible throughput for a single core copying blocks of memory larger than 256KB. This requires dedicated *copy cores* which support copy offloading that we discuss in Section V. We conclude with comparing the discussed approaches and our experience with the SCC platform in Section VI.

II. SCC platform

The SCC is a single chip with 48 Intel IA-32 P54C cores connected by an on-chip mesh network which has a 256 GB/s bisection bandwidth [8]. Its features are intended to allow CPUs scaling up to hundreds and potentially thousands of cores. The mesh is organized as six voltage islands containing four tiles with two cores per tile, creating a 6x4 mesh with 24 tiles total. Each tile has its own mesh router to access the mesh for memory access and inter-core communication.

The basic communication paradigm for the SCC is message passing, and therefore each tile has a local 16KB Message Passing Buffer (MPB). The MPB is suitable for sending short messages between cores. A special library (RCCE) to easily use the MPBs to send/receive messages is available, as well as an MPI channel implementation. With two cores per tile, each core has an 8KB area in the local MPB only by convention, as all MPBs are memory mapped and can be addressed directly by each core.

The chip features extensive frequency and voltage control on a per tile and voltage island basis. For all measurements in this paper the cores were clocked at 533 MHz, and the mesh network and memory controllers at 800 MHz.

A. Caches

Each core has both a 16KB L1 data and an instruction cache which cache the complete address space, including the MPBs. Therefore an extra memory type for MPB data (MPBT) was added to the virtual memory system, together with an instruction to invalidate all cachelines in the L1 D-cache that are flagged with this type. As the P54C core originally only supports a single outstanding write request, a Write Combine Buffer (WCB) has been added to combine adjacent writes up to a whole cacheline which can then be written back at once. However, this is only used for MPBT flagged writes.

Each SCC core has a private unified 256KB L2 cache which does not feature a cache coherency protocol. Also, in contrast to the L1 cache, there is no native way to flush or invalidate the

L2 cache; the WBINVD/INVD instructions that can be used to flush or invalidate L1 do not affect the L2 cache. To solve the cacheability issues, users can turn off the L2 cache on a per-core basis. It is also possible to set the cacheability for each individual virtual memory page. This can be done with the standard PCD cache disable flag to disable both caches, or with the MPBT flag to bypass L2 and use the WCB. The L2 cache can be reset separately from the core using a special control register, which initializes all lines into invalid state. However, this operation halts the core and therefore can not be used to invalidate the cache during execution.

Both the L1 and the L2 are 4-way set associative with a cacheline size of 32 bytes, are *write back*, and do not allocate on write miss, i.e. are *write around*.

B. *Memory structure and look-up tables*

The individual P54C cores have a 32 bit core-physical address space, while the chip has four DDR3 memory controllers which each use 34 bits addressing. Combining the 34 bits with the memory controller address, the system can address 64GB in total. The translation of core-physical addresses to system-physical addresses is done through look-up tables (LUTs). Each core has a private LUT with 256 entries, where each entry covers 16MB of the 4GB core-physical address space, which we refer to as a *LUT page*.

The translation is done by indexing the LUT with the highest 8 bits of a core-physical address, and then extending it to form a 34 bit address and adding routing information for the mesh network. An entry can map to a special memory anywhere on the chip or to an addresses on any of the four memory controllers. The MPB and System Configuration Registers of each tile are examples of such special on-chip memories, but also the LUT itself. The LUTs are usually set up when booting a core, but can be changed dynamically when the system is running, having effect immediately. This allows sharing data between cores without having to copy it. A core can map system-physical memory used by any other core at the granularity of these 16MB LUT pages.

C. *Memory subsystem properties*

In the standard configuration, each memory controller runs at 800 MHz, corresponding with a theoretical peak transfer rate of 6.4 GB/s. Figure 1 shows the measurements of the maximum memory throughput for a single core. This benchmark reads or writes data in 4-byte operations from/to memory areas ranging in size from 8KB to 2MB, where each area is prefetched before every measurement. This shows a reading and writing bandwidth around 400 MB/s for the L1 cache for size 8-16KB, 285 MB/s for reading the L2 cache at size 32-256KB and 107 MB/s for reading external memory. Writes to the L2 cache perform at 116 MB/s, or 130 MB/s when using the write-combine buffer (WCB) by flagging the data as MPBT. Writing to external memory shows the real benefit of the WCB where MPBT flagged data is written at 125 MB/s against 22 MB/s for non-MPBT writes. MPBT tagged reads do not benefit from the L2 cache as it is bypassed, but therefore perform slightly better than non MPBT writes to memory.

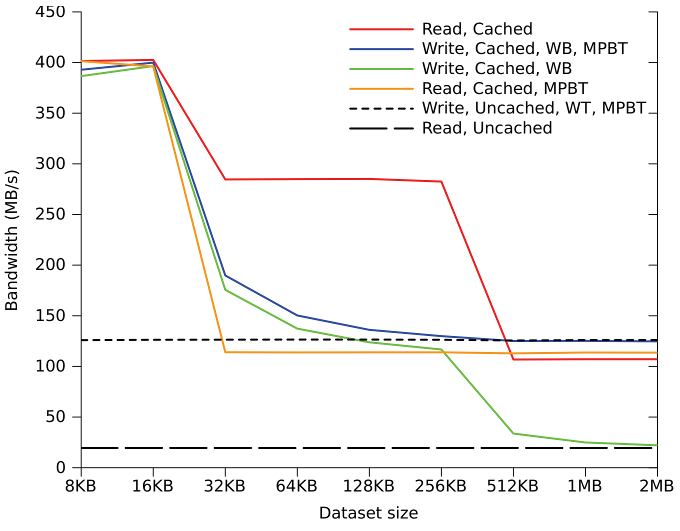


Figure 1. Memory throughput benchmark result of one core reading and writing prefetched memory areas of several sizes, showing the effect of different memory flags.

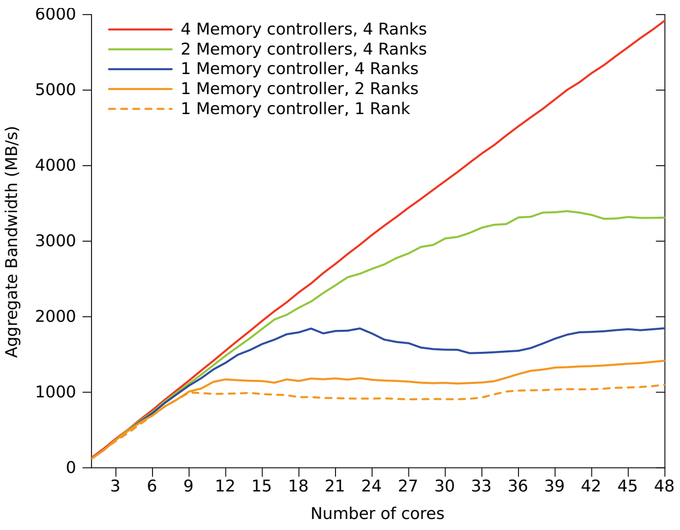


Figure 2. Memory throughput benchmark result showing the aggregate memory bandwidth of 1 to 48 cores performing memory reads divided over a different number of memory controllers and ranks.

The results in Figure 1 show us that a single core can get nowhere near saturating a memory controller. Figure 2 shows how many cores are required to saturate one or more memory controllers, or conversely, how many cores can access the same controller without a large impact on performance. Each memory controller has two banks that each consist of two ranks, and requests to different banks and ranks are interleaved [9]. However, the controller does not interleave the address space between the four ranks. The default LUT mappings map the main memory address space of a core to a contiguous physical address range on a single controller, mapping it to only a single rank. Rank and bank conflicts impact memory performance, so we measured what it takes to saturate a rank, a bank, the whole controller, two controllers, and all four controllers.

Figure 2 shows us that a single rank is saturated by 9 cores

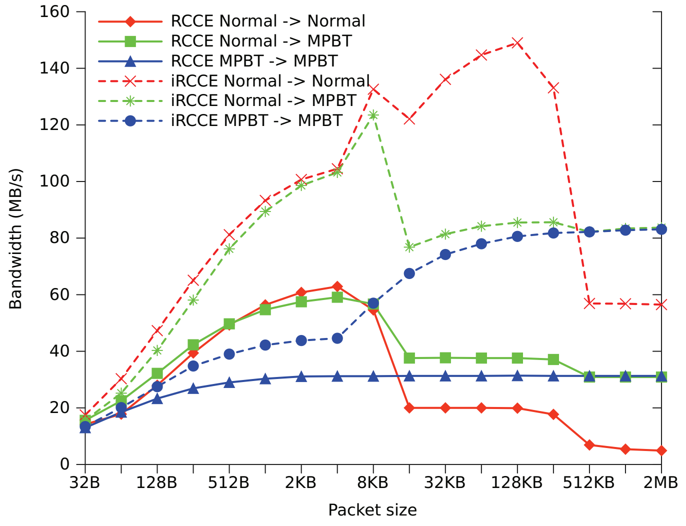


Figure 3. Pingpong benchmark results using the RCCE and iRCCE communication libraries for a range of packet sizes, showing the effect of different memory flags on send and receive buffers.

at an aggregated bandwidth of 1.0 GB/s. A whole controller saturates at 19 cores delivering 1.8 GB/s, two controllers at 40 cores with 3.4 GB/s and all the 48 cores together are unable to saturate the four memory controllers, scaling linearly to deliver a peak of 5.9 GB/s.

III. Memory Copy Operations

A. RCCE and iRCCE

The initial intuitive approach to copy data from one core to another is to use the on chip message passing buffers. This can be done by using the supplied RCCE [10], [6] framework. However, as we can see in [6], as well as in Figure 3 where this is shown as *RCCE normal->normal*, it has its peak performance at 60 MB/s only for 4KB messages. For larger messages up to 256KB it drops to around 20 MB/s and for even larger messages the performance collapses to 5 MB/s.

iRCCE [7] was developed by RWTH Aachen to improve RCCE performance. It uses pipelining when sending messages larger than the MPB so that the read/write operations do not happen in lockstep, and prefetches the target addresses into the L2 cache so that data is not suffering from *write around* for every 4 bytes. A specialized memcpy function is used to copy data to and from the MPBs. This improves throughput a factor of two compared to RCCE, and with messages larger than 4KB the pipelining gives an even greater advantage. At 128KB messages the peak performance is reached, around 145 MB/s. However, as soon as messages are larger than the L2 cache, the performance drops to 60 MB/s. Figure 3 shows these results in the plot as *iRCCE normal->normal*.

Section II-C showed us that using MPBT flagged memory accesses can improve throughput, therefore we ran two more experiments with RCCE and iRCCE. In the first experiment we flag the receive buffer as MPBT type memory, and in the second both the send and the receive buffers. Figure 3 shows that this improves the performance for iRCCE messages larger than 256KB to a throughput of 84.9 MB/s, and improves the throughput of standard RCCE from 5 to 31 MB/s. As MPBT flagged operations bypass the L2 cache, it shows worse performance in our measurements than the normal RCCE/iRCCE for messages between 4KB and 256KB. However, this is a red herring; the pingpong test sends the same message back and forth many times, effectively measuring the throughput when the message data is already present in the cache. If the data is not in the cache, the MPBT modification would outperform normal iRCCE, even for small messages, similarly as it does for messages larger than 256KB.

Another issue with these message passing approaches is that it keeps two cores busy to copy data while they can not perform any computation. This is less of an issue in message passing based SPMD programming paradigms, such as MPI, where usually all processes alternate between a computation and communication step, but it is inefficient for our dataflow style approaches. In our case it is more beneficial for the sending core to copy the data into a shared memory location and then asynchronously signal the receiving core that the data is ready to be used. In the meantime, the receiving core can potentially do other useful work without being tied up in the copy process. Furthermore, when we are sending large messages, data comes from memory, goes through the MPB and is then written back to memory again by another core. Therefore, we expect that we can be more efficient with a direct memory to memory copy.

B. Optimizing memcpy

Naive memory copy operations with the standard *memcpy* function performs very poorly on the SCC at only 17.4 MB/s. This is because it does not take the cache hierarchy into account which has a *write around* policy on a write miss. When copying data to a new location, it is not likely to be in the cache, which means that every 4 byte chunk is individually written to memory instead of whole cachelines of 32 bytes at a time. Also, the read data is unnecessarily cached in both L1 and L2 cache, while only the spatial locality of a single cacheline will help for the copy throughput, as the SCC does not use prefetching.

To improve memory write performance, the SCC has the WCB but this is only enabled for MPBT flagged data. As this can be set per virtual memory page it can be used to force normal memory writes to use the WCB. By applying this to the target buffer we achieved a dramatic improvement in throughput up to 69.4 MB/s. Besides the largest advantage that comes from having a single 32 byte burst instead of 8 individual 4 byte writes on the memory bus, this also removes the delay of accessing (and then writing around as it is a miss) the L2 cache. The latter inspired us to a further optimization; by flagging the input buffer as MPBT as well, the data is loaded directly into the L1 cache. This still gives us the locality advantage for subsequent reads, but removes the delay of allocating in L2, then moving and allocating in L1. Using this approach we measured a throughput of up to 70.9 MB/s.

We also investigated the optimized memcpy implementation that was developed for iRCCE. It moves data through two registers taking advantage of the dual-issue pipeline in the

P54C core, in contrast to the standard memcpy which uses the special repetition prefix and 4 byte copy instruction. This apparently has its advantages when copying data into the MPB, but not when copying from memory to memory. In the best case, again achieved by flagging the source and target memory as MPBT, it reaches a throughput of 49.7 MB/s.

C. L2 Cache flush

An important issue when communicating through shared memory is the L2 cache. As the SCC does not have support for flushing or invalidating the L2 cache, care has to be taken when the receiver wants to use the data copied by the sender. There might be address conflicts in the L1 and L2 cache causing the receiver to read stale data. This is not a problem for the L1 as it can be flushed with an instruction, but the L2 can only be flushed by making sure that the contents of the entire cache are replaced, i.e. reading in 256KB of *clean* data. This is not an issue on the sending side, assuming we use the MPBT based memcpy method; MPBT flagged data does not go to the L2 cache, and even the L1 cache can be turned off for the target addresses at as good as no performance hit as shown earlier in Section II-C. Of course the sender needs to make sure that the data it wants to send is not in dirty state in its own L2 cache, so it would require a flush before starting the copy operation, unless it can be absolutely certain that it can not be in dirty state in its cache. Note that the L1 cache can be used as write through, but the L2 cache can only operate as write back.

We have worked on optimizing the L2 cache flush routine by using on-tile MPB mapped addresses to flush the cache. As the MPB is mapped into a single LUT-page, it only occupies a small portion of the 16MB address space. Memory accesses beyond the MPB within the LUT-page wrap around back into the MPB as the logic probably ignores the higher address bits. However, from the perspective of the L2 cache, these are still unique addresses, so they can be used to flush the cache. By using a 256KB region beyond the MPB to avoid interference with normal MPB accesses, we improved the flush operation from over 1 million cycles to around 580K cycles. Of course this is still a heavy impact on performance, as this takes slightly more then 1 ms.

IV. LUT Based Copy

The disadvantage of both the message passing and memcpy approaches discussed in the previous section is that all data needs to go through the core, being read and written in 4 byte quantities. The L1 cache and the WCB alleviates the problem slightly so that beyond that point only 32-byte cachelines are transferred, but still 8 read and 8 write operations have to be performed individually by the pipeline of the core to transfer the contents of each line.

The SCC has a unique feature with the programmable LUTs which do a second layer of address translation outside the boundary of each core. As this happens transparently to the core, this property can be exploited to efficiently duplicate large blocks of memory. If two cores need to share data and it is guaranteed to be used read-only, the receiving core can simply map the memory anywhere (on a 16MB granularity) into its address space by adding the correct LUT entries and the sender only needs to make sure that the off-chip memory is up to date with a cache flush.

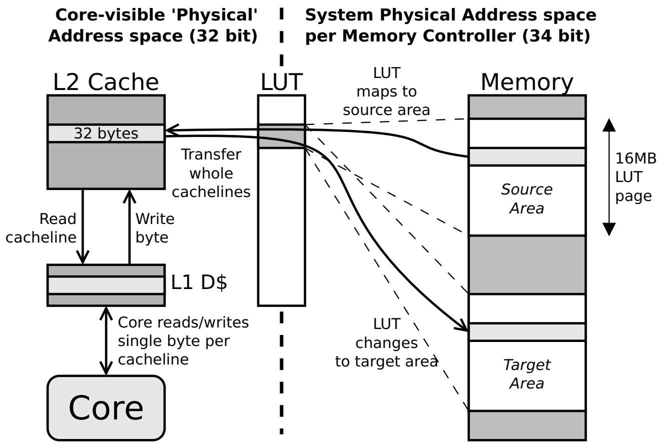

Core-visible 'Physical' Address space (32 bit)
System Physical Address space per Memory Controller (34 bit)
L2 Cache
LUT
Memory
LUT maps to source area
32 bytes
Transfer whole cachelines
16MB LUT page
Source Area
Read cacheline
Write byte
L1 D$
Core reads/writes single byte per cacheline
LUT changes to target area
Target Area
Core

Figure 4. Memory copy operation using L2 cache and LUT remapping

Memory duplication of large blocks using the programmable LUTs works as follows, and is illustrated in Figure 4; the core reads in the first block of 256KB into the L2 cache. It only needs to read a single byte per cacheline to fill the entire cache. While doing this, it makes sure that every cacheline is put in modified state by writing each byte it reads back again to the L2 cache. The content of the data is unchanged, but the L2 is unable to tell the difference. After the cache is filled with modified lines, the core switches the LUT entry for that core-physical address range to map to the destination range in the system-physical address space. The core-physical addresses that are present in the L2 cache now map onto other system-physical addresses. Data is then pushed out of the cache, for example with a cache flush, and is therefore copied to the target location.

When a multiple of 256KB blocks needs to be transferred, the copy procedure can be pipelined; Using a second address range in the core-physical address space (i.e. another LUT entry) the next block of 256KB is read in while it pushes out the previous block to the target address range instead of the cache flush. This needs to be a second core-physical range as the first range is still used to push data out, and can therefore not yet be reused. For the block after that, the first entry can be used again, and so on, alternatingly using the two LUT entries until all data has been copied.

We believe that this method achieves the highest possible throughput for memory copy operations on a single core, which we measured at 73 MB/s for large blocks. Similarly to memcpy using MPBT flagged data only whole cachelines are transferred from and to the memory controllers, most effectively using the powerful on-chip network [8]. The advantage it has over the memcpy approach is that only a single byte per cacheline needs to pass through the core itself, resulting in less read and write operations to copy a cacheline. Still a whole cacheline is transferred between L2 and L1, but by using L1 in write-through mode, only a single byte is written back from L1 to L2. Unfortunately the L1 can not be bypassed completely.

The proposed approach has a few downsides. First of all the target address needs to be aligned at the same offset within a 16MB LUT page within the system-physical address range, though it does not have to be on the same memory controller. This restriction can be compensated somewhat by the use of virtual memory mappings, but as these map using a 4KB page granularity, you still have to use the same offset within such a 4KB page. A second issue is that both the source and target areas need to be contiguous blocks in physical memory, or at least at a granularity of the L2 cache which is 256KB. Otherwise, conflicts would happen and as a side effect data outside the source area would be copied as well as. A third disadvantage is that care has to be taken that the L2 cache is not influenced during a copy operation. This can be done by using the MPBT flags to bypass L2 for all other memory used by the program to avoid interference, however this impacts performance. To keep this manageable, we can use dedicated cores to which we delegate these memory copy operations.

V. Dedicated Copy Cores

We introduce dedicated *copy cores* that run a memory copy service. These cores can be asynchronously messaged by writing meta-data into their MPB and sending them an interrupt. This uses a mailbox protocol similar to what the Barrelfish developers proposed in their report [11]. A message tells a copy core to initiate a memory copy operation of a given size between two addresses, and to send a notification on completion, but not necessarily back to the requesting core. As this requires minimal functionality, it can be implemented as a small efficient kernel running directly on the *bare metal* hardware, likely even fitting completely in the 16KB I-cache. This makes it very suitable for our LUT based copy approach that we just described, as it will have no cache interactions. Furthermore, it has the advantage that we can completely control the virtual memory types used, and therefore guarantee that for any memory interaction required to run the code, the L2 cache is bypassed and therefore remains untouched. A second advantage is that such a kernel does not suffer from the penalty of around 2K cycles to switch between kernel and user mode on receiving interrupts, making message delivery much cheaper taking around 600 cycles [11]. These cores are not limited to using the LUT based copy approach, but can implement any memcpy method discussed previously.

In an SPMD setting, these copy cores are probably not of much use. In most cases, it would be better for the performance to have the core participate in the computation than to have it copy memory for other cores. However, this is not the case in our dataflow style runtimes. Work might not be divided as evenly as with SPMD, and in order to make more progress at a critical point, having the aid of another core to copy memory can be very beneficial. For example, if core A requires a copy of a range of memory in order to start a new computation on core B, it can ask copy core C to copy in the background while A continues working, and C can notify B directly when it is done. Furthermore, if a large amount of data needs to be transferred, A could split the range and employ more than a single copy core to copy the data, exploiting data parallelism.

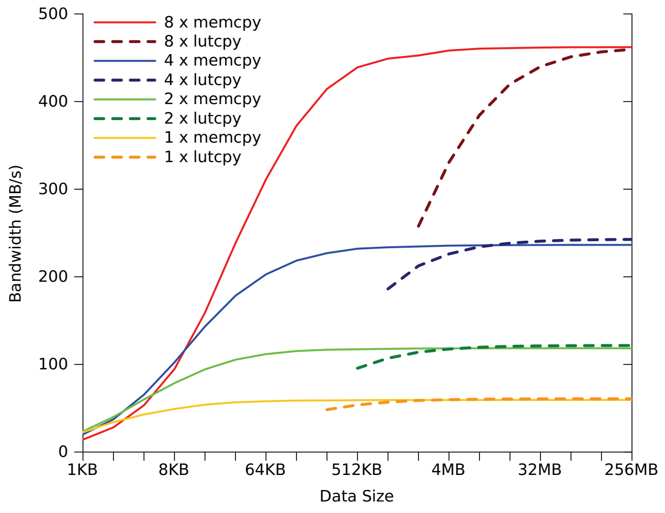


Figure 5. Memory copy throughput benchmark using 1 to 8 *copy cores* to copy data using either MPBT flagged *memcpy* or the LUT based *lutcpy* copy method.

This then delivers more throughput, as a single core can get nowhere near saturating the bandwidth of a mesh network link or a memory controller. We measured that it takes 9 cores to saturate a single bank, single rank on a memory controller.

A. Benchmark

Figure 5 presents some preliminary results of a copy core implementation. This implementation is currently still running under SCC Linux and uses polling instead of interrupts to receive messages, but it gives us an initial idea of the results that can be achieved with the techniques that we have described. Our implementation supports both the LUT based copy operation, *lutcpy*, and a *memcpy* operation that uses MPBT flagged source and destination areas which delivers the best memory throughput as discussed in Section II-C. It should be noted that the LUT based copy method will not reliably copy all the data in this environment, but the measurements will still show the correct performance characteristics. Furthermore, *lutcpy* can only be used to copy data larger than 256KB per copy core.

The results in Figure 5 show that *lutcpy* is only marginally faster than *memcpy* on very large copy operations. *memcpy* outperforms *lutcpy* because it uses uncached MPBT flagged memory for its target address range, which means it does not have to perform an expensive L2 cache flush at the end of the operation. For larger copy operations this becomes a less dominant factor for *lutcpy*, and then it slightly outperforms *memcpy* in the way originally expected.

The most important result of our copy core benchmark is that the technique scales very well. Even for very small piece of data, such as 4KB, the copy operation benefits from being split across two copy cores. However, this would still be slower than performing the operation locally due to the messaging latency. The advantage starts at 8KB, where two copy cores together deliver 78.7 MB/s, that is including messaging overhead, whereas the requesting core itself would only be able to copy at 70.9 MB/s, see Section III-B.

VI. Conclusion

In this paper we have surveyed several options for efficiently copying memory on the Intel SCC. Using the RCCE or iRCCE message passing implementations has the advantage that it does not require cache flushes, but has the disadvantage that two cores are occupied in the copy process. It originally has a low throughput for regions that are larger than the MPB and/or L2 caches. We then showed how the standard *memcpy* operation can be improved a four-fold by enabling the use of the WCB with MPBT flags on virtual memory pages, and that this also improves message passing performance.

We proposed a novel approach to copy memory, unique to the SCC platform, by switching LUT entries to copy data with the L2 cache using less interaction with the core. It performed 3% faster than the fastest memcpy method in our initial results, and we argued that this achieves the highest possible throughput for a single P54C core. The disadvantage of this approach is that it is cumbersome to use, with restricted alignments and sizes for the data that is copied.

Complementary to the LUT based copy method, but orthogonally to the improved memcpy approach, we proposed the introduction of *copy cores* to be able to asynchronously offload copy operations, similar to DMA engines. We then showed the benchmark results of a preliminary copy core implementation, comparing MPBT flagged memcpy and LUT based copy. Offloading to copy cores scales very well, but the LUT based copy performance was not as good as we expected. This is because a copy core requires an L2 cache flush at the end of a LUT based copy operation, which is not required for MPBT flagged memcpy.

The largest bottleneck for reading/writing memory, and therefore also for inter-core message communication, as this involves reading/writing an MPB, is the fact that a P54C core can only have a single outstanding memory operation, and stalls until it completes. For read operations this is partially alleviated by the L1 cache as a few consecutive reads will hit in the same lines. For write operations the WCB can be used, however, as it is only enabled for MPBT flagged data this is not easy. A solution to this could be to add one or more programmable DMA engines capable of having multiple outstanding memory requests to the platform. They would be more simple than a P54C core, but could achieve a much higher memory throughput. Now we require the combination of multiple copy cores to achieve a higher throughput when copying a large amount of data, possibly wasting precious computing cycles. And to generally have a more optimal memory and communication performance in a runtime system, it needs to fully manage both virtual and physical memory, applying MPBT flags where necessary to speed up operations. This is what we are currently planning to do in the future for our SVP and S-Net runtimes on the SCC.

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson, “A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS,” pp. 108–109, February 2010.

[2] C. R. Jesshope, “A model for the design and programming of multi-cores,” *Advances in Parallel Computing*, vol. High Performance Computing and Grids in Action, no. 16, pp. 37–55, 2008.

[3] C. Grelck, S.-B. Scholz, and A. Shafarenko, “A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components,” *Parallel Processing Letters*, vol. 18, no. 2, pp. 221–237, 2008.

[4] M. W. van Tol and J. Koivisto, “Extending and implementing the self-adaptive virtual processor for distributed memory architectures,” *CoRR*, vol. abs/1104.3876, April 2011.

[5] C. Grelck, J. Julku, and F. Penczek, “Distributed S-Net: High-level message passing without the hassle,” in *1st ACM SIGPLAN Workshop on Advances in Message Passing (AMP’10), Toronto, Canada, 2010* (G. Bronevetsky, C. Ding, S.-B. Scholz, and M. Strout, eds.), ACM Press, New York City, New York, USA, 2010.

[6] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, “Light-weight communications on Intel’s Single-chip Cloud Computer processor,” *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.

[7] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, “Evaluation and improvements of programming models for the Intel SCC many-core processor,” in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011) – to appear, Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, (Istanbul, Turkey), July 2011.

[8] P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Y. Hoskote, S. Vangal, G. Ruhl, and N. Borkar, “A 2 Tb/s 6×4 mesh network for a Single-chip Cloud Computer with DVFS in 45 nm CMOS,” *Solid-State Circuits, IEEE Journal of*, vol. 46, pp. 757–766, April 2011.

[9] Intel, “SCC extended architecture specification,” November 2010. Revision 1.1.

[10] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, “The 48-core SCC processor: the programmer’s view,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[11] S. Peter, T. Roscoe, and A. Baumann, “Barrelfish on the Intel Single-chip Cloud Computer,” Tech. Rep. Barrelfish Technical Note 005, ETH Zurich, September 2010. http://www.barrelfish.org.

A Fast Inter-Kernel Communication and Synchronization Layer for MetalSVM

Pablo Reble, Stefan Lankes, Carsten Clauss, Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
{reble,lankes,clauss,bemmerl}@lfbs.rwth-aachen.de

***Abstract*—In this paper, we present the basic concepts for fast inter-kernel communication and synchronization layer motivated by the realization of a SCC-related shared virtual memory management system, called MetalSVM. This scalable memory management system is implemented in terms of a bare-metal hypervisor, located within a virtualization layer between the SCC's hardware and actual operating system. In this context, we explore the impact of the mesh interconnect to low level synchronization. We introduce new scaling synchronization routines based on SCC's hardware synchronization support targeting improvements of the usability of the shared memory programming model and present first performance results.**

***Keywords*—Inter-kernel Communication, Low-level Synchronization, Shared Virtual Memory**

I. Introduction

The Single-chip Cloud Computer (SCC) experimental processor [1] is a *concept vehicle* created by Intel Labs as a platform for many-core software research, which consists of 48 cores arranged in a 6×4 on-die mesh of tiles with two cores per tile.

Each core possesses 8 kByte of a fast on-die memory that is also accessible to all other cores in a shared-memory manner. These special memory regions are the so-called *Message-Passing Buffers* (MPBs) of the SCC. The SCC's architecture does not provide any cache coherency between the cores, but rather offers a low-latency infrastructure in terms of these MPBs for explicit message-passing between the cores. Thus, the processor resembles a *Cluster-on-Chip* architecture with distributed but shared memory.

The focus of this paper is a layer to realize fast inter-kernel communication and synchronization for *MetalSVM*. *MetalSVM* will be implemented in terms of a bare-metal hypervisor, located within a virtualization layer between the SCC's hardware and the actual operating system. This new hypervisor will undertake the crucial task of coherency management by utilizing special SCC-related features such as on-die Message-Passing Buffers (MPBs). In that way, common Linux kernels will be able to run almost transparently across the entire SCC system.

A requirement to inter-kernel synchronization is to control the access to shared resources such as physical devices or units of information. Limited availability of resources can be controlled by defining the access as a *Critical Section*. A critical section can thus be mapped to low-level synchronization methods such as a simple spin-lock.

Synchronization is a challenge in shared-memory programming. Especially for the SCC, whereas a high contention is possible, the currently available synchronization primitives (lock and barrier) could become a scalability bottleneck.

II. Motivation behind MetalSVM

A common way to use a distributed memory architecture is the utilization of the message passing programming model. However, many applications show a strong benefit when using the shared memory programming model. *Shared Virtual Memory* is an old concept to enable the shared memory programming model on such architectures. However, the success story of SVM systems could be greater. Many implementations are realized as additional libraries or as extensions to a programming language. In this case, only parts of the program data will be shared and a strict disjunction between the private and the shared memory is required. Intel's Cluster OpenMP is a typical example of such SVM systems, which in turn is based on TreadMarks [2], and first experiences are summarized in [3] . However, the disjunction between private and shared memory has side effects on traditional programming languages like *C/C++*. For instance, if a data structure is located in the shared virtual memory, the programmer has to guarantee that all pointers within this data structure refer also to the shared memory. Therefore, it is extremely complex to port an existing software stack to such an SVM system.

Furthermore, an SVM system virtualizes only the memory. Therefore, on distributed systems, the access to other resources like the file system requires additional arrangements. The integration of an SVM system into a distributed operating system, in order to offer the view of a unique SMP machine, increases the usability. Such systems are often specified as *Single System Image* (SSI) and *Kerrighed*[1] is an typical example for an SSI system. However, an extreme complexity and difficulties in maintainability are attributes of SSI's.

In the *MetalSVM* project, the SVM systems will be integrated into a hypervisor so that a common Linux is able

[1] http://www.kerrighed.org

to run as a virtual machine on the top of *MetalSVM*. Such a system is more simple to realize than a huge distributed operating system because it realizes only a unique view to the hardware and not to a whole operating system (including the file system and the process management). Several other projects have been started in this area. An example for a hypervisor-based SVM system is vNUMA [4] that has been implemented on the Intel Itanium processor architecture. For x86-based compute clusters, the so-called vSMP architecture developed by ScaleMP[2] allows for cluster-wide cache-coherent memory sharing.

The main difference between these approaches and *MetalSVM* is that they explicitly use traditional networks (Ethernet, InfiniBand) between the cluster nodes in order to realize their coherency protocols, whereas *MetalSVM* should support the SCC's distinguishing capabilities of transparent read/write access to the global off-die shared memory. However, a recent evaluation [5] with synthetic kernel benchmarks as well as with real-world applications has shown that ScaleMP's vSMP architecture can stand the test if its distinct NUMA characteristic is taken into account. We have already developed optimized applications for vSMP. In our scenario, vSMP produces only an overhead between 6% and 9%. In turn, 97% of this overhead is created by migrating the pages between the nodes.

vSMP benefits that InfiniBand supports DMA transactions, which don't stress the cores. The SCC doesn't support a similar feature. However, in MetalSVM the shared pages or the write notices – this depends on the memory model – are located in the shared off-die memory. Therefore every core has a transparent access to the shared data and doesn't take computation time of the other cores. MetalSVM has only to signalize the changes with small messages, which are sent via our inter-kernel communication layer. This layer will be explained in detail in Section VII.

Figure 1 shows exemplarily the simplest form of an SVM subsystem. In this case, only one core has access to a page

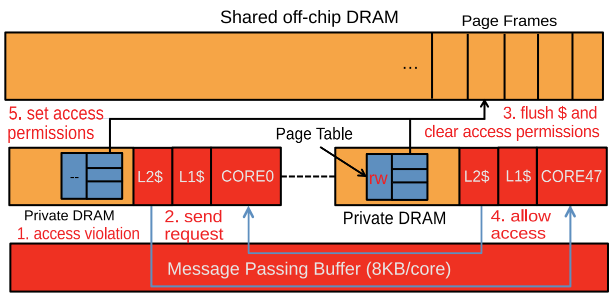


Figure 1. One Strategy to Realize a SVM Subsystem

frame.

If core 0 tries to access to a page frame, that holds core 47, an access violation will be triggered. Afterwards, the kernel of core 0 sends an access request to core 47. Core 47 receives this request, flushes its cashes, deletes the access permissions in its page table and sends the allowance for the page access back to core 0. Following this, core 0 sets the read/write permissions to its page table and continues its computation.

[2]http://www.scalemp.com

Nevertheless, the evaluation of vSMP has also shown that expensive synchronization is the big drawback of this architecture. We believe that especially this drawback will not occur in the context of our solution for the SCC, because *MetalSVM* provides better options to realize scalable synchronization primitives.

III. Basic Concept of MetalSVM

Figure 2 depicts *MetalSVM's* design approach, that allows in principle a common Linux version without SVM-related patches to run on the SCC in a common multicore manner. The light weight hypervisor approach is based upon the idea of a small virtualization layer, which is based on a self-developed kernel. The well established interface to run Linux as a para-virtualized guest will be used to realize the hypervisor. This interface is part of the standard Linux kernel thus no major modification to the guest kernel is needed. With the IO virtualization framework *virtio*, a common way to integrate IO device into the guest system exits.

Additionally, this framework will be used to integrate the eMAC Ethernet interface, which is part of the *sccKit 1.4.0* and already supported by our self-developed kernel. [6] While common processor virtualization aims for providing multiple virtual machines for separated OS instances, we want to use processor virtualization for providing *one* logical but parallel and cache coherent virtual machine for a single OS instance on the SCC. Hence, the main goal of this project is to develop a bare-metal hypervisor, that implements the required SVM system (and thus the memory coherency by applying appropriate consistency models) within this hardware virtualization layer in such a way that an operating system can run almost transparently across the entire SCC system.

The Realization of such a hypervisor needs a fast inter-core communication layer, which will be used to manage resources between the micro-kernel. Therefore, an important requirement for the communication layer is the support of asynchronous receiving and sending of request messages because it is not predictable when a kernel need an exclusive access to a resource, which is owned or managed by an other kernel. As a consequence, the synchronous communication library *RCCE* is not suitable for *MetalSVM*. [7] The inter-core communication layer is based on enhancements of our asynchronous *RCCE* extensions, which we called *iRCCE* [8], [9].

In the future version, the Linux kernel will forward its synchronization requests via the para-virtualization interface to the hypervisor, that has to use SCC-specific features to

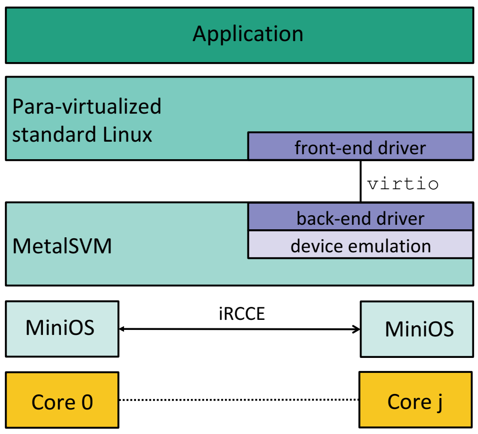


Figure 2. Basic Concept of MetalSVM

reach optimal performance. Therefore, we are in progress to develop an optimized synchronization layer, which will be discussed in the following section.

IV. Synchronization Layer

The SCC has 48 Test-and-Set (T&S) registers to enable a fast locking method. Therefore, each register an atomic bit. Whereas, reading access to target register returns a 0 if its status is unchanged 0, a return value of 1 indicates a previously atomic change from 1 to 0. Furthermore, the status can be changed over a write access.

This behaviour is a limitation compared to atomic functionality provided by shared-memory architectures. Hereby various operations such as *atomic increment* or *compare exchange* at word size using the `LOCK` prefix are provided. In contrast to that flexibility the syntax of T&S registers does not provide a read access without a possible changed value. Another restriction is the limited amount of 48 T&S registers. A synchronization layer for *MetalSVM* will thus require an allocation scheme for the T&S registers. The gory RCCE API provides an interface to the memory mapped T&S registers of the SCC. The function `RCCE_acquire_lock()` implements a simple busy wait loop (spin-lock) with the target register as parameter. The corresponding function `RCCE_release_lock()` releases the granted lock.

Access latencies are dependent of the location of an Unit in Execution (UE) due to the 2D interconnect, as depicted in Figure 3. For this measurement the UE was located at tile 00. A T&S register access is routed through the mesh and therefore affected as well as the MPB accesses. Depending on the distance of an UE to the target, a characteristic latency-rise regarding the 2D mesh and the tiled structure can be examined.

To evaluate the fairness and performance of different spin-lock implementations, it is useful to count the number of concurrent accesses. Whereas such a counter should be accessed with a low latency, due to the lack of atomic counters[3], the preferred location for the value is within the MPB. The counter access can be defined as a critical section and be protected by a spin-lock. Nevertheless, if the location of a UE is fixed we can calculate the expected latencies. Furthermore, a generic `Fetch and` Φ function targeting the MPB has to be filled with `nops` to emulate a symmetric and thus fair accessing scheme. Such a simulated *fair counter* can be used either for debug or for statistical purposes with the single impact to increase the duration of the critical section and thus generate an offset to the latency.

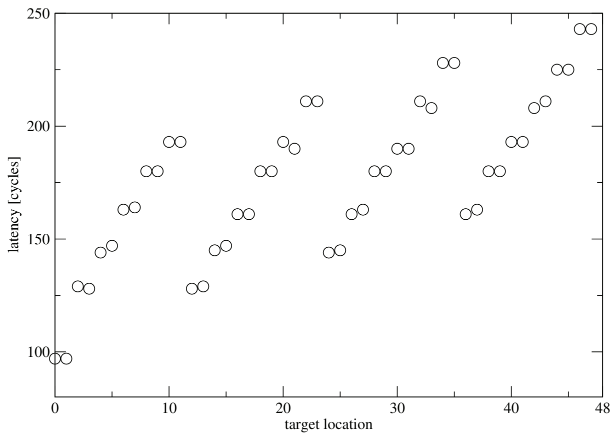


Figure 3. Latencies of uncontended T&S register accesses

V. Spin-Lock Variant: Exponential Backoff

However, by rising contention, a limited scalability results from the access to a single Test-and-Set register, as depicted in Figure 4. Each core contends for a spin-lock a million times for this benchmark. [10] The SCC cores runs with 533 MHz and the mesh interconnect with 800 MHz in our setup.

Expecting such a high contention, it could be useful to expand the simple spin-lock with the goal to minimize the time to acquire a lock. In the following, spin-lock alternatives will be explored with a focus on the implementation of a scaling synchronization layer for the SCC architecture regarding the hardware restrictions.

Programs even based on the gory RCCE API have no possibilty to realize a customized waiting strategy if a lock is occupied. Therefore, a method is needed that returns, just by indicating that a lock has been occupied. We call this method `trylock()`, that is needed to realize a spin-lock with a backoff.

Typically, a random component is introduced to minimize the chance that retries fall into the same point in time. We identified the stdlib `rand()` call to be time consuming with a high variance by using the standard SCC configuration.

[3]at least in sccKit v1.3.0

Therefore, we used a linear congruential generator to generate pseudo random numbers for the backoff. This way, a deterministic amount of time is ensured for the generation and this makes the results comparable to other spinlock methods because the generation is in the critical path and therefore degrades the performance.

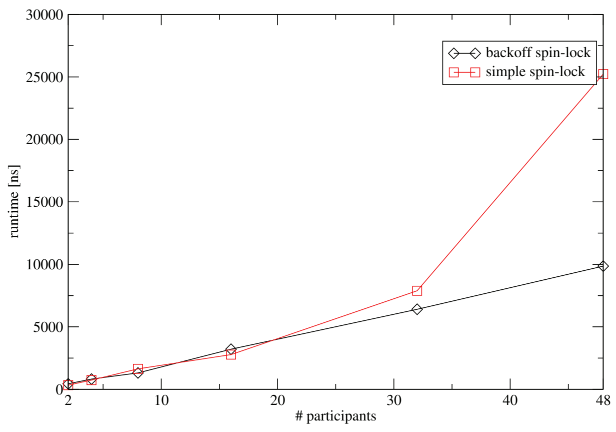


Figure 4. Average Times to acquire a Spin-lock

By using a backoff it is sufficient to use the presented `RCCE_release_lock` method. Thus, in the case of no concurrency a lock competitor needs two T&S register accesses.

VI. Test-and-Set Barrier

Using hardware support for synchronization such as T&S registers, it is evaluated in the following how a linear barrier can be realized. Assuming a communicator size of n, Listing 1 shows an implementation of a barrier using only T&S Register for signaling incoming UE's and wait for release.

```
// linear search for a free lock
  while( !Test_and_Set(step) ) ++step;
  if(step != n-1) {
// wait for release, signal exit
// and wakeup right neighbor
    while(!Test_and_Set(step));
    *(virtual_lockaddress[step]) = 0x0;
    *(virtual_lockaddress[step+1]) = 0x0;
  } else {
// last UE: wakeup first UE and
//          wait for release
    Test_and_Set(step);
    *(virtual_lockaddress[0]) = 0x0;
    while(!Test_and_Set(step));
    *(virtual_lockaddress[step]) = 0x0;
  }
```

Listing 1. Test-and-Set Barrier algorithm in C

A requirement for the algorithm are n available T&S registers with an initial value of 0. However the implementation works without the need for MPB initialization, by not touching it.

Each UE that has entered the barrier tries to grab the first available T&S register, for instance ordered by the numerical count of the located core. After this first step each UE except the last one will spin on the T&S register granted before. Hereby, the contention is minimal and therefore the simple spinlock is sufficient.

Figure 5 shows the runtimes of the presented Barrier to the standard `RCCE_barrier`. A dynamic mapping of linear access ordering and Core-ID by indicating the location of a T&S register is a benefit of the linear cycles. However, a static mapping could provide a better performance by using a tree-based communication pattern [11].

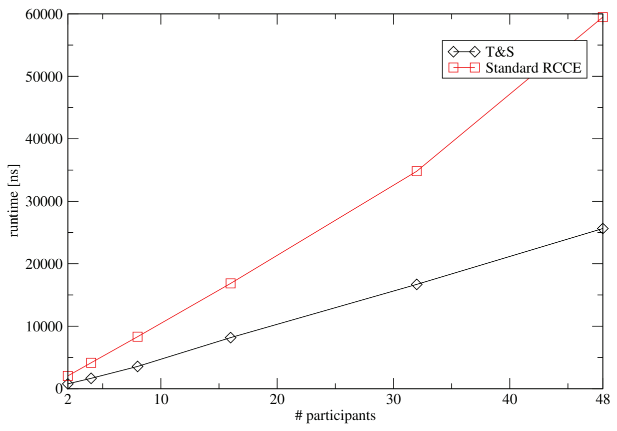


Figure 5. Barrier Runtimes of RCCE Compared to our Approach

VII. Integration of iRCCE into MetalSVM

Due to the lack of non-blocking communication functions within the current RCCE library, we have started to extend RCCE by such asynchronous communication capabilities. In doing so, we aim at avoiding to interfere with the original RCCE functions and therefore we have placed our extensions into an additional library with a separated namespace called iRCCE [9].

An obvious way to realize non-blocking communication functions would be to use an additional thread that processes the communication in background while the main thread returns immediately after the function invocation. Although this approach seems to be quite convenient, it is not applicable in *bare-metal* environments where a program runs without any operating system and thread support. Since we have planed from the start to integrate iRCCE into *MetalSVM*, we had to waive this thread-based approach. Therefore, we have followed another approach where the application (or moreover the *MetalSVM* kernel) must drive on the communication progress by itself. For this purpose, a non-blocking communication function returns a *request handle*, which can be used to trigger its progress by means of additional `push`, `test` or `wait` functions [8].

In the context of *MetalSVM*, these progress functions are called by entering the kernel via interrupt, exception or system call. Additionally, *MetalSVM* checks at this point if the kernel has recently received a message. Therefore, all messages begin with a header, which specifies the message type and the payload size. The definition of a message type is important because this layer will be used to send messages between the cores from the SVM system and also from the TCP/IP stack. Our self-developed kernel, that builds the base of *MetalSVM*, supports LwIP [4] as a lightweight TCP/IP stack. The TCP/IP stack will be used to collect status information from the kernels and to feature the support of administration tools in the future.

The maximum delay between sending and receiving a message header is as large as a time slice, because at least after one slice a (timer) interrupt will trigger, which checks for incoming messages. Additionally, *MetalSVM* allows the sender to trigger a remote interrupt on the side of the receiver in order to reduce the delay. However, this creates an additional overhead because the calculation on the remote core could be unnecessarily interrupted. Further analysis will be done to decide which communication model (with or without triggering an interrupt) will be the best choice for *MetalSVM*.

VIII. Conclusion and Outlook

In this paper, we have presented our first steps to design and implement a low-latency communication and synchronization layer for *MetalSVM*, a shared virtual memory system for the SCC. We have pointed out the demand especially for fast and hardware-based low-level synchronization primitives and we have shown how such primitives can be realized on the SCC be means of its special hardware features. In this context, we have presented an optimized barrier algorithm that utilizes the SCC's Test-and-Set registers and we have shown that this algorithm outperforms the RCCE-related approach of using the SCC's on-die message-passing buffers for this purpose. We have also shown that especially in case of highly contended locks an exponential back-off algorithm can lead to an improved scalability compared to the straight forward approach of the common RCCE library. Moreover, we believe that we are even able to further improve our low-level locking mechanism by applying a *tree-like* access pattern to the Test-and-Set registers. Although such a tree-based locking algorithm would cause an addition overhead in the order of $O(log(n))$ in the non-contented case, we think that this approach will play out its strength especially in highly contended situations, as they will usually occur in the context of *MetalSVM*.

Acknowledgment

The research and development was funded by Intel Corporation. The authors would like to thank especially Ulrich Hoffmann, Michael Konow and Michael Riepen of Intel Braunschweig for their help and guidance.

[4] http://savannah.nongnu.org/projects/lwip/

References

[1] *SCC External Architecture Specification (EAS)*, Intel Corporation, November 2010, Revision 1.1.

[2] P. K. A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the USENIX Winter 1994 Technical Conference*. Berkeley, CA, USA: USENIX Association, 1994, pp. 10–10.

[3] C. Terboven, D. an Mey, D. Schmidl, and M. Wagner, "First experiences with intel cluster openmp," in *OpenMP in a New Era of Parallelism*, ser. Lecture Notes in Computer Science, R. Eigenmann and B. de Supinski, Eds. Springer Berlin / Heidelberg, 2008, vol. 5004, pp. 48–59.

[4] M. Chapman and G. Heiser, "vNUMA: A virtual shared-memory multiprocessor," in *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, Jun 2009, pp. 349–362.

[5] D. Schmidl, C. Terboven, D. an Mey, and M. Bücker, "Binding Nested OpenMP Programs on Hierarchical Memory Architectures," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, 6th International Workshop on OpenMP (IWOMP 2010)*, ser. Lecture Notes in Computer Science, Tsukuba, Japan, June 2010.

[6] *The SccKit 1.4.0 Users Guide*, Intel Labs, March 2011, Revision 0.92.

[7] T. Mattson and R. van der Wijngaart, *RCCE: a Small Library for Many-Core Communication*, Intel Corporation, May 2010, Software 1.0-release.

[8] C. Clauss, S. Lankes, J. Galowicz, and T. Bemmerl, *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer – User Manual*, Chair for Operating Systems, RWTH Aachen University, December 2010, Users' Guide and API Manual.

[9] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor (accepted for publication)," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011) – to appear, Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, Istanbul, Turkey, July 2011, accepted for publication.

[10] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6 –16, January 1990.

[11] N. S. Arenstorf and H. F. Jordan, "Comparing barrier algorithms* 1," *Parallel Computing*, vol. 12, no. 2, pp. 157–170, 1989.

Scalable Runtime Support for Data-Intensive Applications on the Single-Chip Cloud Computer

Anastasios Papagiannis[*] and Dimitrios S. Nikolopoulos[*]
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
GR–70013, Heraklion, Crete, GREECE
{apapag,dsn}@ics.forth.gr

***Abstract*—Many-core processors, due to their complexity and diversity, necessitate high-productivity, domain-specific approaches to parallel programming. These approaches should hide architectural details and low-level parallelization constructs, while enabling scalability and performance portability. This paper presents a scalable implementation of MapReduce, a runtime system used widely by domain-specific languages for large-scale data processing, on the Intel SCC. We address the scalability bottlenecks of MapReduce with data partitioning, combining and sorting algorithms that we customize for the SCC network on-chip architecture. We achieve linear or superlinear speedups for representative MapReduce workloads with data sets that fit on a single SCC node. We also show that the SCC node outperforms the IBM Cell QS22 Blade, when the latter uses the fastest implementation of MapReduce available for the Cell processor.**

***Index Terms*—MapReduce; Single-Chip-Cloud; Resource management; Runtime systems; Operating Systems; Parallel Programming Models.**

I. Introduction

Programming models for future many-core processors should disengage from low-level parallel programming constructs –such as threads, locks, and messages– and embrace high-productivity domain-specific alternatives. Domain-specific frameworks for parallel programming will require scalable runtime systems to exploit many-core architectures. As more many-core processor architectures forgo cache coherence and use fast on-chip communication to improve performance and energy-efficiency, runtime systems for parallel programming face the challenge of scaling, while hiding the complexities of explicit communication from programmers [1].

Google's MapReduce programming model [2] is widely used for implementing domain-specific languages to support large-scale data processing applications. MapReduce borrows idioms from functional programming to express parallel operators on distributed collections of data and to aggregate data. The MapReduce programming framework has been implemented on a variety of parallel architectures, including clusters, shared-memory multi-core systems with coherent caches, graphics processing units, and multi-core processors with software-managed local memories [2], [3], [4], [5], [6].

This paper presents the first, to the best of our knowledge, implementation of the MapReduce programming model and runtime system on the Intel Single-Chip Cloud Computer (SCC). We present a design that utilizes effectively the SCC interconnection network and on-chip shared communication buffers to alleviate two fundamental scalability bottlenecks of MapReduce: data partitioning and data sorting. The artifact of our contribution is a fast and scalable implementation of MapReduce, based on customized on-chip data exchange, combining, and sorting algorithms. We achieve linear or superlinear speedups for representative MapReduce workloads, which process data sets that fit in the memory of a single SCC node. We further show that an SCC node outperforms an IBM QS22 Cell blade with two Cell processors, when the latter uses the fastest implementation of MapReduce for the Cell processor available to date [6].

II. Background

We provide background for our work by presenting the MapReduce program execution stages and discussing the key architectural properties of the Intel SCC processor.

A. MapReduce

MapReduce [2] is a high-level parallel programming model based on two primitives, *map* and *reduce*. Parallel computation in MapReduce is expressed as processing and aggregation operators applied on distributed data sets. A MapReduce application processes an input of (key, value) pairs to produce an output of (key, value) pairs. A typical MapReduce program executes in four stages, a *map* stage, where parallel workers (called *mappers*) produce a set of intermediate (key, value) pairs for each input pair, a *partition* stage which exchanges intermediate data between mappers, a *group* stage which groups all intermediate (key, value) pairs associated with the same key, and a *reduce* stage which merges the values associated with each key. The map and reduce stages execute user-defined data processing and aggregation operators. MapReduce implementations typically have an explicit barrier between the map and partition stages, although this barrier can be replaced by a software pipeline [7].

Figure 1 shows a typical MapReduce execution flow. The MapReduce runtime splits the input data into fixed size chunks and assigns each chunk to a *mapper*. Each mapper executes

[*]Also with the Department of Computer Science, University of Crete, Heraklion, Greece.

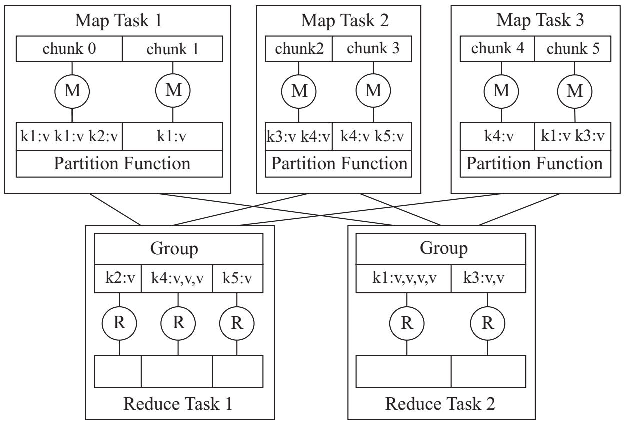


Fig. 1: A typical MapReduce execution flow (*M* stands for *Mappers*, *R* stands for *Reducers*)

the map function on its assigned chunk, which consists of a series of input (key, value) pairs. The map function generates zero or more intermediate (key, value) pairs for every input (key, value) pair. The input and output types of the map function may differ. The runtime system maintains a number of *reducers* that perform data aggregation. During the map stage, each mapper exports as many different partitions as the number of reducers. It is permissible for mappers and reducers to execute on the same compute node.

To split the output of a mapper, the user may define a *partition* function. The partition function takes as input a key and the number of reducers and returns an index to the reducer to which the output should be sent. The typical implementation of this step is to hash the key and compute the partition index as the key's hash value modulo the number of reducers. It is important to pick a partitioning function that gives an approximately uniform distribution of data per reducer for load balancing purposes, otherwise the runtime may stall waiting for slow reducers to finish. The partition stage executes between the map and reduce stages and moves each intermediate (key, value) pair from the node running the mapper that produced it to the node on which it will be reduced.

Following the partitioning stage, an optional *group* stage sorts the intermediate data on each reducer. The runtime has to rearrange the intermediate (key, value) pairs so that the data is organized as a set of unique keys and a list of values for each key. Finally, the runtime executes the user-defined reduce function to aggregate intermediate (key, value list) pairs. The reducer iterates through the values that are associated with each key and produces zero or more output (key, value) pairs.

B. Intel Single-Chip-Cloud-Computer (SCC)

The Intel SCC [8] (Figure 2) is a many-core processor with 24 tiles and 2 IA cores per tile. The tiles are organized in a 4×6 mesh network with 256 GB/s bisection bandwidth. The processor has 4 integrated DDR3 memory controllers, one for each group of 6 tiles. Each core has a private L1 instruction cache of 16 KB, a private L1 data cache of 16

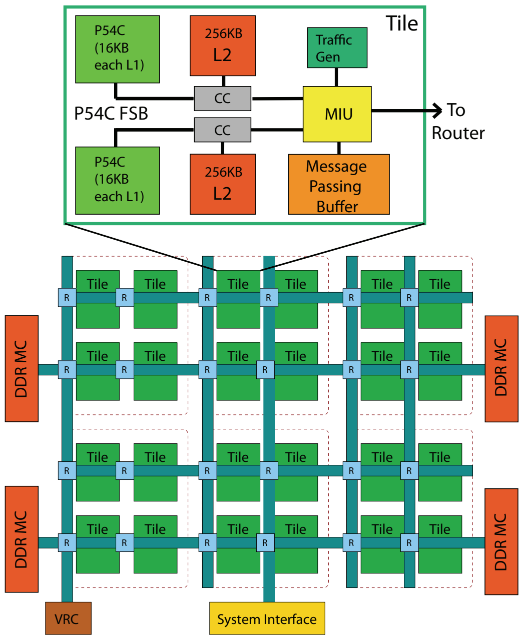


Fig. 2: SCC processor diagram

KB and a private unified L2 cache of 256 KB. Each dual-core tile has a 16 KB message passing buffer (MPB), which is the only component of the SCC on-chip memory hierarchy that is shared between cores. The MPB provides space for direct core-to-core communication. On-chip communication data is read from the MPB through the L1 data cache and bypasses the L2 cache. For writes, a no-allocate policy is used, in conjunction with a write combining buffer at the L1 cache. Software needs to maintain coherence between the MPB and the L1 caches by using a, unique to the SCC, L1 cache invalidation instruction, when data is stored in the MPB.

The 32-bit address space of the system is mapped to an extended 34-bit address space to allow access to up to 64 GB of off-chip memory (up to 16 GB from each group of 6 tiles). This is accomplished through a Look-Up Table (LUT) attached to each core. The address space of the system is configurable and can be distributed between private off-chip memory associated with each core, shared off-chip memory, and shared on-chip SRAM, which corresponds to data stored in the message buffers and cached in the L1 cache.

We implemented MapReduce using the the standard software environment of SCC compute nodes available by Intel, namely a configuration running a Linux kernel on each core and RCCE, the Intel one-sided communication library [9].

III. MAPREDUCE DESIGN

We implement a seven-stage runtime system for MapReduce. The seven stages are *map*, *combine*, *partition*, *group*, *reduce*, *sort* and *merge*. The *combine* and *merge* stages are optional in typical MapReduce setups, whereas the *group* stage replaces the intermediate *sort* stage of the original MapReduce

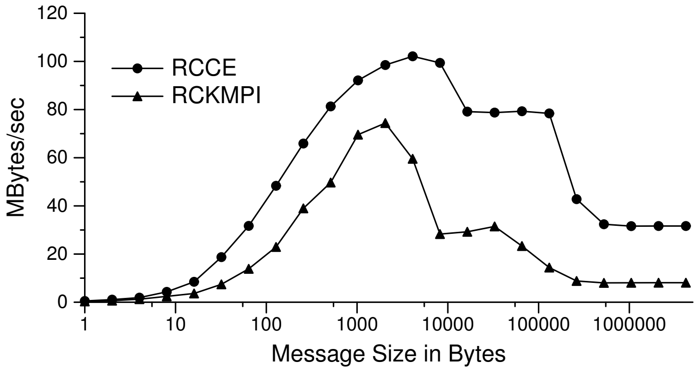


Fig. 3: RCKMPI vs. RCCE bandwidth in a ping-pong benchmark

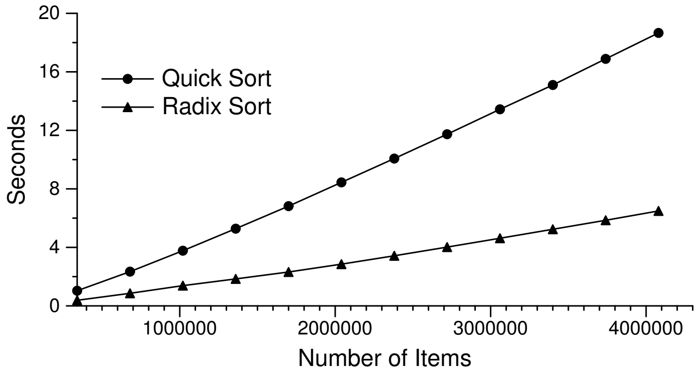


Fig. 4: Libc qsort vs. radix sort, for a variable number of word-size elements

pipeline, to reduce overall computational complexity. We describe the stages of MapReduce in more detail in the following paragraphs.

a) Map: During the map stage, the runtime system divides the input evenly to as many parts as the number of mappers. In our implementation, we use as many mappers as the number of cores. Each core then executes the user-defined map function over its assigned input data. We preallocate a large chunk of memory (64 MB) for the output of the map stage. If the volume of intermediate data produced is more than 64 MB, we allocate a new output buffer on demand. Each core exports as many intermediate data partitions as the number of cores in the system. To split intermediate data between partitions, we use either a user-defined hash function or a default generic hash function available by our MapReduce runtime system. Each core emits keys and values in a contiguous buffer.

b) Combine: This stage is optional and executes if the user provides a combiner function. The purpose of this stage is to reduce locally the size of each partition produced during the map stage. The combine function takes as input a key and a list of partially aggregated intermediate values associated with that key. It produces as output a single (key, value) pair where the value is an updated partial aggregation associated with the same key. We use the combine function to reduce data volume and balance the partitions that will be processed by different cores in the following stages of MapReduce. We use the same strategy as in the group stage described later to group together all values with the same key. The combiner function produces a new intermediate (key, value) pair for each intermediate key and its corresponding list of values. The combine stage is equivalent to applying the reduce stage in each of the intermediate partitions.

c) Partition: The partition stage requires an all-to-all exchange between cores. Data partitions generated during the map stage may be different in size. We implement a custom all-to-all exchange algorithm for the SCC to achieve scalable data partitioning. The algorithm first executes an all-to-all exchange of the intermediate partition's sizes, followed by an all-to-all exchange of the intermediate data. We implement the all-to-all exchange using pairwise exchanges. Let p be the number of available cores and *rank* the core ID. This algorithm uses $p-1$ steps and in each step k, core *rank* receives data from core $rank-k$ and sends data to core $rank+k$. We opted to use the *RCCE_*$\{$*send, recv*$\}$ functions to implement this all-to-all exchange. RCCE is an SCC communication runtime environment based on one-sided get-put communication primitives [9]. Figure 3 shows that native RCCE achieves better throughput than RCKMPI, when communication flows through the SCCMPB channel, which uses exclusively the on-chip message-passing buffers. RCCE also uses the MPB buffers to exchange data.

d) Group: The group stage groups together all (key, value) pairs with the same key, taken across all intermediate data partitions. In previous works [6], [3], [4], [5], [10], [11], a generic sorting scheme with a user-defined comparator was used to perform grouping. We replace this scheme with a radix sort algorithm [12] for grouping on the SCC. The sorting algorithms employed in prior MapReduce implementations on multi-core systems have complexity $O(nlogn)$, whereas radix sort has complexity $O(kn)$ where k is the size of the key in bytes. Figure 4 shows a comparison of the libc quicksort implementation and our radix sort implementation for different input sizes. Radix sort outperforms quicksort with the caveat that radix sort sorts strings of bytes and can not use a user-defined comparator for sorting. This caveat implies that in applications where the key data type is not a string, radix sort may produce unsorted sequences that need to be processed further in the following stages of MapReduce.

Previous sorting algorithms used in MapReduce swap (key, value) pairs by copying the actual data of these pairs. Our radix sort algorithm swaps pointers to (key, value) pairs instead. Thus, in every swap we only exchange two pointers, making the cost of the swap independent of the size of the (key, value) pair. The output of this stage is an array of pointers to the actual data. This array needs to be transformed to a structure containing pairs of keys and value lists. We accomplish this by simply iterating through the array and finding the unique keys. We initiate an iterator for accessing the values with no need to rearrange the data in memory. We statically know the sizes of all the buffers needed for the sorting stage, therefore we preallocate these buffers. This optimization minimizes the

Application	Class	Input size
Word Count	partition-dominated	60 MB
Histogram	sort-dominated	400 MB
Linear Regression	map-dominated	32 MB
Kmeans	map-dominated	115 MB

TABLE I: MapReduce application workloads

overhead of dynamic memory allocation.

e) Reduce: The reduce stage executes a user-defined key aggregation function. The prior group stage exports an array of all distinct keys where each key contains the number of occurrences of the key and a pointer to an array of its values. The output size of the reduce stage can be statically identified, therefore we preallocate the stage's output buffers, once again to minimize dynamic memory allocation overhead.

f) Sort: The sort stage sorts the (key, value) pairs produced following the reduction, using quicksort and a user-specified comparator. This stage is necessary because the earlier group stage may produce unsorted sequences. However, this sort stage is necessary only if the following data merging stage is needed as well.

g) Merge: The merge stage optionally merges the output of all cores in one core. In the default configuration of SCC, each core has its private memory, therefore in applications that require merging, we need to produce the final output in the memory of a single core. We use the binomial merge algorithm for this stage [13], which completes in *logn* steps.

IV. EXPERIMENTAL ANALYSIS

Table I lists the MapReduce application workloads that we used for experiments. Following conventions from [11], we classify applications as map-dominated, partition-dominated and sort-dominated, according to the phase where these applications fail to scale on a multi-core system.

Histogram counts the frequency of occurrences of each RGB color component in an image file. The map function emits the occurrences of each color component in pixels and the reduce function produces the sum of occurrences of each component. *Word Count* counts the number of occurrences of each word in a text file. The map function splits the input text into words, whereas the reduce function sums the number of occurrences of each word to produce a final count. *Kmeans* creates clusters from a set of data points, by finding the closest cluster for each data point in the map function and computing the cluster means in the reduce function. *Linear Regression* computes a line of best fit for a set of points, given their 2D coordinates. Map computes intermediate summary statistics for the points, while reduce gathers all data of each of the summary statistics and calculates the best fit.

In our experiments, we use the standard frequency configuration of the SCC chip. In this configuration, each tile runs at a frequency of 533MHz, the mesh interconnect runs at a frequency of 800MHz and DRAM runs at a frequency of 800MHz.

Figure 5 illustrates speedup and Figure 6 illustrates execution time of application workloads, with and without a

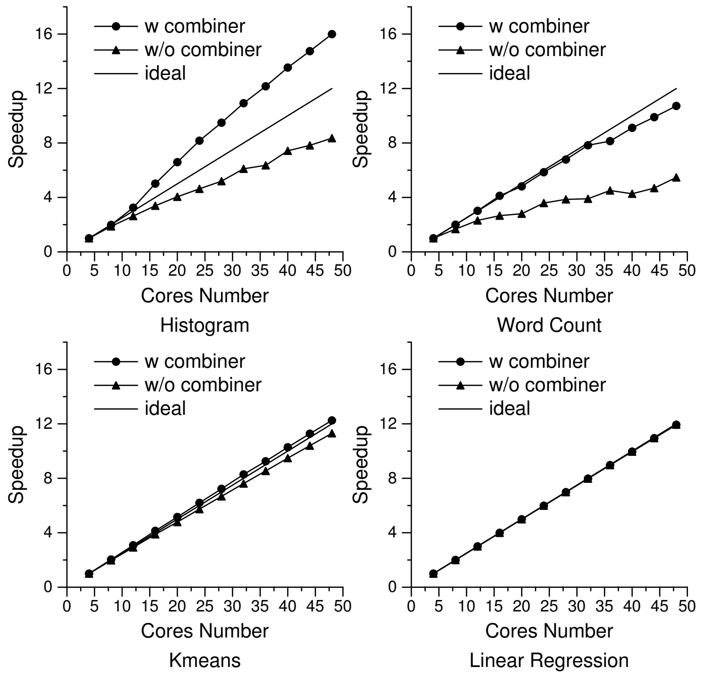


Fig. 5: Speedup of MapReduce workloads

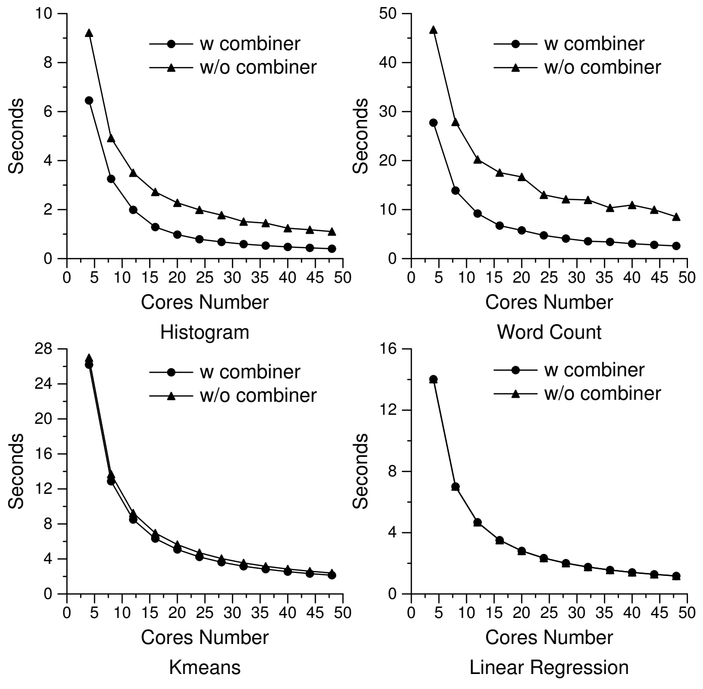


Fig. 6: Execution time of various applications

combiner function. Speedup is calculated using execution time on 4 cores (2 tiles) as the nominator, therefore ideal linear speedup is 16 for the entire SCC chip. Figure 7 and Figure 8 show breakdowns of execution time for all applications. All applications scale well on the chip. With the use of a combiner function, applications have nearly ideal linear or in some cases, superlinear speedup. The partition stage exhibits the worst scaling behavior. The combine stage improves performance by reducing the intermediate data exported from the map stage.

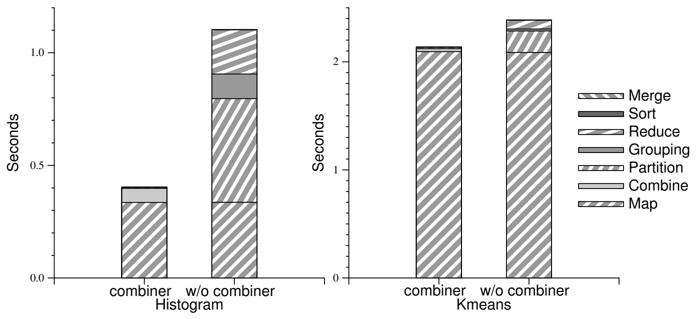


Fig. 7: Execution time breakdowns

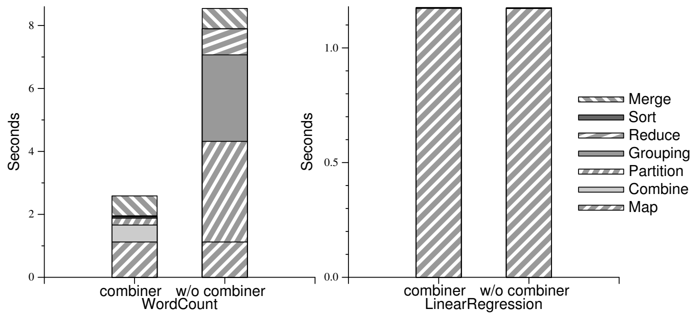


Fig. 8: Execution time breakdowns

This in turn means that the partition stage has to exchange less data between cores. The reason for superlinear speedup is that the complexity of the group stage decreases exponentially with the number of cores. In applications where the grouping stage dominates execution time, the overall application speedup may therefore be superlinear. We analyze briefly individual applications in the following paragraphs.

Histogram does not achieve perfect speedup without a combiner (Figure 5), because the partition stage does not scale. Partitioning overhead dominates execution time (Figure 7). Reducing the intermediate data size with a combiner alleviates the bottleneck. Using the combiner also decreases the execution time of the grouping and reduce stages. The combiner function is the same function as the one used in the reduce stage in this benchmark. Therefore, the combine stage executes a part of the reduce stage on the intermediate values available locally to each core. *Histogram* exports a maximum of only 3×255 different keys, which makes the merge stage time insignificant.

KMeans and *Histogram* have similar behavior (Figure 5), with the exception that in *KMeans* the map stage dominates execution time, therefore the combiner has a less significant impact on overall execution time (Figure 7). This is also the reason why in *Kmeans* we do not achieve superlinear speedup.

Linear Regression is an entirely map-dominated benchmark and therefore scales perfectly. Each map function exports five (key, value) pairs and therefore group and reduce times are insignificant. Since the map stage exports only five different keys, we can only use five cores in the execution stages after map. From the breakdowns (Figure 8) we observe that this is

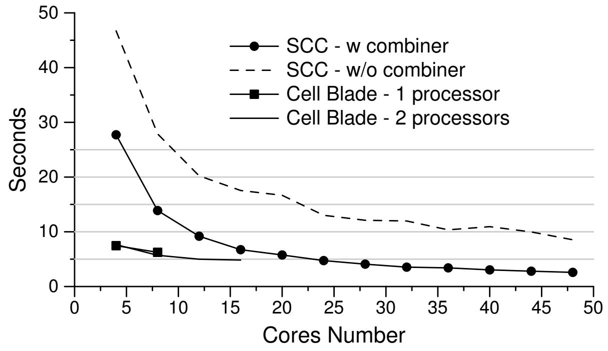


Fig. 9: Comparison of SCC and Cell BE processors using wordCount benchmark

not a scalability bottleneck. Merging the output results of five cores has negligible overhead.

Word Count incurs load imbalance in the grouping stage. This leads to erratic speedup (Figure 5). However, the problem is easily alleviated with a combiner function that rebalances the volume of intermediate data between cores. We have compared many hash functions for strings while experimenting with *Word Count*. We ended up using the *djb2* hash function. This function initially sets $hash = 5381$ and then for each character of the string it sets $hash = hash * 33 + c$ where c is the ASCII value of each character. The selected hash function results in better distribution of intermediate keys among different partitions in comparison with other hash functions.

Figure 9 illustrates a comparison of our implementation of MapReduce on the SCC with a competitive implementation of MapReduce on the Cell processor, which is the fastest implementation for that processor published to date [6]. We used the Word Count benchmark with a 60 MB input size. We ran this benchmark on a Cell QS22 Blade with 8GB RAM and report execution time with the Cell MapReduce runtime published in [6], using one or both of the Cell processors of the QS22 blade. Each Cell processor has 9 cores, out of which 8 (the SPE vector cores) are used for MapReduce tasks and one (the PowerPC PPE core) is used for the runtime system. The maximum number of mapper and reducer cores is 8 when using one Cell processor and 16 when using two Cell processors. The SCC node with a single SCC processor outperforms the dual-processor Cell QS22 blade by up to $1.87\times$, when the SCC MapReduce uses combiner functions. We note that the Cell processors on the QS22 run at 3.2 GHz and that each core on the Cell has a software-managed local store of the same size as the L2 cache of each core on the SCC.

V. Related Work

Several prior research efforts ported MapReduce to prominent hardware platforms for high-performance computing, including multicore processors [4], [5], [14], GPUs [3], [15] the Cell processor [11], [6], [16] and FPGAs via direct software to hardware translation [17].

Phoenix, a port of MapReduce for cache-coherent shared-memory multicore systems [4], [5], exploits locality implicitly by controlling the granularity of tasks and the assignment of tasks to cores. Phoenix performs dynamic assignment of map and reduce tasks to cores. It controls task sizes so that the working set of each task fits in the L1 cache of each core. Phoenix also provides an option to perform prefetching in the L2 data cache. The main focus in the design of Phoenix is on achieving scalability through NUMA-aware memory management. Each map thread emits intermediate results on a space allocated on the closest memory module to the CPU the thread is scheduled on. In the most recently published version of Phoenix [5], the authors use a multi-layer approach to optimize the runtime system. These layers include the algorithm, the implementation and and the runtime-OS interaction. A different approach to optimize Phoenix is proposed in [14] where the authors use tiling to minimize task memory footprints and improve cache locality.

MapReduce has also been ported to the Cell BE processor [11], [6]. In the implementation presented in [6], which is the fastest, the runtime system controls locality explicitly, using DMAs and software prefetching via multi-buffering in the map and merge-sort stages. Contrary to Phoenix, the runtime system does not hash and does not partition keys in per-core buffers, thereby eliminating memory copies, while still allowing a balanced distribution of work during the sort and reduce stages.

Implementations of MapReduce on GPUs also consider the implications of explicitly-managed local memories [10], [3], [15]. Mars [3] uses mock map tasks to compute the sizes of buffers needed by each core for emitting results of real map tasks. Other optimizations of MapReduce on GPUs focus on achieving fine-grain interleaving of memory accesses from threads on the GPU, to utilize the available GPU memory bandwidth.

VI. Conclusions

This paper presented a scalable implementation of Google's MapReduce runtime system on the Intel SCC. The implementation attests to the scalability of the chip, as well as its ability to support software stacks and high-level parallel programming models that hide explicit communication from programmers. Our implementation of MapReduce leveraged one-sided on-chip communication primitives and customized data combining algorithms to alleviate bottlenecks that arise during data partitioning and sorting. We demonstrated perfect linear or superlinear scaling of applications with realistic datasets for a single SCC node and performance that exceeds the fastest to date implementation of MapReduce on IBM Cell blades. While our results are promising, our work raises several interesting questions for future research. These include design choices for implementing the full MapReduce execution path, including I/O, alternative management schemes for the SCC memory hierarchy that exploit off-chip shared memory, the implementation of dynamic task scheduling, and further analysis of applications.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the I-CORES project, grant agreement n^o 224759

References

[1] S. Schneider, J.-S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos, "A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2009, pp. 131–140.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[3] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce Framework on Graphics Processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 260–269.

[4] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2007, pp. 13–24.

[5] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009, pp. 198–207.

[6] A. Papagiannis and D. S. Nikolopoulos, "Rearchitecting Mapreduce for Heterogeneous Multicore Processors with Explicitly Managed Memories," in *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, Sep. 2010, pp. 121–130.

[7] A. Verma, N. Zea, B. Cho, I. Gupta, and R. Campbell, "Breaking the MapReduce Stage Barrier," in *Proceedings of the 2010 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2010, pp. 235–244.

[8] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS," in *Proceedings of the 2010 IEEE International Conference on Solid-State Circuits (ISSCC)*, Feb. 2010, pp. 108–109.

[9] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard *et al.*, "The 48-core SCC Processor: the Programmer's View," in *Proceedings of Supercomputing'10: The 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp. 1–11.

[10] B. Catanzaro, N. Sundaram, and K. Keutzer, "A Map Reduce Framework for Programming Graphics Processors," in *Proceedings of the Third Workshop on Software and Tools for Multicore Systems (STMCS)*, Apr. 2008.

[11] M. de Krujif and K. Sankaralingam, "Mapreduce for the Cell B.E. Architecture," *IBM Journal of Research and Development*, vol. 53, no. 5, Sep. 2009.

[12] P. M. McIlroy, K. Bostic, and M. D. Mcilroy, "Engineering Radix Sort," *Computing Systems*, vol. 6, pp. 5–27, 1993.

[13] R. Thakur and R. Rabenseifner, "Optimization of Collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, Feb. 2005.

[14] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 523–534.

[15] W. Ma and G. Agrawal, "A Translation System for Enabling Data Mining Applications on GPUs," in *Proceedings of the 23rd ACM International Conference on Supercomputing (ICS)*, Jun. 2009, pp. 400–409.

[16] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos, "Supporting MapReduce on Large-Scale Asymmetric Multi-core Clusters," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 25–34, Apr. 2009.

[17] J. H. Yeung *et al.*, "Map-Reduce as a Programming Model for Custom Computing Machines," in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2008, pp. 149–159.

Linux Operating System Support for the SCC Platform - An Analysis

Jan-Arne Sobania, Peter Tröger, Andreas Polze
Hasso Plattner Institute, University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
Email: [jan-arne.sobania/peter.troeger/andreas.polze]@hpi.uni-potsdam.de

Abstract—**The Single-Chip Cloud Computer (SCC) is an experimental many-core system created for research purposes by Intel Labs. In this paper, we analyze the necessary adjustments to run a Linux kernel on a processor core of the new processor design. Starting from the Intel-provided set of Linux modifications, we present an alternative strategy for creating a SCC-compliant kernel. Our approach allows to use recent versions with the prototype platform, provides better portability for future versions of Linux, and enables the utilization of latest kernel functionality such as virtualization for future research.**

I. Introduction

The Single-Chip Cloud Computer (SCC) experimental processor [1] is a 48-core concept vehicle created by Intel Labs. It is intended to act as a hardware platform for many-core software research on different system levels. Highlights of the SCC architecture are the on-die mesh network for communication between cores and memory controllers, flexible power management and frequency scaling capabilities, and a reconfigurable shared memory hardware. The prototype board has a *Field-Programmable Gate Array (FPGA)* that connects the internal mesh network to a separate *Management Console PC (MCPC)*. Both together emulate the chipset and devices for the prototype board.

The new processor core design (named "GaussLake") is derived from the Intel Pentium 90 architecture (P54C) [2]. It combines two cores and their L2 caches to a *tile*. Each of the 24 tiles has a *message passing buffer (MPB)* and a *message router* to let the cores communicate on the mesh network. Access to memory is realized through message exchange with the memory controllers. The system does not maintain cache coherency for shared memory regions accessed from multiple cores, since all memory controllers operate decoupled from each other.

Due to the special hardware architecture, standard Windows or Linux kernels cannot run even on a single core. The default operating mode of the current prototype is to run 48 independent instances of a modified Linux distribution (*SCC Linux*). It contains a tailored Linux 2.6.16 kernel, originally released in 2005, with full sources available from Intel.

In this paper, we first analyze the modifications applied to the default Linux kernel sources in Section II. Based on this analysis, we propose an alternative kernel modification strategy in Section III, which enables the utilization of recent and future Linux kernel versions for the SCC hardware.

II. SCC Linux

The neccessary handling of the prototype hardware by a operating system can be summarized in three major categories:

- *Bootloader and Real-Mode Setup:* The SCC prototype system does not contain any firmware implementation, so the early boot stages must be modified to bring up the operating system kernel.
- *Protected Mode Setup:* Initialization of most system hardware is accomplished only after the kernel enters protected mode. This phase again demands consideration of SCC specialties.
- *SCC Hardware Support:* Specialized parts such as the mesh network must be made available to user-mode applications. In addition, networking functionalities must be realized by the help of the FPGA-MCPC bridge.

The following sections discuss these three classes.

A. Bootloader and Real-Mode Setup

Due to the lack of firmware support for raw access to devices, a standard boot loader approach is not applicable in an SCC operating system. Intel's boot procedure therefore prepares an initial memory image on the MCPC, and copies it via mesh network into the core's private memory region while the core is in reset state. Once the reset line is de-asserted, all cores begin to execute the platform-defined reset vector at `0xFFFFFFF0`.

The reset vector just needs to setup real-mode registers, e.g. to provide a stack to the following boot stage, and then jump to the Linux kernel's real-mode setup entry point. The entry-point relies on a parameter block describing basic properties of the system, such as the type of boot loader, address of the loaded (protected-mode) kernel, initial ram disk and kernel command line. On a regular x86 system, this parameter block is filled by the bootloader. On the SCC, the alternative approach is to rely on proper default values in the kernel sources. This demands a set of mandatory compilation options to ensure the correct default values.

When the system passed the real-mode entry point, a standard Linux kernel attempts to query the BIOS for basic hardware information like the amount of installed memory. In the original SCC Linux kernel, all these calls have been removed and appropriate return values have been hard-coded.

B. Protected Mode Operation

For the proper adjustment of the protected mode operation, several special issues for clock management, interrupt management and cache handling must be considered.

For time management purposes, Linux distinguishes between clock sources and timers. A periodic or single-shot *timer* interrupts a core if the timeout elapses. Depending on whether a timer can interrupt all or only a single core, it is classified either as *broadcast* or *per-cpu* timer.

A *clock source* is a permanently updated counter with a constant frequency. Clock sources can be driven by the periodic timer interrupt in x86 systems, which leads to a limited resolution in millisecond range. Alternative clock sources are hardware units like the CPU's *Time-Stamp Counter (TSC)* or chipset devices such as a *High-Precision Event Timer (HPET)*.

In the early stages of the protected-mode boot process, the original Linux kernel assumes the system to have a broadcast timer at hand that generates interrupts on all cores. Per-cpu timers are then calibrated against this global timer. The SCC hardware does not provide such global timer functionality – the only timer available to cores is their local advanced programmable interrupted controller (LAPIC) timer. The kernel therefore needs an according modification to accept the per-cpu timer as the only authoritative source. In addition, the calibration of this timer normally relies on measuring the number of bus ticks per real time unit. Since the bus frequency is statically known on the SCC hardware, the modified LAPIC calibration routine can just return the static configuration value for the bus clock.

The SCC does not have an interrupt controller connected to the cores. Instead, all interrupts are delivered directly via the LINT0 and LINT1 processor pins [2]. The LAPIC-related operating system code needs two adjustments for the SCC hardware:

Firstly, when LAPIC support is enabled in the kernel, it assumes that LINT0 is connected to an according external chip, and LINT1 is used as the NMI. However, on the SCC, there is no external interrupt or NMI logic, and the current device drivers and MCPC software assume these pins to directly generate an interrupt. Secondly, the kernel crashes when attempting to query the base address of the LAPIC in memory, because the corresponding machine status register (MSR) is present only in later version of the Pentium architecture.

The modifications in SCC Linux consider these issues by configuring LINT0 and LINT1 as responsible for interrupt numbers 4 and 3, respectively, which are used for the SCC-specific device drivers.

Another novel feature of the SCC core design is the per-tile message passing support. This lead to a new data type in the caching hierarchy called *Message-Passing Buffer Type* (MPBT) and an according invalidation instruction. Support for this functionality is globally enabled by the *Message Buffer Enable* (MBE) bit in the architectural extension register (CR4); afterward, MPBT caching is enabled via the page-table by the *Message Buffer* (MB) bit. The MB bit shares its location with the *Page-Size* (PS) bit from regular P54C, which itself is enabled by the *Page-Size Extension* (PSE) bit in CR4. PS and MB cannot be used at the same time, so it is also invalid to set both architectural feature bits in CR4. Therefore, if MB shall be used, the Linux kernel must be prevented from setting PS and using large pages. As the kernel recognizes it is running on a P54C, it will incorrectly enable PSE support by default. This is prevented in Intel's SCC Linux via a kernel patch that masks out the corresponding bit from the core's feature bit mask. Afterward, MBE is set manually when the RockCreek-specific *rckmem* driver is loaded.

Even though hardware support for MPBT is enabled, drivers still need a means to specify this caching type for their memory accesses. Intel's SCC Linux exports an internal kernel function (*__ioremap*) to provide this functionality. It allows to specify individual bits for page table entries, in contrast to the regularly-exported *ioremap* and *ioremap_nocache*.

C. SCC Hardware Support

The SCC Linux software provides different driver implementations for accessing SCC-specific hardware features from user mode. The *RockCreek memory driver* (*rckmem*) allows applications to map arbitrary physical memory with configurable caching types, thus allowing direct access to all aspects of the system: message-passing buffers, voltage and frequency settings and configuration registers, both per-tile and globally. There are also two drivers for virtual network devices included: *rckpc* for communication with the MCPC, and *rckmb* for on-die communication between cores.

III. Portability Improvements for SCC Linux

Given the analysis result for the necessary operating system modifications, we developed a new set of SCC-enabling patches for the Linux 2.6 kernel series. Our modifications fit to the latest Linux kernel architecture at the time of writing (2.6.37), but also allow an easy consideration of future kernel versions for research purposes. The resulting kernel is fully compatible to existing SCC applications (e.g. RCCE [3]). Our approach focuses only on a very restricted set of source code locations, by introducing a *BIOS emulation* and a new *x86 sub-architecture*. We further propose extended SCC driver functionalities, as explained in the following sections.

A. The RockCreek sub-architecture

To distinguish processor- from mainboard-specific initialization code, the Linux kernel supports the concept of *sub-architectures* [4]. A sub-architecture defines code that is only used for a specific set of systems, typically sharing the same kind of mainboard architecture. It bundles mainboard-specific code in one place – the sub-architecture file – instead of spreading it over different parts of the kernel sources. Due to the nature of the SCC hardware specialties, the according kernel adjustments fit smoothly into this concept. We therefore introduced the "RockCreek" sub-architecture in our version of SCC Linux. Examples for existing sub-architectures include the Standard PC, AMD's Elan micro controller, and Intel's

"Moorestown" Mobile Internet Device. The latter is comparable to the SCC architecture, since it does not contain typical legacy PC peripherals. We therefore used it as template for the newly introduced "RockCreek" sub-architecture.

Sub-architectures are defined by a set of callback routines. Upon system start, the boot loader identifies which sub-architecture is present and specifies the corresponding value in the setup code's parameter block. We therefore modified the reset vector routine accordingly.

On kernel startup, the sub-architecture's initialization routine is responsible for installing its callback routines. This includes functions for initialization of broadcast and core-specific timers, initialization of hardware interrupts, TSC calibration, detection of i8042 keyboard controller / multiprocessor table / BIOS extension ROMs, and for the reservation of ISA resources.

1) Clocks: Since the SCC hardware does not implement a broadcast timer, we disabled the corresponding callback, but the only available clock source LAPIC cannot be configured via the sub-architecture mechanisms. In order to solve this issue, we used the static bus frequency approach as with the original SCC Linux. It must be noted that this missing callback is currently under work in the Linux kernel community. The TSC is calibrated using the same mechanism, but unlike the LAPIC timer, we can return the bus clock value from the centralized sub-architecture in this case.

By calibrating the clock sources in the described way, the kernel can be unintentionally triggered to detect an unstable TSC clock source. In such cases, the original kernel would switch over to *jiffie*-based clock counting, which results in low-quality millisecond resolution for the overall operating system. In order to disable this safeguard in a standardized way, we added the *tsc=reliable* option to the kernel command line.

2) Interrupts: The original SCC Linux modifications compensated the missing interrupt controller by configuring the LAPIC directly at boot time. Our version of the kernel adjustments perform the same operations, but as part of the *timers.setup_percpu_clockev* sub-architecture callback. This callback is invoked at a comparable point to the original modification in the control flow of the Linux startup process.

The x86 Linux kernel normally expects the system to have 8259A programmable interrupt controllers (PICs) in the default cascade configuration. Instead of commenting out only their usage in the sources, we used the sub-architecture features to switch of PIC utilization completely. This brings up the new issue that ISA interrupts 0 to 15 do not automatically get an interrupt vector number assigned. When the LAPIC sends the LINT0/1 interrupts in this state, the kernel does not have a corresponding mapping and cannot dispatch the call. To remedy this, we insert the mappings manually into the according data structures during the sub-architecture callback.

3) Message Passing Memory: If MPBT support is enabled, drivers need a way to express their demand for mapping message passing buffer memory. The original SCC Linux just exported an internal Linux kernel function (*__ioremap*) for this purpose. Since this function is no longer present on newer kernels, we chose to export our own function *ioremap_mpbt*, which then delegates to the internal kernel function responsible for this task. For Linux 2.6.37, *__ioremap_caller* fulfills this role; we extended it to support the MB page bit.

B. BIOS Emulation

As described in Section II-A, all references to BIOS functions were directly removed in the original SCC Linux. While this approach works for a fixed version of the sources, it has major implications for later kernel versions. The Linux setup code has been rewritten almost completely, from pure assembler in the 2.6.16 version to mostly C based routines starting from 2.6.17.

In order to realize a portable version of SCC enabled Linux, we chose not to modify the setup code directly, but instead to supply an emulation of the BIOS functions requested by the kernel. This requires two changes to the initial memory image:

1) An *Interrupt Vector Table* (IVT) needs to be installed at physical address 0, since BIOS calls are accomplished via software interrupts.
2) The *BIOS Emulation* code that implements the software interrupts needs to reside below +1MB, the highest address reachable in real-mode.

The corresponding binaries are integrated into the memory image by modifying the *load.map* file for Intel's SCC Linux tool chain. We manually created the IVT in a hex editor, as it just specifies addresses in *segment:offset* form – these addresses are located within the BIOS binary.

The setup code uses interrupt numbers *0x10* for the VGA BIOS, *0x15* for general BIOS services, *0x16* for keyboard and *0x1A* for CMOS access. However, only *0x15* needs to be implemented – all others can just return to their caller without a defined result (IRET), since the setup utilizes a suitable standard configuration in this case. Interrupt *0x15* acts as multiplexer for various functions, but only the physical memory configuration querying (*0xe820*) needs to return a valid result at the moment. Our BIOS emulation therefore returns the same three entries that are hard-coded into the original SCC Linux here: Two RAM ranges for node-local memory, and a reserved entry for the core's local APIC range (4KB starting at *0xFEE00000*).

The BIOS emulation concept allows us to dynamically construct the e820 memory map in future versions of the new SCC kernel. For example, sccKit 1.4.0 contains the size of node-local memory in a register in the FPGA, which our BIOS emulation could read to construct an appropriate memory map.

C. Driver adjustments

In addition to the new kernel modifications, we developed refactored versions of the original SCC device drivers.

Similar to SCC Linux, we provide the two major SCC network drivers – an *MCPC NIC Driver* (rckpc) for off-die and a *Message-Buffer NIC Driver* (rckmbx) for on-die communication. Both drivers were updated to fit to the most recent kernel interfaces.

Our *rckmbx* driver extends the original version by allowing different *kernel subsystems* to co-exist and send messages to other cores. These subsystems are situated below the socket layer in the kernel software hierarchy, and allow different message formats without using an IPv4 header. With this implementation strategy, transmission of IPv4 packets (that is provided by the original *rckmb* driver) is just a single subsystem implementation. Other subsystem identifiers can be used if desired. One example is our current research work on a cache-coherent memory driver, which can then communicate without involving the IPv4 protocol stack.

We further introduced the *RockCreek /proc Driver* (rckproc) that exposes low-level hardware information (like the contents of the TILEID register, or the current LUT mappings) via the */proc* pseudo file system. It allows applications to read these information via regular file accesses, instead of having to access the device drivers directly as with the original SCC Linux. With this modification, Intel's *pid* application that queries the coordinates of the current core on the SCC die can be rewritten as a simple shell script.

Another new driver, the *RockCreek Performance Meter Driver*, is intended to report the core's utilization to the MCPC. This functionality is necessary for sccGui's performance meter to recognize an active core. In the original SCC Linux, the GUI relies on a modified version of the CPUUTIL user-space program, which reads the values from the /proc file system and writes them to a statically known location in shared memory. Our implementation uses the same algorithm, but being a kernel driver, it can read the relevant statistics directly without having to go through the /proc interface.

D. Other issues

Several concepts of the original SCC Linux were kept in our version. Besides other things, we rely on the same bootstrap method as SCC Linux does, in order to pre-load the operating system image in main memory. We also replaced the proprietary initrd generation from SCC Linux with a standardized approach based on *BuildRoot*.

On standard x86 architecture, the CMOS RAM is accessed via the I/O ports of the real-time clock hardware. As this peripheral is not present on the SCC, we commented out the appropriate routines the same way as the original SCC Linux. There is no means, by either the kernel configuration or sub-architecture, to build a kernel without clock hardware access routines. It is possible to disable the according driver, but that only removes the user-visible *rtc* device, not the support code in the kernel.

Newer kernels provide a means to specify a built-in command line, so one does not need to rely on the boot loader to pass correct arguments. We use this feature for the *mmu=nopentium* option (see Section II-B) and the *tsc=reliable* option (see Section III-A1).

SCC Linux is based on the TinyCore Linux distribution. Since this distribution is optimized for a small memory footprint, its lacks a proper package management for easy access to advanced user mode tools. As alternative, we provide a SCC-compatible Linux distribution based on Gentoo as service for the MARC community. This distribution works both with the original SCC Linux kernel, and with the optimized kernel version presented here.

IV. Related Work

Porting Linux to alternative platforms is an established approach in operating system development and research [5]. Typical work focuses on real-time and embedded system hardware optimizations [6].

An alternative approach to Linux on the SCC is the Barrelfish operating system [7], whose satellite kernel approach matches the SCC hardware characteristics more specifically than that of the original Linux. As most of the operating system adjustments we have presented are hardware-related, we expect them to be of interest to Barrelfish as well. The same holds for all future developments that target the bare-metal environment, like the SCC bare metal framework developed at ETH Zürich.

CoreBoot [8] is an open-source BIOS implementation that currently supports over 230 mainboards from all major processor and chipset vendors. It could act as alternative to the BIOS emulation approach. Another option is the Unified Extensible Firmware Interface (UEFI) as provided by the TianoCore project [9].

V. Conclusion

In this paper, we analyzed the necessary operating system kernel modifications to run Linux on SCC processor cores. Our proposed improvements for such modification provide better compatibility with recent and future versions of the Linux kernel. The updated kernel architecture also acts as foundation for our ongoing SCC operating system research. Future work will focus on a scalable cache-coherent memory driver approach, and on a distributed hypervisor concept for single system image operation of the SCC.

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, and et al., "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," *2010 IEEE International SolidState Circuits Conference ISSCC*, vol. 9, pp. 58–59, 2010.

[2] Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, 1999.

[3] E. Chan, *RCCE comm: A Collective Communication Library for the Intel Single-chip Cloud Computer*, http://communities.intel.com/docs/DOC-5663, 2010.

[4] R. Love, *Linux Kernel Development*, 3rd ed. Pearson Education, 2010.

[5] C. yue Bi, Y. peng Liu, and R. fang Wang, "Research of key technologies for embedded Linux based on ARM," in *International Conference on Computer Application and System Modeling*, pp. 373–378.

[6] R. Lehrbaum, "Using Linux in Embedded and Real-Time Systems," *Linux Journal*, Jul. 2000.

[7] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the Barrelfish manycore operating system," in *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[8] Coreboot, http://www.coreboot.org.

[9] Intel Corporation, "TianoCore," http://www.tianocore.org.

Early experience with the Barrelfish OS and the Single-Chip Cloud Computer

Simon Peter, Adrian Schüpbach, Dominik Menzi and Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zurich

***Abstract*—Traditional OS architectures based on a single, shared-memory kernel face significant challenges from hardware trends, in particular the increasing cost of system-wide cache-coherence as core counts increase, and the emergence of heterogeneous architectures – both on a single die, and also between CPUs, co-processors like GPUs, and programmable peripherals within a platform.**

The *multikernel* is an alternative OS model that employs message passing instead of data sharing and enables architecture-agnostic inter-core communication, including across non-coherent shared memory and PCIe peripheral buses. This allows a single OS instance to manage the complete collection of heterogeneous, non-cache-coherent processors as a single, unified platform.

We report on our experience running the Barrelfish research multikernel OS on the Intel Single-Chip Cloud Computer (SCC). We describe the minimal changes required to bring the OS up on the SCC, and present early performance results from an SCC system running standalone, and also a single Barrelfish instance running across a heterogeneous machine consisting of an SCC and its host PC.

I. Introduction

The architecture of computer systems is radically changing: core counts are increasing, systems are becoming more heterogeneous, and the memory system is becoming less uniform. As part of this change, it is likely that system-wide cache-coherent shared memory will no longer exist. This is happening not only as specialized co-processors, like GPUs, are more closely integrated with the rest of the system, but also as core counts increase we expect to see cache coherence no longer maintained between general purpose cores.

Shared-memory operating systems do not deal with this complexity and among the several alternative OS models, one interesting design is to eschew data sharing between cores and to rely on message passing instead. This enforces disciplined sharing and enables architecture-agnostic communication across a number of very different interconnects. In fact, experimental non-cache-coherent architectures, such as the Intel Single-Chip Cloud Computer (SCC) [1], already facilitate message passing with special hardware support.

In this paper, we report on our efforts to port Barrelfish to the SCC. Barrelfish is an open-source research OS developed by ETH Zurich and Microsoft Research and is structured as a *multikernel* [2]: a distributed system of cores which communicate exclusively via messages.

The multikernel is a natural fit for the SCC that can fully leverage the hardware message passing facilities, while requiring only minimal changes to the Barrelfish implementation for a port from x86 multiprocessors to the SCC. Furthermore, the SCC is a good example of the anticipated future system types, as it is both a non-cache coherent multicore chip, as well as a host system peripheral.

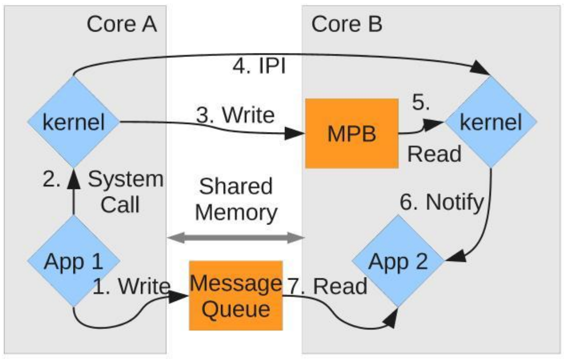


Fig. 1. Sending of a message between SCC cores

We describe the modifications to Barrelfish's message-passing implementation, the single most important subsystem needing adaptation. We give initial performance results on the SCC and across a heterogeneous machine consisting of an SCC peripheral and its host PC.

II. Message Passing Design

Message passing in Barrelfish is implemented by a *message passing stub* and lower-level *interconnect* and *notification drivers*. The message passing stub is responsible for (un-)marshaling message arguments into a message queue and provides the API to applications. Messages are subsequently sent (received) by the interconnect driver, using the notification driver to inform the receiver of pending messages. Messages can be batched to reduce the number of notifications required.

Message passing is performance-critical in Barrelfish and thus heavily tied to the hardware architecture. In this section, we describe the interconnect and notification driver design between cores on the SCC, as well as between host PC and the SCC. We mention changes to the message passing stub where appropriate.

A. SCC Inter-core Message Passing

The SCC interconnect driver reliably transports cache-line-sized messages (32 bytes) through a message queue in non-coherent shared memory. Shared memory is accessed entirely from user-space, shown by steps 1 and 7 in Figure 1, using the SCC write-combine buffer for performance. The polling approach to detect incoming messages, used by light-weight

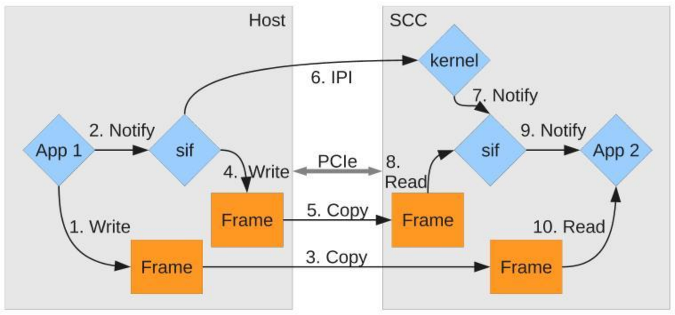


Fig. 2. Sending of a message from host to SCC

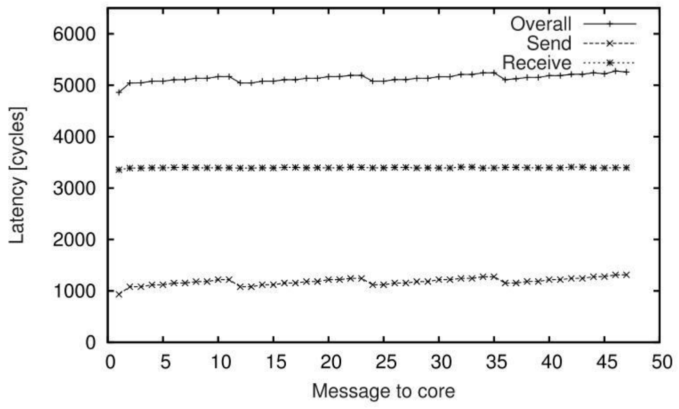


Fig. 3. Average notification latency from core 0 (*Overall*). *Send* and *Receive* show time spent in sending and receiving, respectively.

message passing runtimes, such as RCCE [3], is inappropriate when using shared memory to deliver message payloads, since each poll of a message-passing channel requires a cache invalidate followed by a load from DDR3 memory.

Consequently, the notification driver uses inter-core notifications, implemented within per-core kernels, to signal message arrival. Notifications are sent by a system call (2) via a ring-buffer on the receiver's on-tile message passing buffer (MPB) and reference shared-memory channels with pending messages (3). An inter-core interrupt (IPI) is used to inform the peer kernel of the notification (4), which it forwards to the target application (6).

At first sight, it may seem odd to use main memory (rather than the on-tile MPB) for passing message payloads, and to require a trap to the kernel to send a message notification. This design is motivated by the need to support many message channels in Barrelfish and more than one application running on a core. The SCC's message-passing functionality does not appear to have been designed with this use-case in mind. We discuss this issue further in Section IV.

B. Host-SCC Message Passing

The SCC is connected to a host PC as a PCI express (PCIe) device and provides access to memory and internal registers via a system interface (SIF). The host PC can write via the SIF directly to SCC memory using programmed I/O or the built-in direct memory access (DMA) engine.

The interconnect-notification driver combination used between host PC and SCC, called SIFMP, employs two proxy drivers. One on the host, and one on the SCC. New SIFMP connections are registered with the local proxy driver. When the interconnect driver is sending a message by writing to the local queue (1), the notification driver notifies the proxy driver (2), which copies the payload to an identical queue on the other side of the PCIe bus (3). The proxy driver then forwards the notification to the receiver of the message via a private message queue (4, 5), sending an IPI to inform the receiving driver of the notification via its local kernel on the SCC (6, 7). The peer proxy reads the notification from its private queue (8) and forwards it to the target application (9), which receives the message by reading the local copy via its interconnect driver (10). This implementation, shown in Figure 2, uses two message queues (one on each side) and two proxy driver connections (one for each driver) for each SIFMP connection.

III. Evaluation

We evaluate message passing efficiency by running a series of messaging benchmarks. All benchmarks execute on a Rocky Lake board, configured to 533MHz core clock speed, 800MHz memory mesh speed and 266MHz SIF clock speed. The host PC is a Sun XFire X2270, clocked to 2.3GHz.

A. Notification via MPB

We use a ping-pong notification experiment to evaluate the cost of OS-level notification delivery between two peer cores. Notifications are performance critical to notify a user-space program on another core of message payload arrival. The experiment covers the overheads of the system call required to send the notification from user-space, the actual send via the MPB and corresponding IPI, and forwarding the notification to user-space on the receiver.

Figure 3 shows the average latency over 100,000 iterations of this benchmark between core 0 and each other core, as well as a break-down into send and receive cost. As expected, differences in messaging cost due to topology are only noticeable on the sender, where the cost to write to remote memory occurs. The relatively large cost of receiving the message is due to the direct cost of the trap incurred by the IPI, which we approximated to be 600 cycles, and additional much larger indirect cost of cache misses associated with the trap.

B. Host-SCC Messaging

We determined the one-way latency of SIFMP for a cache-line size message from host to SCC to be on the order of 5 million host cycles. As expected from a communication channel that crosses the PCIe bus, SIFMP is several orders of magnitude slower than messaging on the host (approximately 1000 cycles). To gain more insight into the latency difference, we assess the performance of the PCIe proxy driver implementation, by evaluating read access latency of varying size from SCC memory to the host PC, using DMA.

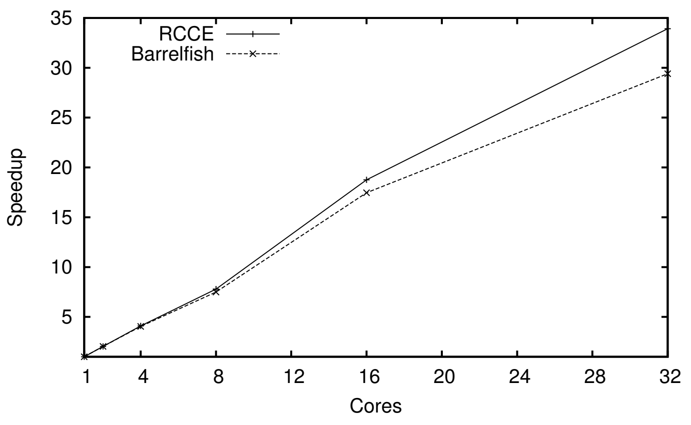


Fig. 4. RCCE LU benchmark speedup comparison

The results show a baseline read overhead of about 500,000 host clock cycles, which is 10% of the messaging overhead. Thus, PCIe bus latency explains only a fraction of the measured messaging overhead and we have yet to determine the cause of these overheads.

C. *Application-level benchmarks*

The standard software environment available on the SCC uses RCCE [3], a library for light-weight communication, highly optimized for this platform, requiring exclusive access to the MPB. We implemented a substrate supporting the RCCE message-passing interface using Barrelfish stubs and interconnect drivers for messaging, which multiplexes MPB access to applications. We evaluate the LU benchmark shipped with RCCE to compare the performance achieved on Barrelfish to that of RCCE.

From the result, shown in Figure 4, we can see that, at the application level, Barrelfish shows only slightly lower performance and scalability than direct MPB access via RCCE. This overhead is due to multiplexing and the early version of our port, which we seek to improve.

IV. Discussion

The port of the IA-32 version of Barrelfish to the SCC required 2235 lines of SCC-specific C code and 130 lines of assembly, only 17% of which execute in privileged mode. By and large, the bring-up for SCC was straightforward and took about 2 person-months. The prototype is fully functional and shows adequate initial application performance.

We experienced no significant problems with Barrelfish due to the lack of coherent caches on the SCC. This was not a big surprise to us, but it was a confirmation of our expectations, and a validation of the OS design. We regard the lack of coherent caches as a useful feature from a research perspective.

In this section, we cover the most influential architectural issues to our design and discuss possible improvements in either software or hardware.

A. *Cache issues*

The caches do not allocate a cache line on a write miss, treating it as an uncached write to memory. Furthermore, a core is allowed only one outstanding write transaction; when such a write miss occurs, any subsequent memory or L2 cache access causes the core to stall until the write to memory completes (typically around 100 cycles). Combined with the lack of a store buffer, this policy causes severe performance degradation for word-sized writes to data not already present in the cache. When storing to a fresh stack frame, or saving registers in a context switch path, each individual memory write instruction will stall the processor.

For example, in Barrelfish kernel code, we observed that simple function calls in hot paths of the system regularly have an order of magnitude greater overhead (in cycles) on SCC compared to newer x86 processors. Our code is not optimized for this behavior and shows major inefficiencies, in particular on function call boundaries immediately after kernel crossings. Our tentative explanation is that caller-saved registers will be pushed onto the stack upon a function call and then restored upon return from the function. Both cases miss in the cache, but the call incurs particularly high overhead, as each individual write goes to memory, and the cache lines are only allocated when reading them on return. We have also observed substantially increased costs for exception and trap handling, which save register state to memory not commonly present in the cache.

It is possible that an OS workload is a particularly bad case for this cache design. More work is required to both confirm this as the cause, and explore possible solutions. Ideally this could be fixed in hardware, through changes to the cache architecture, the addition of a store buffer, or simply allowing the write-combining buffer to be used for non-message-buffer memory, which would mitigate the problem by allowing full cache-line writes to memory. A possible software fix would involve reading from each cache line before it was written, to ensure its presence in the cache; in the case of context save code this could be done explicitly, but for stack access would probably require compiler modifications.

B. *Message-passing memory*

The ability to bypass the L2 cache for areas of address space designated as messaging buffers (MB), combined with an efficient L1 invalidation of all such lines, is one of the most interesting features of the SCC.

As with other message-passing features of the SCC, this functionality may have been designed with a single-application system in mind. When using MB memory for the operating system, as in Barrelfish, we typically have a number of communication channels in use at any one time. For this reason, although the CL1INVMB instruction is extremely cheap to execute, its effects may be somewhat heavyweight, since it invalidates all messaging data, some of which we may wish to have remain in the L1 cache.

In our message-passing implementations, we generally know precisely which addresses we wish to invalidate. Consequently, we would find more fine-grained cache control very useful. An instruction which would invalidate a region around a given address would be ideal for us.

Better still would be to extend such functionality to the L2. Receiving data in an MPB generally involves an L1 miss (ideally to the on-tile MPB, but see below why this is problematic), followed by a miss to main memory caused by copying the data somewhere where it can be cached in L2, followed by a second L2 miss when the data needs to be subsequently read, due to the non-allocation policy of the cache on a write miss. The final penalty can be mitigated somewhat by performing a read of the destination location (and so populating the L2) before writing the received data.

This coarse-grain cache access can have far-reaching implications for applications. For example, in its current form the cache architecture seems to prohibit any efficient zero-copy I/O implementation, since if the message passing buffers are used, any cached data will be invalidated any time further I/O occurs. Our position overall is that explicit cache management is good, and Barrelfish (and, we believe, other OS code) would benefit from future SCC implementations providing much more fine-grained control over it.

C. The on-tile message passing buffer

We experienced two significant challenges when using the on-tile message passing buffers on SCC. These challenges are related, but different.

1) Size: The small size of 8192 bytes constrains the size of message queues. If messages cannot be lost (a typical design assumption for message-passing applications, including Barrelfish), this results in blocking and tighter coupling between communicating processes.

Barrelfish uses message-passing throughout for communication, and consequently requires a large number of independent message channels to share the MPB. This leads to the question of how to allocate space in the MPB to message channels. Obviously, the space is too small to reserve parts for message payload. For example, reserving 8 cache lines for each message queue would allow only 32 message channels per core, which is impractical.

It is conceivable to multiplex all Barrelfish channels onto a single channel per pair of cores, requiring only 47 channels in the MPB and privileged code to demultiplex incoming messages. This allows 170 bytes of buffer per core pair, still small, but possibly useful for small messages. The downside to this approach is a kernel crossing and increased contention.

Finally, message payload could be written to the MPB per application, but this increases contention even further. In both cases, direct payload access by applications is prohibited, as the MPB has to be freed up as quickly as possible to reduce contention. This requires copying the message in and out of the MPB, which is a costly operation.

2) Multiplexing: An operating system must mediate access to the MPBs to ensure safe sharing of the buffers between applications. Unfortunately, the very high MPB access performance is completely dominated by the cost of kernel crossing to validate the access. As an additional cost, since multiple cores can be expected to be accessing each tile's MPB, write access to each core's memory must be done under a lock.

Like most resources, the MPBs can be multiplexed in both in space and time.

Space-multiplexing the MPBs requires a high-performance protection mechanism to divide the buffer between applications or other resource principals. For main memory, this is performed by each core's MMU. The P54C chip used in the SCC provides hardware protection of memory segments of sizes smaller than a page and thus could be used to space-multiplex the MPB when small messages are sufficient. However, the size of the MPB prohibits space multiplexing of larger message queues.

Time-multiplexing the MPBs requires copying each application's state into or out of the MPBs on a context switch, or performing this lazily. This is potentially 8KB of message data, a substantial context-switch overhead when caches do not allocate on a write miss.

Unlike memory which is under the exclusive use of an application, memory used for communication between applications on different cores is shared between a pair of principals. Time-multiplexing on-tile MPB memory in software is possible via co-scheduling [4] of communicating principals on different cores. However, this constrains the system-wide schedule and requires considerable communication overhead in itself [5].

In addition, our experience with Barrelfish so far suggests that some kind of inter-core notification mechanism is an important complement to polled message-passing support. The fact that we can access interrupt pins on cores remotely on the SCC is very nice in this regard, but even better would be some kind of fast user-space control transfer. One option is to introduce address space identifiers (these should be orthogonal to virtualization in any case), and cause a lightweight same-address-space jump if and only if that address space is running.

D. System Interface

A number of different approaches are possible for communication across a PCIe bus, such as using only a single driver in the host PC which controls all memory operations on the SCC. While this approach is both simpler and more resource efficient than our current implementation, the lack of notification mechanism from the SCC to the host PC makes it untractable. The host PC has to poll every possible message channel in SCC memory, incurring a huge performance overhead. The double proxy approach reduces this cost by requiring only one channel to be polled.

To be able to better leverage a single image OS across both host and peripheral and facilitate message passing across the PCIe bus, we are missing an efficient notification mechansim from SCC to host. Given PCIe bus latencies, a simple PCI device interrupt would be sufficient.

V. Related Work

Helios [6] introduces the concept of satellite and coordinator kernels to enable heterogeneous computing. Satellite kernels are light-weight run-times that execute on the peripheral and provide a limited set of services. System calls pertaining to functionality implemented in the coordinator kernel are relayed to the host PC. This enables applications and operating system components to run unmodified on the peripheral. Barrelfish has no concept of satellite kernels and follows a fully distributed model.

The standard operating system model for the SCC treats the chip as a cluster of independent machines and runs a complete Linux operating system on each core that communicate via the TCP/IP network protocol [3]. Using TCP/IP to communicate incurs high overheads for unnecessary functionality, such as message fragmentation, replay and sequencing. Furthermore, executing a complete OS on each core has high memory overhead, while complicating global resource management, which has to be carried out by coarse-grained user-level cluster resource management software.

VI. Conclusion

We have demonstrated that it is possible to run a single image OS across a heterogeneous, non-cache-coherent machine consisting of an SCC and its host PC with reasonable performance (the remaining issues with our PCIe interconnect driver notwithstanding).

Our experience so far has provided insight into messaging performance on the SCC when the on-tile message buffers have to be multiplexed by an operating system. Barrelfish is a work in progress, and we believe that we can improve on the performance numbers presented here. Nevertheless, we feel our SCC experience provides useful insights for future designs of more OS-friendly message-passing hardware.

In addition to performance, asymmetry and latency will continue to be issues for future multicore architectures. We are looking at using improved scheduling to address this challenge, using the SCC port of Barrelfish as a research vehicle. For example, threads that communicate or synchronize frequently should not be placed on cores separated by a high latency link, and the cost of transferring a program and its working set to a different core should not outweigh the expected speed-up.

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *International Solid-State Circuits Conference*, Feb. 2010, pp. 108–109.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Oct. 2009.

[3] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.

[4] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 306–318, 1992.

[5] J. Ousterhout, "Scheduling techniques for concurrent systems," in *IEEE Distributed Computer Systems*, 1982.

[6] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *Proceedings of the 22nd ACM Symposium on Operating System Principles*, 2009, pp. 221–234.

On Mapping Distributed S-NET to the 48-core Intel SCC Processor

Merijn Verstraaten, Clemens Grelck, Michiel W. van Tol, Roy Bakker, Chris R. Jesshope
Informatics Institute, University of Amsterdam
Science Park 904, 1098 XH Amsterdam, The Netherlands

***Abstract*—Distributed S-NET is a declarative coordination language and component technology primarily aimed at modern multi-core/many-core chip architectures. It builds on the concept of stream processing to structure dynamically evolving networks of communicating asynchronous components. These components themselves are implemented using a conventional language suitable for the application domain. Our goal is to map Distributed S-NET to the Intel SCC processor in order to provide users with a simplified programming environment, yet still allowing them to make use of the advanced features of the SCC architecture.**

Following a brief introduction to the design principles of S-NET, we sketch out the general ideas of our implementation approach. These mainly concern the use of SCC's message passing buffers for lightweight communication of S-NET records and control data between cores as well as remapping of large data structures through lookup table manipulation. The latter avoids costly memory copy operations that would result from more traditional message passing approaches. Last, but not least, we present prototypical performance measurements for our communication primitives.

I. Introduction

The Single-chip Cloud Computer (SCC) experimental processor is a *concept vehicle* created by Intel Labs as a platform for many-core software research. It provides 48 P54C cores, an on-chip message passing network, non cache-coherent off-chip shared memory and dynamic frequency and voltage scaling on different subsets of cores [1]. Creating programs for systems which support large amounts of parallelism is already difficult; even in the absence of specific exploitation of the power saving features. Trying to also utilise these power management features at the same time complicates the programmer's job considerably.

Currently RCCE [2] (including the community contributed iRCCE [3]) and RCKMPI [4] are available for programming the SCC. RCCE is an SCC-specific low-level message passing library and RCKMPI is an adaptation of MPI largely based on RCCE. To the best knowledge of the authors, no higher level programming environments are (yet) available on the SCC. While message passing is a very efficient way programming it also requires a lot of attention to detail from the programmer. Our goal is to make this task simpler.

S-NET[1] [5], [6], [7] is a coordination language whose aim is to simplify the programming of parallel systems. As a pure coordination language S-NET provides no features to express any sort of computation; it relies on an auxiliary language for the implementation of sequential, state-less components. S-NET in turn organises the interaction of these components — called boxes — in a streaming network. A type system on streams gives essential static guarantees on the behaviour of S-NET streaming networks.

Our two step approach, internally sequential components on the one hand and orderly component interaction, somewhat reflects the hardware design principle of the SCC and other contemporary multicore chip architectures: cores that were originally designed as central processing units combined on a single die.

This paper reports on our on-going efforts to map S-NET to the Single Chip Cloud Computer as an alternative programming environment. Starting from our MPI-based distributed implementation of S-NET [7] the obvious choice is to simply use RCKMPI as underlying middleware layer. However, the performance of MPI is less than what can be achieved by an implementation which uses the SCC's hardware features directly. Additionally MPI restricts us to a static number of nodes, which is something which is undesirable in light of future development directions. Instead we focussed our efforts on implementing asynchronous message passing using the message passing buffers (MPBs) as communication channel and using interprocessor interrupts as out-of-band notifications. To accomplish this we had to come up with a way for our userspace code to receive the interrupts which are being trapped by the kernel.

In this paper we will analyse the approaches we considered and their impact on our goal of having a programming model which is more convenient to use than writing programs directly using RCCE or MPI, while still capable of using the features provided by the SCC. We will begin with an overview of both S-NET (Section II) and Distributed S-NET (Section III) before we discuss the effects of the SCC's design on our design (Section IV). We will discuss our initial measurements in Section V and conclude in Section VI.

II. S-Net

S-NET turns functions written in a standard programming language (e.g. C) into asynchronously executed, stateless stream-processing components termed *boxes*. Each box is connected to the rest of the network by two typed streams, one for input and one for output. Messages on these typed streams are organised as non-recursive records, i.e. sets of label-value pairs. The labels are subdivided into *fields* and *tags*. Fields are

[1]The development of S-Net has been funded by the European Union through the FP-6 project Aether and the FP-7 project Advance.

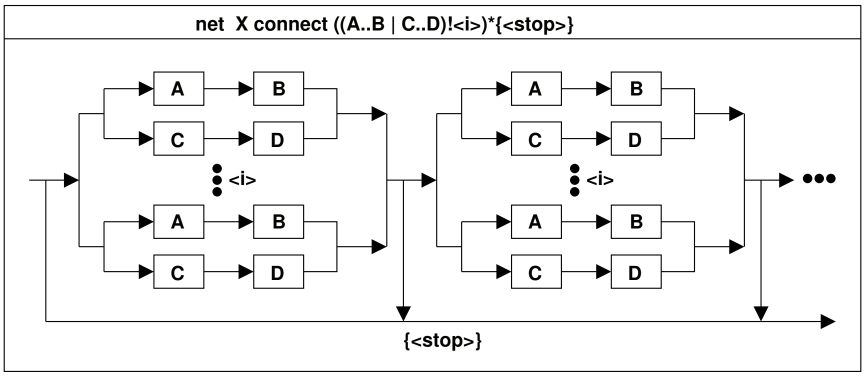


Figure 1. Example of an inductive streaming network constructed using network combinators. A, B, C and D denote boxes or networks defined elsewhere. Then A..B|C..D denotes the subnetwork where any record either goes through A and then B or through C and then D. The actual routing depends on the types of A, B, C and D and is left out in this figure. The whole subnetwork is then replicated in parallel based on the index i carried along by all records in the system. At last, this network is replicated serially with the presence of stop tag acting as a dynamic drop-out condition for a records after having passed each instance of the replicated network.

associated with values from the box language domain, they are entirely opaque to S-NET. Tags are associated with integer numbers that are accessible on both the coordination and on the box level.

A box triggers when it receives a record on its input stream, the box then applies its box function to that record. In the course of function execution the box may output records to its output stream. Once the function has finished the S-NET box is ready to receive and process the next record on the input stream.

On the S-NET level a box is characterised by a *box signature*; a mapping from an *input type* to a disjunction of types, named the *output type*. For example,

```
box foo ((a,<b>) -> (c) | (c,d,<e>));
```

declares a box that expects records with a field labeled a and a tag labeled b. The box responds with an unspecified number of records that either have just field c or fields c and d as well as tag e. The associated box function foo is supposed to be of arity two: the first argument is of type void* to qualify for any opaque data; the second argument is of type int.

The box signature naturally induces a *type signature*. For a proper specification of the box interface it is essential to have a concrete ordering of fields and tags. However, on the S-NET level we ignore the ordering when reasoning about boxes: treating them as sets of labels instead of tuples of labels. Hence the type signature of box foo is {a,<b>} −> {c} | {c,d,<e>}.

This type signature states foo accepts *any* input record that has *at least* field a and tag b, but may well contain further fields and tags. The formal foundation of this behaviour is *structural subtyping* on records. Any record type t_1 is a subtype of t_2 iff $t_2 \subseteq t_1$. This subtyping relationship extends to multivariant types. A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$.

Subtyping on input types of boxes raises the question what happens to the excess fields and tags. These are not just ignored in the input record of a network entity, but are attached to any outgoing record produced by it in response to that record. Subtyping and flow inheritance are indispensable when it comes to getting boxes that were designed separately to work together in a streaming network.

It is a distinguishing feature of S-NET that it neither introduces streams as explicit objects nor defines network connectivity through explicit wiring. Instead, it uses algebraic formulae to describe streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-NET provides four network combinators: static serial and parallel composition of two networks and dynamic serial and parallel replication of a single network. These combinators preserve the SISO property; any network, regardless of its complexity, again is a SISO entity.

Let A and B denote two S-NET networks or boxes. Serial combination (A..B) constructs a new network where the output stream of A becomes the input stream of B, and the input stream of A and the output stream of B become the input and output streams of the combined network, respectively. As a consequence, A and B operate in pipeline mode.

Parallel combination (A|B) constructs a network where incoming records are either sent to A or to B and the resulting record streams are merged to form the overall output stream of the combined network; the type system controls the flow of records. Each network is associated with a type signature inferred by the compiler. Any incoming record is directed towards the subnetwork whose input type better matches the type of the record. If both branches match equally well, one is selected non-deterministically.

The parallel and serial combinators have infinite counterparts: serial and parallel replicators for a single subnetwork. The serial replicator A*type constructs an infinite chain of replicas of A connected by serial combinators. The chain is inspected before every replica to extract records that match the type specified as the second operand.

The parallel replicator A!<tag> also replicates network A infinitely, but the replicas are connected in parallel. All incoming records must carry the tag; its value determines the replica to which a record is sent.

Fig. 1 shows an example for the use of network combinators to construct streaming networks.

Whereas boxes can easily split one incoming record into multiple records sent to the output stream one after the other, the opposite operation, i.e. merging two records on the input stream to form a single record on the output stream, is not possible with the means of S-NET introduced thus far. Merging independent records is the essence of synchronisation in the streaming context of S-NET. To isolate this coordination-level feature as far as possible from the box or application level, S-NET uses a special built-in box for this purpose called *synchrocell*. For example, the *synchrocell* {[|{ a,b},{c,d }|]} awaits records containing fields a/b and c/d and merges them into a record containing all four fields. For an exact definition of synchrocells and an extended discussion of the whole issue of synchronisation in the context of S-NET see [8].

We refer interested readers to [6], [5] for a detailed account of the design of S-NET and to [9], [10] for application examples.

III. DISTRIBUTED S-NET

S-NET as described so far is an abstract notation for streaming networks of asynchronous components. There is no notion of computing resources in S-NET, nor does S-NET make any specific assumptions about the execution environment.

Distributed S-NET [7] is a conservative extension of S-NET that introduces the concept of abstract compute nodes as an organisational layer on top of the logic network of boxes defined by standard S-NET.

We introduce two *placement* combinators. *Static placement* places the execution of a box or a network onto a logical node. *Indexed dynamic placement* places the execution of a box or a network onto a logical node on a per-record basis, based on a specific tag of that record. More precisely, each record is routed through a replica of the box or network instantiated on the relevant logical node.

We deliberately restrict ourselves to plain integer values for identifying nodes to retain the advantages of an abstract model as far as possible. The concrete mapping of numbers to machines is implementation-dependent. Our prototype implementation of Distributed S-NET is based on MPI where numbers correspond to MPI task identifiers.

Implementation-wise, Distributed S-NET takes care of initially setting up and dynamically maintaining disconnected S-NET streaming graphs on multiple nodes. Input and Output managers at node-boundaries render the distributed memory transparent by automatically serialising, transmitting and deserialising S-NET records when moving from one node to another.

While tags are easily sent around the network, field data can potentially be large. In this case serialisation, transmission and deserialisation at every node boundary between the creating node and the consuming node of some field would be costly. We instead transmit a qualified reference which allows the actual data associated with some field to be fetched, on-demand, from its current location to where it is needed.

Interested readers are referred to [7] for a more thorough introduction of the design and implementation of Distributed S-NET. Several case studies in [7] as well as [11] illustrate that implementing real-world distributed applications in Distributed S-NET is easier than using a low-level message passing middleware directly.

IV. S-NET ON THE SCC

Our initial work focussed on preparing the S-NET codebase to support multiple implementations of its distribution layer, implementing and experimenting with communication primitives on the SCC and later, when it became available, getting the current MPI implementation running on the SCC using RCKMPI. As mentioned in Section II, the MPI implementation of Distributed S-NET copies (potentially large) application data structures by (de-)serialising them over MPI messages. This is an expensive operation which can and should be avoided on a shared memory machine like the SCC.

Instead of copying it should be much faster to remap the memory from one core to another using the programmable lookup tables (LUTs). These control the translation of memory requests from "virtual" addresses to actual physical memory addresses. This means that changing an entry in the LUTs lets us move data in and out of a core's visible memory space without having to actually copy or move the data.

One problem with this approach is that the LUTs are behind the L2 cache. This mean cache hits and misses are checked *before* the final physical address look up is done. Hence, the L2 cache may contain stale data, e.g. the cache contains data at virtual address x (which translates to physical address y), meanwhile the LUTs have been changed so virtual address x translates to physical address z. When the core attempts to read from virtual address x it will still read the cached data at y instead of z.

This isn't a problem if it is possible to — efficiently — invalidate or flush the relevant L2 entries (or the entire L2). However, since the P54C cores of the SCC never had an L2 cache they don't have such an instruction. There is an L2 reset pin available on the machine, but using it crashes the core. This leaves two possible solutions for remapping memory: marking the used address range as uncacheable or flushing the L2 after every remap. Since there is no flush instruction the only way to accomplish a flush is to manually read in a full cache worth of data, thereby evicting the entire contents of the cache.

Aside from the big field data S-NET also sends a large amount of smaller meta-data messages consisting of tags. We want to be able to quickly deal with these messages, while wasting as little time as possible on this "non-work" computation.

Secondly, we need to be able to deal with multiple incoming message queues. This is important because S-NET utilises fixed size buffers, which propagate back pressure through the network, to prevent a box which produces huge number of output records from overwhelming the network.

If communication to one of the incoming queues can also block communication to other queues, then the network suddenly becomes susceptible to deadlocks. For example, imagine we use a synchronous communication method (like RCCE) or an asynchronous communication method which uses the MPB as a single queue for all incoming messages. When a node blocks on a send — because it is synchronous or the receiving MPB queue is full — the network deadlocks when further progress of the network depends on the same node (further down the network) doing a receive.

Since a node can have an arbitrary number of incoming message queues, the MPB with its limited size is not suitable for holding all incoming queues for a node. A node should move incoming messages from the MPB to queues in its own memory as soon as possible to keep receive space free. There are two obvious mechanism for a core to know when to move messages from the MPB: the core can either occasionally poll its MPB to see if there is anything new or we can use an out-of-band signal, in the form of an interprocessor interrupt.

The downside of polling is that when done infrequently it can take too long to make room in the MPB for new messages. If done too frequently time and energy are wasted. The interrupt based approach means having to context switch to the kernel to handle the interrupt and then context switch back to the program. The viability of this later option depends on the cost of a context switching.

Once the more basic work of getting S-NET running on the SCC is finished we intend to investigate the possibilities of the power management features. Since not all tasks take the same time and S-NET propagates back pressure through the network; there will always be one or more bottlenecks in a network, potentially leaving the parts of the network which are behind this bottleneck idle. The runtime system should be capable of lowering the clock speed on these idle nodes to save power, while increasing it on the bottleneck nodes to increase throughput of the network.

This functionality could then be extended to take into account that S-NET networks are dynamic, they grow and shrink as the computation moves through phases. Dynamically splitting and merging networks into smaller or bigger components would allow the runtime to allocate more cores to bottleneck tasks or deallocate cores from idle components, potentially even shutting down cores entirely or starting them up as computation demands.

V. MEASUREMENTS

As discussed in the previous section, there are only two ways to deal with the possibility of stale data in the L2 cache when remapping memory using the LUTs. Manually flushing the entire cache or marking the memory as uncacheable. Our measurements show that it takes approximately 1 million cycles to flush the cache manually; plus the cost of evicting still useful data, which has to be fetched from memory again.

The difference between L2 cache hits and L2 cache misses/uncached memory is a factor 6 writing speed reduction. Uncached memory and L2 cache misses write at a speed of up to 20 MB/s, cache hits write at up to 125 MB/s. For reading the speed difference is a factor 14. Uncached memory and L2 cache misses read at up to 20 MB/s whereas cache hits read at up to 285 MB/s.

The uncached speeds translate to a read and write speed of approximately 102 cycles per 4 bytes, or 25.5 cycles/byte. This means it costs 51 cycles in total to transfer one byte using uncached memory, from here we can see that the performance penalty of disabling caches start to outweigh the 2 million cycles required to flush two L2 caches once we want to transfer more than 38 KB.

Hops	Remote	Kernel	Roundtrip
0	2.8	4.6	7.4
1	2.9	4.5	7.5
2	3.0	4.6	7.6
3	3.0	4.7	7.7
4	3.1	4.7	7.8
5	3.0	4.6	7.6
6	3.0	4.7	7.7
7	3.0	4.6	7.6
8	3.1	4.6	7.7

Table I
INTERPROCESSOR INTERRUPT LATENCIES (IN 1000 CYCLES)

The Barrelfish developers already did some measurements [12] with regard to the speed of notifications using the MPBs and the latencies of interprocessor interrupts (IPI). But since we plan to run in user space under Linux, unlike Barrelfish, we are interested in how much overhead we would incur doing this. We did all our measurements on a system running the cores at 533 MHz and mesh network and DDR controllers running at 800 MHz.

To free up one of the two available interrupt pins we modified the rckmb driver to expose a kernel parameter (using sysfs) which lets us switch between interrupt-driven and polling mode. Next we wrote a simple Linux kernel module which would register a handler for the freed interrupt. This handler simply sends a POSIX signal to a user process; this way we can use interrupts to effectively send POSIX signals between processes running on different cores. Which signal is send and to which user process it is sent is configured via kernel parameters exposed using sysfs.

After the kernel module was done we had the basic infrastructure needed to test the latencies of IPIs. Each core, except core 0, was running a simple program with an infinite loop doing nothing and an installed signal handler which would raise an interrupt on core 0. Core 0 was running a program which would run through 100,000 iterations of the below steps for every core:

1) Read the cycle counter
2) Raise an interrupt on the other core
3) Read the cycle counter upon entering the interrupt handler in the kernel
4) Read the cycle counter again upon entering the process' signal handler
5) Read the cycle count read inside the kernel in using the sysfs

These tests give us three cycle counts for each iteration, from which we can compute three time intervals: the time it took from sending the interrupt to getting a return interrupt from the remote core, the time from entering the kernel locally to entering the user space signal handler and the full roundtrip time. Table I shows the averages of these measurements under the columns “Remote”, “Kernel” and “Roundtrip”.

As the table shows, the difference in hops between cores has a negligible impact on the latency. This is as expected when comparing the network's high speed network with the relatively slow core speed. However, it is surprising to see the difference between the remote and kernel columns. The former shows the time between raising an interrupt on the remote core, the remote core trapping to the kernel, running its interrupt handler, signalling the user space process and sending an interrupt back. This took around 3000 cycles on average. This is peculiar because the kernel time, the time between core 0 trapping into the kernel and the user space process receiving the signal, took around 4700 cycles and this time is also included in the remote time measurement.

Some additional tests show that this difference always appears between the initiating and replying core, the core who receives the reply and prints out measurements always takes more time to go from the kernel interrupt handler to the user space signal handler.

A possible explanation for this behaviour is that the remote kernel is idle aside from running the interrupt handler and the process' signal handler, whereas the local kernel has to access the filesystem and execute other system calls to retrieve the cycle counts stored by the kernel's interrupt handler. This is done outside of the reads from the cycle counter, so it should not impact the measurements, but it might still evict parts of the working set from the L1 and L2 caches causing a slow down by needing to go to main memory.

In [12] the Barrelfish developers measure a roundtrip latency of approximately 5000 cycles on bare metal, though this time includes reading and writing to MPB memory twice, making their measurements time higher than the raw IPI roundtrip time they had. This means the overhead of running in user space under Linux is around 2000 cycles more than bare metal. Sending and receiving interrupts is costly, but less so than we expected.

Using the above interrupt implementation we then implemented sending and receiving of S-NET records using interrupts to notify the receiver of a message. For our tests we used records consisting of 8 bytes of metadata and 25 tags (ints) and their 25 corresponding values (ints). This is more than the average S-NET record should have, but still well within range of realistic sizes. We then measured the roundtrip latency of sending these records between two cores to see what sort of latency we could achieve. For comparison we also implemented a busy-waiting version of the interrupt code, an MPI version and a RCCE version. Table II shows the results of these measurements (averaged over 1 million roundtrips).

For S-NET, as mentioned earlier, we are not that interested in low latencies and more in lightweight communication. Our interrupt based messaging implementation outperforms the MPI version by a good 20%, but has double the latency of the RCCE version and almost quadruple that of its busy-waiting equivalent.

The last column in Table II shows the average number of cycles that the interrupt based implementation was idle each roundtrip. This free time can be used for computational work and makes up 75% of the roundtrip time. The RCCE and polling implementations on the other hand waste their time in a busy-wait loop, preventing the core from doing useful computation. This problem can of course be diminished by polling every n micro-/miliseconds instead of busy-waiting, at the cost of increased the latency.

Hops	Polling	RCCE	MPI	Interrupt	Free
0	5.4	9.9	25.7	18.8	14.4
1	5.6	9.9	25.8	21.0	16.2
2	6.6	10.4	26.2	21.8	16.9
3	5.9	10.1	26.2	21.2	16.2
4	6.1	11.0	26.6	21.6	16.6
5	6.3	10.4	26.7	21.6	16.4
6	6.5	10.6	27.3	21.6	16.4
7	6.9	10.7	27.0	22.2	16.8
8	7.4	10.8	26.8	22.4	17.0

Table II
ROUNDTRIP MESSAGING LATENCY (IN 1000 CYCLES)

It would be interesting to compare the efficiency of various polling intervals for the polling and RCCE implementations and see whether a trade-off in latency can improve their efficiency to be on par with that of the interrupt implementation. Unfortunately we did not have time to finish these measurements and include them in this paper. The biggest problem in measuring this information is the granularity of the Linux scheduler. Most ways for a thread to yield its execution (so computational code can run) cause the next poll to be postponed until the next time the scheduler runs it. This causes blocks to last far to long for the polling to be effective. To solve this problem we will need to utilise Linux' real time scheduling support to make suspending the polling threat frequently for intermediate amounts of time feasible.

VI. CONCLUSION

In this paper we have introduced the declarative coordination language S-NET. It is designed to provide a more convenient way of programming for multi-core/many-core chip architectures by viewing programs as independent components with an input and output stream. These components are then used to construct programs using a set of combinators.

We sketched out our ideas for implementing the S-NET runtime system on top of the SCC. Focussing on the two most important short-term goals for S-NET: utilising the LUT capabilities of the SCC to eliminate needless copying of data by being able to quickly remap it and using the message passing buffers and on-chip network to implement a lightweight communication mechanism for S-NET's records.

In the near future we also want to investigate the ability to dynamically change the power and speed at which the cores run, allocating more resources to bottlenecks in the S-NET network and reducing the energy wasted by the parts which are idle.

After this presentation of our ideas we went on to discuss the various tests we implemented, their performance and how their results impacts our ideas for S-NET. We determined that the latencies for sending interrupts are not as small as we would like, but are low enough what we need. Combined with the message passing buffers they provide us with enough to let us implement lightweight asynchronous message passing.

Unfortunately we were not able to finish implementing tests to compare the efficiency of asynchronous messaging to our polling and RCCE implementations (using various polling intervals), this means we don't have a conclusive "best" solution, but the numbers for our asynchronous implementation make us cautiously optimistic about how well it will stack up against the polling and RCCE implementations.

The biggest problem we encountered during our work was the lack of control over the L2 cache. Without control over the cache's behaviour it is difficult to accurately predict the performance (and in some cases correctness) of code. Making it difficult to accomplish some of the things we want to do, such as remapping memory using the LUTs. We can work around this, but at the cost of convenience and performance.

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," pp. 108–109, feb. 2010.

[2] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.

[3] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor (accepted for publication)," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011) – to appear, Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, (Istanbul, Turkey), July 2011. accepted for publication.

[4] Intel, "RCKMPI User Manual," February 2011.

[5] C. Grelck, A. S. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, S. Scholz, and A. Shafarenko, "S-Net Language Report 2.0," Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.

[6] C. Grelck, S. Scholz, and A. Shafarenko, "Asynchronous Stream Processing with S-Net," *International Journal of Parallel Programming*, vol. 38, no. 1, pp. 38–67, 2010.

[7] C. Grelck, J. Julku, and F. Penczek, "Distributed S-Net: High-Level Message Passing without the Hassle," in *1st ACM SIGPLAN Workshop on Advances in Message Passing (AMP'10), Toronto, Canada, 2010* (G. Bronevetsky, C. Ding, S.-B. Scholz, and M. Strout, eds.), ACM Press, New York City, New York, USA, 2010.

[8] C. Grelck, "The essence of synchronisation in asynchronous data flow," in *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, USA*, IEEE Computer Society Press, 2011.

[9] C. Grelck, S.-B. Scholz, and A. Shafarenko, "Coordinating Data Parallel SAC Programs with S-Net," in *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, USA*, IEEE Computer Society Press, 2007.

[10] F. Penczek, S. Herhut, C. Grelck, S.-B. Scholz, A. Shafarenko, R. Barrière, and E. Lenormand, "Parallel signal processing with S-Net," *Procedia Computer Science*, vol. 1, no. 1, pp. 2079 – 2088, 2010. ICCS 2010.

[11] F. Penczek, S. Herhut, S.-B. Scholz, A. Shafarenko, J. Yang, C.-Y. Chen, N. Bagherzadeh, and C. Grelck, "Message Driven Programming with S-Net: Methodology and Performance," *Parallel Processing Workshops, International Conference on*, vol. 0, pp. 405–412, 2010.

[12] S. Peter, T. Roscoe, and A. Baumann, "Barrelfish on the Intel Single-chip Cloud Computer," Tech. Rep. Barrelfish Technical Note 005, ETH Zurich, September 2010. http://www.barrelfish.org.

The Benefit of Topology-Awareness of MPI Applications on the SCC

Steffen Christgau, Bettina Schnor
Institute of Computer Science, University of Potsdam
August-Bebel-Strasse 89, 14482 Potsdam, Germany
Email: {hyperion, schnor}@cs.uni-potsdam.de

Simon Kiertscher
Potsdam Institute for Climate Impact Research
P.O. Box 60 12 03, 14412 Potsdam, Germany
Email: kiertscher@pik-potsdam.de

***Abstract*—In this paper the scaling of two MPI applications, a parallel answer set solver and a parallel climate simulation, on the SCC is presented and discussed. Further, the paper discusses issues of the implementation of the MPICH/CH3 device used on the Single-Chip Cloud Computer (SCC), the so-called RCKMPI. We present work in progress regarding an optimization of RCKMPI that improves the communication bandwidth by evaluating the application's communication graph and making optimal use of SCCs Message Passing Buffers. This way, we make RCKMPI *topology-aware*.**

I. Introduction

The SCC experimental processor is a 48-core "concept vehicle" created by Intel Labs as a platform for many-core software research [1]. All IA-32 cores are connected with an on-die network without providing cache coherency through hardware. Additionally, two cores share a SRAM which is called message passing buffer (MPB) and is generally used as fast communication memory. On each core an individual Linux instance can be booted, thus providing a cluster on a chip. Since a conventional Pentium-like core is used as its building block the SCC allows to run existing software with only recompilation to be done. RCKMPI, a Message Passing Interface (MPI) implementation for the SCC [2], makes the execution and investigation of existing parallel MPI applications possible.

We experimented with two different MPI applications that could be executed without modifications on the SCC. We selected two applications with different parallelization schemes. The climate simulation *Aeolus* uses a functional decomposition (see section III) where each function is run on a different core. *Aeolus* has shown a bad scaling on the Nehalem processor, but is suited for an architecture with no cache-coherence.

In section IV the parallel answer set solver *Claspar* and its scalability is presented. *Claspar* is the parallel version of *Clasp* suited for compute clusters. It uses a master-worker parallelization. Competition results show that clasp is a very competitive system, i.e, the potassco tools collection of which clasp is the most important part, was first in all categories of the ASP Competition in 2009[1].

Section V presents the concept of a *topology-aware* layout of SCCs Message Passing Buffer. First performance measurements proof the benefit of this approach.

[1]http://www.cs.kuleuven.be/~dtai/events/ASP-competition/Results.shtml

II. RCKMPI Basics

In early 2011 Intel provided RCKMPI an MPICH2 [3] derivative including dedicated CH3 devices for the SCC with the *SCCMPB* [2] among them. The *SCCMPB* device equally divides the MPB of each SCC core into $n-1$ sections where n is the size of the MPI process group. Each section of a core's MPB represents a dedicated area where a remote process writes in and the receiving core reads from. Due to the SCC core architecture the size of one section is always a multiple of a cache line size (32 bytes). A section also includes a flag segment which is 16 byte large. Thus, for an application started with seven processes the section size is 1344 bytes with a maximal MPI payload of 1328 bytes. This static layout of the MPB is initialized on MPI startup and kept over the whole runtime of the application.

III. Aeolus

Aeolus is a climate simulation that computes the evolution of seven atmospheric variables of synoptic scale [4]. It is developed at the Potsdam Institute for Climate Impact Research and is part of Climber-3, an Earth System Model of Intermediate Complexity (EMIC). Such systems attempt to overcome the gap between simple and comprehensive climate models [5]. The computation is dominated by seven variables which are arranged in a three dimensional grid around the Earth's globe and occupy 180 KB each. The evolution is calculated in discrete time steps. Between those variables a producer consumer relationship exists (see Fig. 1): The computed value of one variable in a time step is required by the computation of several other variables in the next step. Additionally, the computed values are regularly stored.

Initially designed as a sequential application, *Aeolus* was later parallelized in cooperation with the Institute of Computer Science at Potsdam University with a *functional* decomposition approach: Each process computes one synoptic scale variable whose computational effort is approximately equal. The exchange of required variables is done via message passing using MPI. The message size equals the size of a variable, i.e. 180 KB. A master process collects the produced variable values and puts them to the storage. Due to the exchange of variable values after each single time step *Aeolus* is a communication intensive application.

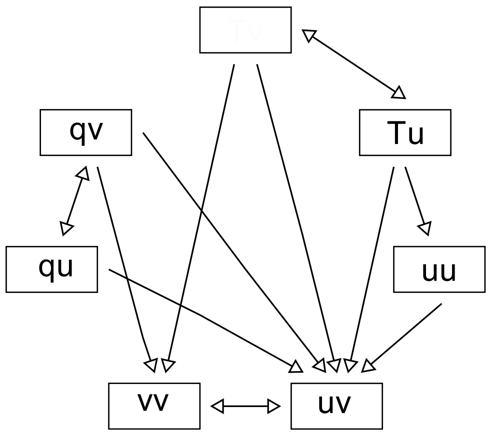


Fig. 1. Dependency graph of the variables computed by Aeolus.

A. *Experiences on a Nehalem cluster*

The parallelized application was executed on a Linux cluster where each of the 28 nodes is equipped with two Intel Xeon E5520 quad-core CPUs with Nehalem cores running at 2.27 GHz (Simultaneous Multi-Threading disabled) and 48 GB of RAM. All nodes are connected via 20 GBit/s Infiniband over a single switch and use Intel MPI as message passing library. The configuration of a single node with eight cores in total allows the whole parallel application to run on one node. Thus the computational power of all cores can be exploited. Since only seven variables are computed the I/O performing master processes can be bound to the eighth core.

Using such a configuration leads to a speedup of 2.47 which is really poor compared with the optimal speedup of 7. This may be related to high cache saturation and pollution effects as one of the variables exceeds the size of the L1 cache (32 KB). Moreover, in the case of *Aeolus* the cache coherency between the cores' caches is not required from the application side.

Therefore, each MPI process was mapped to a single node of the cluster eliminating cache effects, i.e. *Aeolus* was started on 8 nodes using only one core per node. This results in an improved speedup of 3.15. The still not optimal speedup is due to high communication costs, since the updated variables have to be exchanged via the interconnect after each time-step. Performance evaluation revealed that the application spends 10 to 20 percent of its runtime in message passing calls when running on the Nehalem cluster.

B. *Experiences on SCC*

As the SCC is not providing any cache coherency between the cores and offers a fast on-chip network running *Aeolus* on the chip was a clear consequence. Without further platform-specific optimizations the observed speedup was 4.04 while maintaining computational correct results.

IV. Claspar

Claspar [6] is a distributed Answer Set Programming (ASP) solver, based on the ASP solver *clasp* [7]. Thinking of new assistive technologies where people are assisted by an intelligent system which evaluates information like for example sensor data the need for high performance is obvious. The used solvers have to be fast to respond in good time to be helpful.

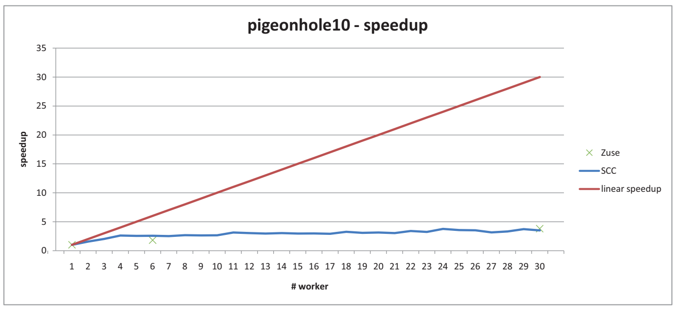


Fig. 2. Speedup comparison of *Claspar* on SCC and on a Nehalem Cluster for pigeonhole10.

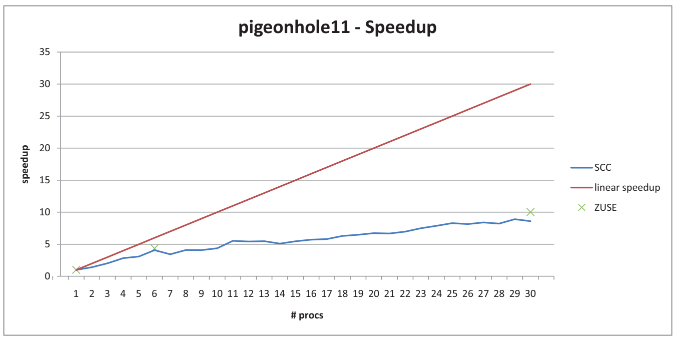


Fig. 3. Speedup comparison of *Claspar* on SCC and on a Nehalem Cluster for pigeonhole11.

Claspar uses a master-worker approach. It uses MPI for the communication between the master and its worker processes. Basically the master takes care about dividing the search space among the workers. If a worker finishes its part, it asks the master for new work. The master sends a split request to a worker which is not finished yet asking to split its search space.

The runtime values for claspar on the Nehalem cluster are taken from [8]. The considered benchmark is the pigeon hole benchmark where the search space has to be traversed completely (since there is no way to put $N+1$ pigeons into N holes allowing only one pigeon per hole).

Figure 2 and Figure 3 show the scaling of claspar for 1 up to 30 workers for pigeonhole10 and pigeonhole11. While the runtime improvements are impressive the scaling is poor on both platforms. In case of pigeonhole10 the runtime for one worker is 263 seconds, for two workers 168 seconds, and for four workers 100 seconds. In case of pigeonhole11 the runtime for one worker is 3377 seconds, with two workers 2388 seconds, and 1189 seconds with four workers.

To investigate the bad scaling of claspar, we made another run with pigeonhole12 on the Zuse cluster and used the scalasca tool for performance analysis. The sequential runtime

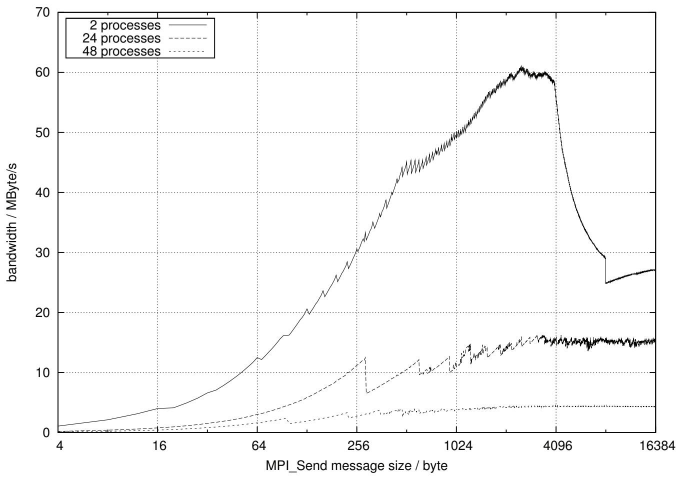


Fig. 4. Ping-pong bandwidth between SCC core 0 and 47 for different numbers of started MPI processes.

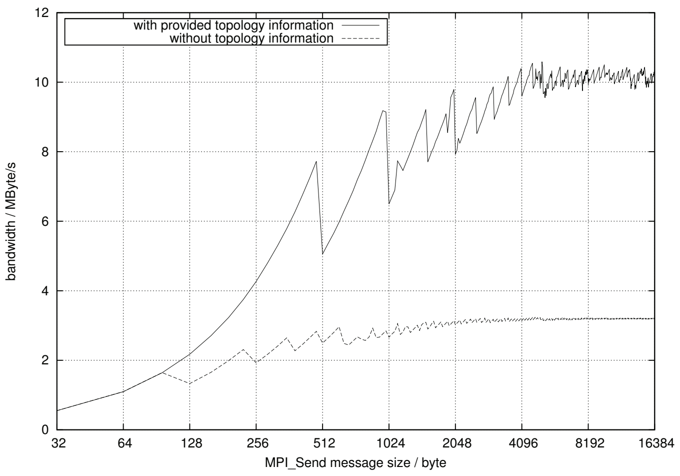


Fig. 5. Ping-pong bandwidth between two of 48 MPI processes with and without topology information

was about 2747 s and nearly completely spent with solving. In case of the parallel run using 62 workers, each worker was uniformly loaded and spent about 116 s with solving, i.e. the parallel version needed $116 \cdot 62 = 7912$ s solving time. So, the reason for the bad scaling is not overhead due to communication but that the solving time increases. The sequential solver has obviously learned clauses which are more efficient in the solving process which the parallel version does not. This is a hint for the claspar team that the parallel version may benefit from the exchange of clauses.

V. Making RCKMPI Topology-Aware

The partitioning of the MPB has influence on the bandwidth between two cores. Fig. 4 shows the bandwidth of a MPI ping-pong between core 0 and 47 for different message sizes and different number of processes in *comm_world* (2, 24 or 48 started processes). So only 2 processes exchanged messages, the remaining 22 resp. 46 in the second and third experiment had no really work to do.

Fig. 4 shows that the benefit of using the MPB is obviously: As long as the message size fits into the MPB section at the receiver side, the bandwidth is increasing. Each time the message size exceeds a multiple of the section size the bandwidth drops due to the necessary additional packet transfer of the remaining data. Additionally, increasing cache conflicts arise in case of larger messages. Therefore, the bandwidth decreases as smoothly as with RCCE, the communication framework shipped with the SCC [9].

But what also can be seen is that the effect of the limited MPB section size manifests with a decreasing maximum bandwidth with increasing number of processes in use. A comparable impact can not be observed with a MPICH2 CH3 device for Infiniband or even TCP/IP. The reason is that while in the test scenario the ping-pong communication took place only between two processes, RCKMPI cuts the MPB in smaller sections when more processes are started.

Since the minimum MPB section size is 160 byte, one would expect that the bandwidths would be equal up to a message size of 160 byte. In Fig. 4 one can observe different bandwidths already for small message sizes.

A. RCKMPI Improvement

To compensate the equally limited bandwidth one can evaluate an application's intrinsic communication topology. As for *Aeolus*, it is not required to communicate with all cores. For example the process computing the variable *TV* does not interact with the producer of *QV* (see Fig. 1). In case of *claspar* the communication graph is a star.

Hence, parallel applications can gain additional bandwidth and performance gain if the communication graph of the application is considered, i.e., if the bandwidths for message transfers are adjusted for processes that communicate frequently with each other. MPI allows an application to introduce topology information in two fashions. The first one, via `MPI_GRAPH_CREATE` e.g., is a general graph based approach where nodes represent MPI processes and edges stand for communication between those processes. The second approach are Cartesian topologies, which are defined via `MPI_CART_CREATE` for convenience reasons to facilitate the definition of grids, tori or even hypercubes. With each approach a new communicator is allocated by MPI [10].

In MPICH2, which is the base for RCKMPI, the creation of a communicator can be hooked by a CH3 device. By doing so, it is also possible to hook into the introduction of topology information. Thus, these information can be used by the SCCMPB CH3 device. This process is completely transparent to the application programmer. The required steps to make the CH3 device topology-aware consist of simple MPI calls. In case of a Cartesian topology one can distribute all processes among all dimensions with the help of `MPI_Dims_create`. Afterwards a new topology communicator is derived from an existing communicator with the process distribution created before (`MPI_Cart_create`) as shown in Listing 1. When

```
#define NUM_DIMS 2

int grid_dims[NUM_DIMS], grid_periods[NUM_DIMS];
MPI_Comm comm_topo;

/* set all items of grid_periods to true
   to get a torus */

MPI_Dims_create(numProcs, NUM_DIMS, grid_dims);
MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, grid_dims,
     grid_periods, true, &comm_topo);
```

Listing 1. Example MPI Code to introduce a Cartesian topology

such a communicator is generated, the CH3 device enlarges the MPB sections of communication partners.

Thereby, two problems need to be solved. On the one hand, an improved MPB layout must consider both communication neighbors and all other MPI processes since collective operations like barriers have to be supported. On the other hand, each MPI process has to know its offset within all remote MPBs. Since the MPB sections shall be enlarged for communication neighbors, a process needs to know the neighbors of the owner of the remote MPB to calculate its remote write offset.

The MPB layout of the original RCKMPI implementation was changed in the following way: The flag areas were moved from the end of a write section to the beginning of the MPB. This had to be done to prevent the flags from being destroyed during the MPB layout rearrangement when topology information are introduced. The remainder part of the MPB is generally reserved for data exchange. In case of Cartesian topology this space is divided into regions for neighbor and non-neighbor processes. For each of the latter, 96 bytes are resevered, e.g. for collective operations. For the neighbor proceses, the remaining MPB area is equally subdivided for each dimension of the Cartesian topology. Within such a dimension area, half of the area is devoted to the upper resp. lower neighbor process.

The original and the new MPB layout for an application which consists of nine processes using a 2-D domain composition is shown in Fig. 6. Since domain decomposition is a popular approach for parallelization, this represents a big class of parallel applications. The corresponding communication topology is a 2-D Cartesian topology where each process has four communication neighbors. After the flag area follow the 96 bytes areas for all $n-1$ processes (expect the MPB owner). The remaining MPB area is given to the four communication partners.

B. First Results

The implementation of the proposed improvement is developed within a master thesis at Potsdam University [11]. In Fig. 5 we show first results and compare the bandwidth between two processes with and without topology-aware MPB layout. In both cases all 48 SCC cores were used. Again, we simulated a typical application with a 2-D Cartesian communication topology. The topology information is introduced to

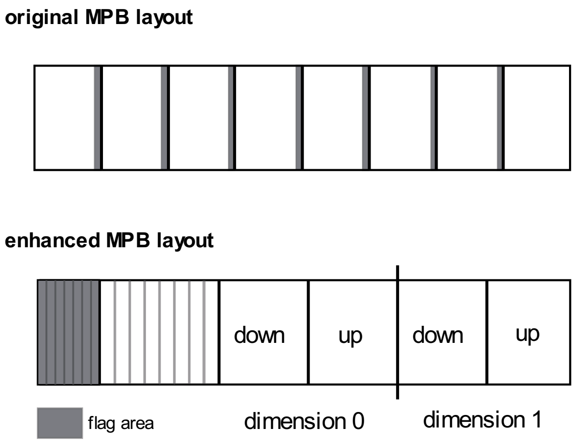


Fig. 6. MPB layout of the original RCKMPI SCCMPB CH3 device (top) and the enhanced version (bottom) for 9 processes

RCKMPI via `MPI_GRAPH_CREATE`. So the lower curve in Fig. 5 is the same as the 48-processes curve in Fig. 4. As a result of our optimization the bandwidth is enhanced by a factor of three in this case. From this improvement a compute-intense application can benefit additional performance. In contrast to Fig. 4, one can observe that the bandwidths are the same for small message sizes.

VI. Conclusion and Current Work

Our experience with 2 MPI applications from different application areas on the SCC was very positive. Both applications were runnable out-of-the box. Both applications are communication intensive with known scaling problems on current multicore architectures resp. current cluster platforms. So, it was interesting to investigate their behavior on the SCC platform.

While the application Aeolus benefits from the SCC architecture, especially from the non cache-coherence approach, the parallel answer set solver claspar seems to have also scaling problems on the SCC. But the reason is application-inherent and not related to the communication demands of claspar. Currently, we are investigating the efficient exchange of learned clauses [12], so-called *nogoods*. This seems to be a very big challenge on traditional compute clusters since the communication demands are enormous: every solver-process learns thousands of clauses each second which must be communicated to all other solvers.

In the field of cluster computing, the MPI feature to specify topology information was unnecessary as long as the cluster belongs to the uniform communication architecture (UCA). In case of the SCC, we have special hardware, the MPBs, which makes it NUCA. We use the communication topology of an application to optimize the MPB layout and in this way the bandwidth for frequently communicating processes. More implementation details and measurements will be presented in [11].

References

[1] J. Howard *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 7-11 2010, pp. 108 –109.

[2] I. A. C. Urena, "Lightweight MPI for the Single Chip Cloud," in *Many Core Architecure Research Community Symposium*. Braunschweig: Intel, Nov. 2010. [Online]. Available: http://communities.intel.com/servlet/JiveServlet/previewBody/5844-102-1-8986/MARC-Symposium-Nov-2010-Lightweight-MPI.pdf

[3] W. Gropp, "MPICH2: A new start for MPI implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, D. Kranzlmueller, J. Volkert, P. Kacsuk, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2002, vol. 2474, pp. 37–42.

[4] D. Coumou, V. Petoukhov, and A. Eliseev, "Three-dimensional parameterizations of the synoptic scale kinetic energy and momentum flux in the earths atmosphere," *Tellus*, 2010, in review.

[5] M. Montoya, A. Griesel, A. Levermann, J. Mignot, M. Hofmann, A. Ganopolski, and S. Rahmstorf, "The earth system model of intermediate complexity CLIMBER-3α. Part I: description and performance for present-day conditions," *Climate Dynamics*, vol. 25, pp. 237–263, 2005.

[6] L. Schneidenbach, B. Schnor, M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Experiences Running a Parallel Answer Set Solver on Blue Gene," in *16th European PVM/MPI Users' Group Meeting*. Espoo, Finland: Spinger, Lecture Notes in Computer Science, 2009.

[7] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "Conflict-driven answer set solving," in *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, M. Veloso, Ed. AAAI Press/The MIT Press, 2007, pp. 386–392, available at http://www.ijcai.org/papers07/contents.php.

[8] [Online]. Available: http://www.cs.uni-potsdam.de/claspar/

[9] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: the programmer's view," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.53

[10] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), September 2009.

[11] S. Christgau, "Performance Optimization of Message Passing on the SCC," Master Thesis, University of Potsdam, to appear 2011.

[12] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, and B. Schnor, "Cluster-based ASP Solving with claspar," in *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11) (accepted paper)*, Vancouver, Canada, 2011.

On Efficient Message Passing on the Intel SCC

Randolf Rotta
Brandenburgische Technische Universität Cottbus
Konrad Wachsmann Allee 1
03046 Cottbus, Germany
Email: rrotta@informatik.tu-cottbus.de

***Abstract*—The Single-Chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. Instead of the usual shared memory programming, its design favors message passing over a special shared on-chip memory. However, the design of efficient message passing is still an ongoing research work, because the system differs quite much from traditional hardware. This paper presents design options for message passing protocols on the SCC and discusses some implications.**

I. Introduction

The Single-Chip Cloud Computer (SCC) experimental processor [1] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. Although the SCC features shared on-chip memory, it is not intended for traditional shared memory programming. Instead, the platform is designed for research around many-core message passing concepts. In consequence, the on-chip memory, also called *message passing buffer*, is quite small and deliberately does not provide automatic cache coherence.

While the SCC provides fast access to the on-chip memory, middleware libraries have to provide actual message passing *protocols*, i.e. algorithms organizing the concurrent access to the message memory. The SCC's non-coherent memory and its high number of cores is quite different from previously known processors, and thus, porting existing message passing code from cc-NUMA systems to the SCC does not result in optimal performance (see for example [2]). Therefore, the design of efficient message passing poses a renewed design challenge: Which old or new strategies are feasible on the SCC? What is achievable within the limits of the current hardware? How much could the performance benefit from future adapted hardware support?

This paper discusses some aspects of the design space of message passing protocols on the SCC. The focus will be on non-blocking asynchronous in-order transfer of small messages (<200 bytes) between arbitrary cores, that is without explicitly established point-to-point connections. These requirements are based on active message middle-ware layers like TACO [3], but are useful for other software as well, for example MPI on top of active messages [4].

The next section discusses relevant parts of the SCC hardware, and Section III introduces some useful performance indicators. The main part is Section IV with an overview of the design dimensions and options for message passing protocols. The paper concludes with a discussion of related work and possible directions for future work.

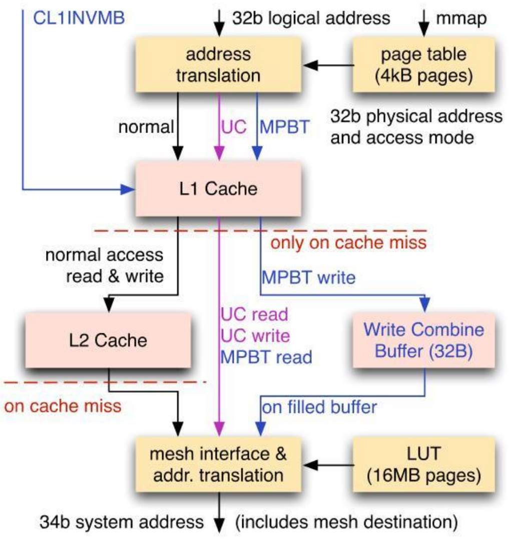


Fig. 1. Conceptual address translation and access modes on the SCC.

II. The Intel SCC

This section start with an overview of the communication capabilities of the SCC. After some notes about the performance, the section concludes with a discussion of differences to other systems.

A. Memory Access over the Mesh Network

The SCC combines 48 standard processor cores, derived from the P54C Pentium, on a single chip. All communication is carried out over a packet-switched 2D mesh network of 6×4 routers. Communication between cores is performed indirectly by writing to and reading from shared on-chip SRAM and external DRAM memory using the standard machine instructions, i.e. `MOV` with byte, word (2 byte), or double word (4 byte) granularity [5]. The destination and access mode of a request is determined in two steps as shown in Figure 1: First, the usual page table maps from the logical to the core's physical address space and sets the access mode on 4kB page granularity. Between core and router, the physical addresses are mapped to system addresses through a lookup table (LUT) with 16MB granularity. These addresses contain the mesh coordinates of the destination router and the local destination (e.g. SRAM, device registers, or attached external devices).

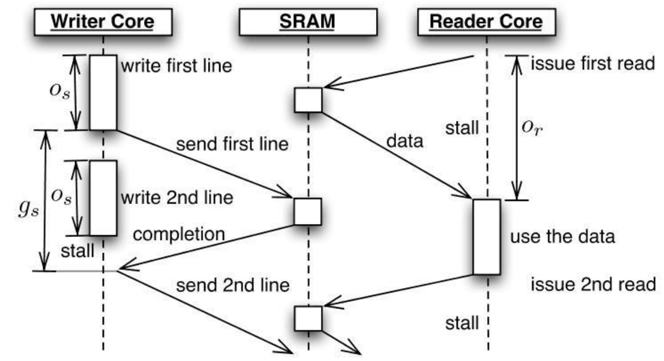


Fig. 2. Writing and reading over the mesh network.

Although all cores can access the same DRAM and SRAM memory, the system has no implicit cache coherence mechanism. Instead, a new access mode called *Message Passing Buffer Type* (MPBT) is provided. Data read from such memory is only cached in the L1 cache and the new instruction `CL1INVMB` invalidates all such MPBT lines from the cache. Write operations to MPBT memory are collected in a write-combine buffer, which tries to fill up an entire cache line before sending the data to the destination.

The usual *un-cached* memory access mode (UC) can be used as well. Data read from such memory is not cached and write operations are directly issued to the network. Concurrent writes to the same memory line do not conflict. Note that it is possible to mix MPBT and UC access to the same physical memory by mapping it twice into the logical address space. In this setting, it is just necessary to invalidate MPBT lines from the cache before accessing the UC-mapped memory.

In addition, one atomic test-and-set bit per core is available in the device registers. Reading from it activates the bit and returns its previous state. Writing resets the bit to zero. The firmware of the system interface provides a set of atomic counters. Reading from a counter increments it and returns its previous value. Unfortunately, the access is expected to be much slower than to the on-chip SRAM.

B. *Performance Model for Memory Access*

The LogP-model of Culler et.al. [6] summarizes the behavior of communication systems in a few parameters, namely the latency L, send overhead o_s, receive overhead o_r, gap g between subsequent messages, and the number of processors P (=cores). While it was not designed to model memory accesses, SCC's behavior can be represented quite well using the overhead and gap as depicted in Figure 2.

The P54C cores are strictly in-order and can handle only a single outstanding memory request. Write operations over the mesh are completed by a small response message from the destination. This results in write ordering when accessing different destinations, but also simplifies costs predictions. It takes time o_s to fill the write-combine buffer or issue an un-cached write operation. These overlap with the previous request, but the core stalls until the previous request is completed. The time between issuing the write and its completion is modelled by the gap g. When reading, the core stalls until the data arrived. This time is represented by the receive overhead o_r.

TABLE I
MEASURED CYCLE COUNTS FOR WRITING AND READING OVER THE MESH. RANGES REFER TO THE SMALLEST AND LARGEST DISTANCE.

	1 byte (UC)	4 byte (UC)	32 byte (MPBT)
o_s	5	5	28
g_s	48–81	48–81	75–105
o_r	53–86	53–86	58–88

All three parameters depend on the memory type (MPBT or UC) and the mesh distance. Micro-benchmarks like [7] can be used to estimate the actual parameters. Table I provides cycle counts for the chip configuration with 800 MHz core clock rate and 1600 MHz mesh clock rate.

C. *Differences to other systems*

An ultra low latency network: With networks like Infiniband the hardware latency (especially between CPU and network controller) dominates most other costs. In combination with high processor speeds, differences between message passing implementations are quite small. In addition, the network already implements many details, for example, hardware-managed message queues. In contrast, on the SCC, the communication latency is quite low, but several communication steps are necessary for a single message transfer. Thus, the performance differences between protocols are much larger.

A different memory model: Most message queue designs for shared memory are based on cache coherent memory and compare-and-swap operations (see for example [8], [9]). The SCC provides no hardware cache coherence. Thus, the location of data in the mesh is known exactly and all data transfers are triggered explicitly. This also eliminates false-sharing problems. Unfortunately, no remote compare-and-swap is available on the SCC, but can be emulated with the lock registers. In contrast to traditional cache-coherent systems, it is much more efficient to use un-cached writes to update individual values directly in the memory instead of performing cache line round-trips. Combining un-cached and MPBT access to the memory provides new design opportunities.

Finally, the scalability of the overal message passing becomes an issue: The increasing number of cores results in an even faster growing number of communication partners and managing these connections can quickly become a bottleneck in respect to computational overhead and memory usage.

III. COMMUNICATION PATTERNS

The LogP model was designed to describe essential performance characteristics of communication systems. Based on the mesh parameters (Table I), predictions of the mesh-induced protocol overhead O_s, O_r, gap G and latency L are easily calculated.[1] However, the raw LogP parameters are difficult to

[1]Here, uppercase letters denote parameters of a protocol while lowercase letters are used for the mesh. The receive overhead O_r is the time to process a transmitted message.

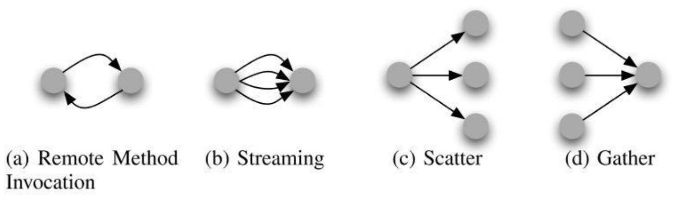


Fig. 3. Communication patterns.

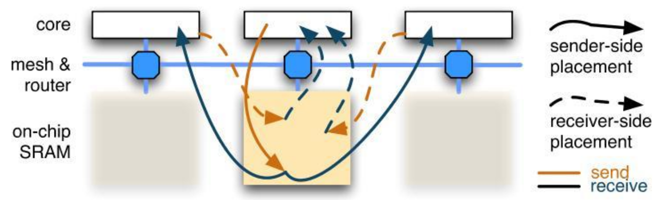


Fig. 4. Message placement on receiver vs. sender side.

interpret. This section introduces four communication patterns that are used to investigate specific aspects and implications of a protocol's performance (Figure 3).

The first pattern, called *Remote Method Invocation* (RMI), describes a function call with result value, which is executed on a remote core. It is similar to message roundtrips and its completion time is roughly $T_{\text{RMI}} = 2(O_s + L + O_r)$.

A second scenario are distributed processing pipelines, where the cores send and receive continuously *streams* of data or asynchronous remote method calls. For small messages like method calls, the steady-state *message throughput* is of interest, which should be about $\text{TP} = 1/(O_s + G)$ messages per cycle. For large data transfers that are split into smaller messages, the *bandwidth* ("goodput") is more interesting. It should be about $\text{BW}_n = n/(O_{s,n} + G_n)$ bytes per cycle when messages of n bytes payload are used.

Some applications use collective operations in which a task is initiated by a core and then performed in parallel on a group of cores. This involves propagation of the task (multicasting) and possibly waiting for its completion (collecting and merging results). Unfortunately, the completion time depends on the multicast topology and the optimal topology depends on the protocol parameters. Instead, the collective operations are dissected into their basic local communication patterns: The *Scatter* pattern delivers a message to k direct receivers. Its completion time at the sender is roughly $k(O_s + G)$ and the arrival time at the last receiver is $k(O_s+G)-G+L+O_r$. The inverse direction is the *Gather* pattern. Its completion time is kO_r (processing k transmitted messages).

Obviously, all of these performance indicators depend on the message size and the mesh distance. More importantly, they also depend on details of the protocols and thus the above formulas provide just a rough impression.

IV. A Design Space for Message Passing

This section presents a design space for message passing protocols and discusses several available options in each design dimension. The discussion begins with the software level at which message passing could be implemented (Section IV-A). The next sections discuss *message placement* (Section IV-B) and *memory allocation* (Section IV-C).

Some kind of flow control is necessary to protect against overwriting of unprocessed messages, which could happen by two cores writing concurrently or by one core writing too early to the same place. This protection is achieved by notification and acknowledgement mechanisms: *Notification mechanisms* signal the arrival of new messages and have to ensure that no message is missed (Section IV-D). *Acknowledgement mechanisms* signal which messages have been processed and thus enable the save reuse of message memory (Section IV-E). When all message slots are in use or no notifications are currently possible, a sender cannot send further messages without conflicts. This is resolved by *wait mechanisms* (Section IV-F).

A. Levels of Abstraction

At which software layer should the message passing be implemented? Independent *one-to-one* queues are easier to adapt to specific communication needs as they can be mixed in the same application. *Many-to-one* protocols exploit that they will receive messages from multiple sources and *one-to-many* protocols optimize sending to different destinations. Combining both into *many-to-many* protocols provides most chances for exploiting synergies. While such approaches complicate incorporating application specific knowledge, their performance might make this unnecessary.

For example, with independent one-to-one queues, the memory requirements grow quadratically with the number of cores (at least $P^2 - P$ individual queues for P cores) and polling slows down linearly ($P - 1$ queues to check). Checking for messages on the SCC while using only MPBT memory takes at least 3200 cycles (47 cores times 58+9 cycles for fetching a line and testing the content). Combining the notification mechanism of all queues into a many-to-one protocol, allows to reduce this overhead to 220 cycles and possibly even lower as will be discussed in Section IV-D.

In summary, efficient protocols will most probably require global approaches. These cannot be implemented at the level of individual message queues inside applications, but should be provided as a shared service by the system.

B. Placement: Pulling vs. Pushing Messages

With *receiver side placement* (aka push mode) the sender writes the message payload into the remote receiver's memory. In contrast, with *sender side placement* (aka pull mode) the message is put in the sender's local memory.

For the method call and streaming patterns there is no direct difference: On the SCC, a remote write with a local read takes as long as a local write with a remote read. The effort just shifts between sender and receiver. However, for the scatter pattern, the sender side placement is better (Figure 4): Local writing reduces the send overhead and gap (sequential local writing), while the more expensive remote transfer to the receivers is parallelized. In contrast for the gather pattern, the receiver side placement is better: Sequential reading is faster on local memory, while the senders would do their remote writing in parallel. What a doubtful choice: When a scatter is followed by a gather, both effects cancel out each other. Therefore, a protocol can exploit the parallelism of appropriate message placement only by breaking the symmetry.

C. Allocation: Managing the Message Memory

This design dimension is concerned with the allocation of message memory. Here, we consider a fixed amount of fixed size *message slots* per core, which reside in the on-chip SRAM. The main focus of this subsection lies in quick memory allocation. A separate acknowledgement mechanism will be necessary to reuse the slots.

1) Static Allocation: Each core uses a specific slot depending just on the destination. In the simplest case just a *single slot* is used for all destinations. In consequence, a new message can be sent only after the acknowledgement of the previous message and thus the send gap G is at least $L+O_r$. This effect is decreased by using a *separate slot per destination*. Then, subsequent messages to the same destination still have the longer gap $G_{\text{same}} = L+O_r$, but sending to varying destinations has a smaller gap $G_{\text{other}} \ll G_{\text{same}}$.

The single slot approach requires just P slots (one for each core), and P^2-P slots are required when using separate slots. Separate slots improve the performance of the gather pattern, while streaming remains inefficient because of the larger gap.

2) Static Allocation by Direction: Each core owns a sender side and a receiver side set of separate slots per destination. The sender side slots are used for scatter communication and the receiver side slots are used to send gather messages. Other messages can use any of the two possible slots. This increases the scatter and gather performance and reduces the send gap problem slightly, but requires even more slots (in total $2P^2$).

3) Receiver-based Allocation: The sender acquires a slot from the receiver, for example, by exploiting the notification mechanism. This works best with receiver side placement.

4) Sender-based Round-Robin: Each sender has a set of slots in its local memory (sender side placement) and the slots are used in a round-robin fashion: Before using a slot, the sender checks for the acknowledgement of the slot's previous message. If not yet free, the next slot is tried. Under normal conditions it should take some time until a slot is revisited and thus the first slot tried is free with a high probability.

With this approach, the sender has to wait just when all slots were checked without success. Thus, especially the streaming throughput and bandwidth is improved in comparison to static allocation. A further advantage is the gained control over the size of the message memory, because it no longer depends on the number of cores. Instead, the number of slots per core can be chosen freely and more slots decrease waiting times.

This idea can be extended to variably sized slots. While this increases the memory utilization even further (less bandwidth degradation for small message sizes), the additional management overhead might increase the send overhead too much.

D. Notification: Discovering new Messages

The notification mechanism's task is to signal the arrival of messages and discovering these at the receiver.

1) Separate Notification Flags: Each core d owns an array of notification flags $N_d(i)$. Using SCC's un-cached write operations, a sender s can set his flag $N_d(s)$ at receiver d without interfering with other cores. Let non-zero values indicate arrived messages. The receiver d polls for messages by scanning N_d, which is speed up by fetching whole lines (i.e. at most 32 flags, each one byte) at once by reading from the MPBT-mapped memory. Therefore, the mesh-induced polling overhead is $\lceil P/32 \rceil o^{\text{local}}_{r,32}$, i.e. 116 cycles. This approach is efficient on the SCC, because the senders do not suffer from false-sharing as would be the case on cache-coherent systems.

The search for notifications is accelerated by looking for nonzero double words first, which reduces the tests from 48 to 12 in case no message arrived. Our benchmarks showed about 220 cycles polling overhead with this method. The idea could be extended to notification trees, reducing the necessary tests to $\mathcal{O}(\log P)$. However, this increases the memory usage and computational overhead considerably.

2) Single Flag with Locking: SCC's 48 atomic locks enable the following strategy [10]: The sender acquires the destination's lock and then writes its notification into a single fixed flag. The receiver polls by reading just this single flag. When a notification was found, the receiver resets the flag and releases the lock. This would yield nearly optimal RMI and scatter performance, but the streaming and the gather performance would be quite bad. Without contention, the notification overhead for the sender will be about 106–172 cycles (lock, write flag). Polling needs just about 53 cycles to check the flag. The acknowledgement overhead will include 106 cycles (reset flag, unlock).

3) Notification Ring Buffer: The read position is incremented only by the owner, while the write position is incremented by all senders and thus ideally is an atomic counter. The buffer is full, when the difference between the acquired write position and the current (or last known) read position is larger or equal the buffer size. Then, a sender must not write to its flag, but new senders shall still be able to reserve a flag by incrementing the write position. This can be achieved with full 32 bit counters and computing the actual flag position just for writing. Note that the receiver cannot compare read and write positions to find new messages, because the sender writes the actual notification after incrementing the counter.

Compared to the previous notification mechanisms, this one is scaleable: It supports multiple concurrent senders while the memory requirements and polling overhead are independent of the number of cores. Emulating the counter with locks results in 265–430 cycles notification overhead (lock, fetch write position, write new position, unlock, write flag). Using the hardware atomic counters should be faster.

4) Message Linking: The previous approaches can be combined with this method in order to increase the throughput and reduce the congestion as follows: Instead of acquiring a new notification flag, each message contains an additional notification flag. The sender writes the location of the new message into the flag of its previous message sent to the same destination, thus forming a linked list of messages. The receiver discovers the first message using the primary notification mechanism. Then it processes each message in the list until reaching the last message and continues with the primary notification mechanism. Special care is necessary with the acknowledgements, because the receiver may see the end of the list while the sender appends a new message.

E. Acknowledgement: Freeing Memory

Once a message is processed by the receiver, it is necessary to give the memory back to its owner for use in future messages. This information can be placed in a separate array of *acknowledgement flags* at the receiver or sender side with one byte per message slot. Alternatively, the structure of message headers can be exploited. For example, active messages contain a non-zero pointer to a handler function and resetting it to zero provides an *in-message acknowledgement*.

The owner of a message may *actively check* the flags during polling. In that case it is more efficient to check just the outstanding acknowledgements. However, because the chance of an acknowledgement increases with time, it is even more efficient to just *check on-demand* as late as possible.

A separate array of byte-sized acknowledgement flags is an interesting option in combination with a dynamic slot allocation, because the fast search method for notification flags (Section IV-D1) can be applied to quickly collect acknowledged slots. For this purpose, acknowledgements are represented by non-zero flags. The collector resets these flags to zero and adds the slots to a free-list. This collection can be performed during idle time and when the free-list is empty.

F. Waiting: Handling full Queues

A sender has to wait when all message slots or notification flags are currently in use. Simple busy waiting and any other blocking behavior is not sufficient as it would lead to deadlocks when two cores try to send a message to each other.

Depending on the middleware framework, several options are available. The sender might just do *repeated polling* until the message can be sent. This has the lowest overhead, in case the channel becomes free again very soon. A second option is to *temporarily suspend* the thread (or coroutine) that issued the message and thus other work can proceed. Once the scheduler activates the thread, it will retry sending the message. However, if all threads are waiting, polling is necessary. Alternatively, the thread can be *completely suspended*. The protocol has to wake up the thread when the channel becomes free, which could be detected from incoming acknowledgements. Instead of exploiting threads, *sender side message queues* can store the pending messages. During polling, the channels could be checked and the next message sent. When this queue is full as well, the protocol has to revert to one of the above strategies.

V. Related Work

The design options presented in the previous section can be used to characterize existing protocols and get a quick impression of their relative performance. In this section, the current state regarding existing SCC software is discussed, namely X10, Barrelfish, RCCE, Rckmpb, RCKMPI, and TACO.

X10 is a parallel object-oriented programming language targeted to multi-core systems [11]. The X10 implementation for the SCC [12] uses sender side message placement, separate notification flags at the receiver side and separate acknowledgement flags at the sender side. The flags are probably accessed through the MPBT mode and are stored in separate cache lines. Thus, polling for new messages will cost about 3200 cycles (see Section IV-A), i.e. the RMI roundtrip time is at least 6400 cycles.

The *Barrelfish* operating system has a SCC variant and first benchmarks reported an average message round-trip time of 8746 cycles [2], including the operating system's scheduling overhead. The message data is placed in shared off-chip DRAM and the on-chip SRAM is used just for notification. The notification mechanism uses the inter processor interrupt (IPI) and a MPBT-based ring buffer that is protected against concurrent writing by the receiver's lock register. Therefore the polling overhead will be low (58 cycles to inspect the cache line at the current read position), but the notification overhead is at least 660 cycles (lock, fetch r/w positions, write notification, update write position, unlock, send IPI).

RCCE is a message passing library for the SCC [13]. The low level interface provides put and get operations to the on-chip SRAM and synchronization flags. On top of this, blocking send/receive operations with sender-side message placement are implemented. The synchronization flags are used for notification and acknowledgement, but any-source polling is currently not supported. Applications have to probe for each possible source sequentially and incoming messages can be processed quickly just as long as the sender is known.

The *Rckmpb* driver provides TCP/IP networking between Linux instances running on the SCC cores [13]. It uses sender side placement with round-robin allocation and receiver side separate descriptor flags (each one byte), which are used for notification and acknowledgement. Polling just reads the two cache lines of the descriptor flags (see Section IV-A). Individual flags are updated without locking by exploiting the behavior of the Write Combine buffer (similar to a UC write).

RCKMPI is a MPI implementation for the SCC that adapts the CH3 streaming channel of MPICH [14]. It uses receiver side placement with static allocation in separate slots. Flow control is managed with sequence counters, which are stored in separate cache lines. As with all protocols that use separate cache lines for notification, polling is particularly slow.

We implemented an over-simplified protocol (Figure 5) for the TACO framework [15]. It uses receiver side placement with static allocation, separate notification flags, separate on-demand acknowledgement flags, and waiting by polling. The implementation has a send overhead of $O_s = 250$ cycles, a latency of $L = 600$ cycles, a send gap of $G_{\text{same}} = 880$ cycles, and a roundtrip time of $T_{\text{RMI}} = 1770$ cycles (all values include the middleware overhead). Although this protocol seems to perform very good, it is still not optimal. For example, the

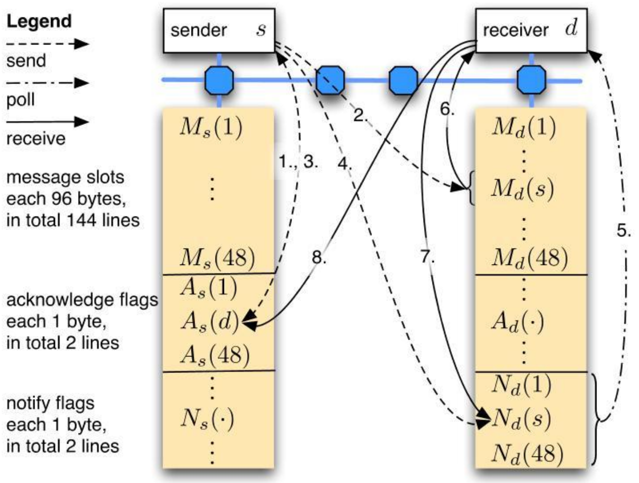


Fig. 5. Exchange of an one-way message over the SRAM.

used allocation and notification mechanisms are not scalable, and the acknowledgement flags could be replaced by checking the notification flags or in-message flags.

VI. Summary and Future Work

Various aspects of message passing on the SCC can be organized in a design space with six dimensions: Abstraction Level, Placement, Allocation, Notification, Acknowledgement, and Waiting. Flow control is a cross cutting concern connecting Allocation, Notification and Acknowledgement. The Level of Abstraction is not an independent dimension, but classifies protocols into a range from independent one-to-one channels to global unified many-to-many protocols.

Future work and collaboration is necessary in the exploration of actual protocols. Cost predictions and micro-benchmark results for individual design options could direct the search towards better solutions. Obviously, some choices conflict, require additional steps to work together, or provide chances for optimization. Thus, the real performance can be compared only on real implementations. To date just few and quite similar protocol implementations exists. Protocols should not be compared on their LogP parameters alone. As discussed, a collection of typical usage patterns will be necessary to provide a more detailed insight into performance differences. The performance indicators should be extended to also compare the power consumption.

The design space can be expanded by new hardware features, e.g. compare-and-swap implemented directly in the on-chip SRAM, or hardware-based notification queues. This would allow to predict the improvements over existing protocols even before the actual hardware exists.

Finally, as the message passing is a critical system component and protocols become more complicated, it would be helpful to verify properties like wait-freeness, lock-freeness, or in-order delivery with formal methods.

Acknowledgements

We wish to express our gratitude to our master students Jana Traue, Thomas Prescher and Peter Sauer, who contributed a lot to the discussions and benchmarks. Furthermore we thank Intel for the access to a SCC and the opportunity to contribute to its MARC program. Most specifically we thank Werner Haas and Michael Konow from Intel Research Braunschweig, Germany, for the tremendous insights into the architecture.

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE, 2010, pp. 108–109.

[2] S. Peter, T. Roscoe, and A. Baumann, "Barrelfish on the Intel Single-chip Cloud Computer, Barrelfish Technical Note 005," Tech. Rep., 2010.

[3] J. Nolte, Y. Ishikawa, and M. Sato, "TACO – Prototyping High-Level Object-Oriented Programming Constructs by Means of Template Based Programming Techniques," *ACM Sigplan, Special Section, Intriguing Technology from OOPSLA*, vol. 36, no. 12, December 2001.

[4] C.-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken, "Low-latency communication on the IBM RISC system/6000 SP," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 1996.

[5] T. G. Mattson, R. F. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC Processor: the Programmer's View," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: towards a realistic model of parallel computation," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 1993.

[7] D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa, "Assessing fast network interfaces," *IEEE Micro*, vol. 16, pp. 35–43, February 1996.

[8] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1996, pp. 267–275.

[9] W. N. Scherer, III, D. Lea, and M. L. Scott, "Scalable synchronous queues," *Commun. ACM*, vol. 52, pp. 100–111, May 2009.

[10] K. E. Schauser and C. J. Scheiman, "Experience with active messages on the Meiko CS-2," in *Proceedings of the 9th International Symposium on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 140–149.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 519–538.

[12] K. Chapman, A. Hussein, and A. Hosking, "X10 on the SCC," Presentation, 2011. [Online]. Available: http://communities.intel.com/docs/DOC-6255

[13] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight Communications on Intel's Single-chip Cloud Computer Processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.

[14] I. A. Comprés Ureña, "Lightweight MPI for the Single Chip Cloud," Presentation, 2010. [Online]. Available: http://communities.intel.com/docs/DOC-5844

[15] R. Rotta and J. Nolte, "Low latency collective operations on the Intel SCC," 2011, work in progress.

Fast Fluid Dynamics on the Single-chip Cloud Computer

Marco Fais, Francesco Iorio
High-Performance Computing Group
Autodesk Research
Toronto, Canada
francesco.iorio@autodesk.com

Abstract—**Fast simulation of incompressible fluid flows is necessary for simulation-based design optimization. Traditional Computational Fluid Dynamics techniques often don't exhibit the necessary performance when used to model large systems, especially when used as the energy function in order to achieve global optimization of the system under scrutiny. This paper maps an implementation of the Stable Fluids solver for Fast Fluid Dynamics to Intel's Single-ship Cloud Computer (SCC) platform to understand its data communication patterns on a distributed system and to verify the effects of the on-die communication network on the algorithm's scalability traits.**

***Keywords: Fast Fluid Dynamics (FFD), Computational Fluid Dynamics (CFD), Conjugate Gradient, Distributed Systems, Intel Single-chip Cloud Computer*.**

I. INTRODUCTION

Simulation of incompressible fluids is often used in conjunction with the design and analysis phases of engineering and construction projects.

Fast Fluid Dynamics (FFD) is a technique originally introduced by Foster and Metaxas [2] for computer graphics, used to simulate incompressible fluid flows using a simple and stable approach.

In recent years FFD has been applied to numerous scenarios and its validity has been independently verified by multiple groups [1]. While simulations results diverge from experimental data, the accuracy of the prediction is often sufficient to provide guidance when fast simulation turnaround is required for design optimization and emergency planning scenarios.

FFD techniques use a regular grid spatial partitioning scheme. In order to simulate very large problems the amount of memory a single system can support is often not sufficient. Aggregating collections of systems is often used to simulate large domains and this technique corresponds to employing distributed-memory architecture. Even when memory on a single system is sufficient, the number of computing cores operating in single-image SMP architectures can exhaust the total available memory bandwidth. The overall algorithms scalability can thus suffer, regardless of the amount of available parallelism the algorithms actually exhibit.

The goal of our work is to design and implement a variant of the FFD method to evaluate its scalability traits on a system that employs an on-die network and not to produce the fastest possible implementation.

II. THE SCC ARCHITECTURE AND THE RCCE COMMUNICATION LIBRARY

Intel's Single Chip Cloud system is a novel microprocessor system design based on computing "tiles" organized in a 2D grid topology [5][6]. No hardware support for cache coherence is provided, but hardware support for message passing, message routing and synchronization primitives is available.

The main message communication library available on the system is RCCE [7], and offers basic synchronous message passing and synchronization facilities.

III. STABLE FLUIDS ALGORITHM

Stable Fluids was introduced by Stam [3][4] as a fast, stable technique to solve incompressible fluid field motion; the fluid domain is decomposed in a regular voxel grid. Each voxel contains the density and the velocity at the corresponding spatial location, thus defining a vector velocity field U and a density scalar field D.

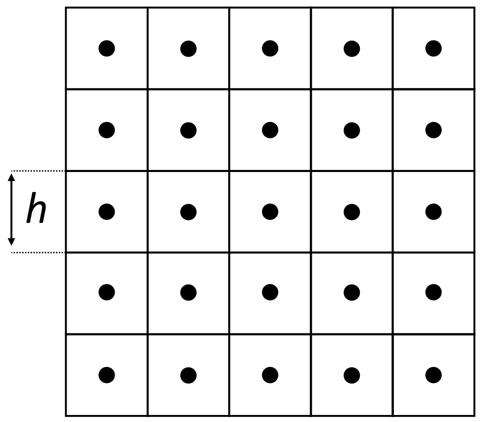


Figure 1. Regular voxel grid

To solve for a time step the simulator performs a series of sequential phases that operate on the velocity field and the density field:

Add velocity forces: compute contribution of external forces on the velocity field.

Diffuse velocity: compute the diffusion effect of the velocity field.

Advect velocity: compute the movement of velocity as affected by the velocity field itself.

Project velocity: compute the effects of mass conservation on the velocity field.

Add density sources: compute the contribution of external density sources on the density field.

Diffuse density: compute the effects of diffusion on the density field.

Advect density: compute the movement of the density field as affected by the velocity field.

IV. Simulating fluids on the SCC

Mapping the Stable Fluids solver on the SCC requires decomposing the fluid field into multiple tiled subdomains and assigning a core to each subdomain. A block partitioning scheme is the most natural solution due to the network topology the cores in the SCC architecture are organized into.

The domain decomposition operation is performed upon starting the simulator. In the current implementation the subdomains' locations and sizes do not change after initialization. The partitions are implicit: system software provides to each core its own network row and column index in the grid, and the total number of cores present in each row and column. Every core can therefore directly compute the size (number of rows and columns) and the origin of its own subdomain.

The effect of this partitioning scheme is that cores that are physical neighbors in the mesh topology operate on adjacent subdomains. This is important for optimizing overall communication latencies, as a significant amount of data dependencies refer to neighboring subdomains.

As a result, almost all communication happens between cores that are direct neighbors in the SCC mesh network. Fig. 1 shows how a part of the domain is mapped onto cores at indexes (0, 0), (0, 1), (1, 0), (1, 1) on the SCC.

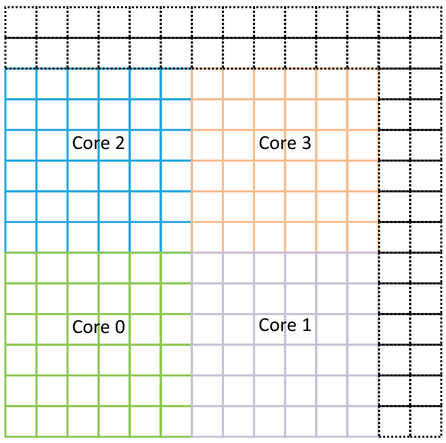


Figure 2. Domain decomposition

For efficiency reasons the fluid domain data is organized in memory in an Array of Structures layout. In this way it is possible to maximize spatial and temporal coherence in the different phases of the algorithm, and hopefully reduce performance degradation due to stalls in the memory hierarchy. Almost all data structures are evenly partitioned among the cores involved in the execution of the solver, the only exceptions are the structures containing details about the voxel type of the other data grids: in the current implementation, these data structures are replicated on each core, as they are involved in handling internal and external boundary conditions.

We implemented the solver as a C++ class library, using template constructs to facilitate changing of the basic data type of the simulation; varying the data type between different levels of precision obviously affects the overall simulation precision, performance and memory usage.

In the next subsections we analyze the individual phases that are performed in solving for a time step in the simulation.

A. Add forces/sources phase

Adding forces to the velocity field and the density field is a trivially parallel operation that doesn't require any data communication across subdomain boundaries; considering F as a vector field of external forces and S an external scalar field of density sources, this phase can be expressed as follows:

The formula for velocity is:

$$U^{n+1}_{i,j} = U^{n}_{i,j} + dt\, F_{i,j}$$

The formula for density is:

$$D^{n+1}_{i,j} = D^{n}_{i,j} + dt\, S_{i,j}$$

B. Diffusion phase

The diffusion phase computes the effect of diffusion on velocity and density in the voxel grid, which involves solving a system of linear equations. In this system *h* represents the size of a voxel as shown in Figure 1. ν represents the fluid viscosity constant, and κ represents the density diffusion constant.

The formula for velocity is:

$$U^{n+1}_{i,j} - \nu\, dt \frac{\left(U^{n+1}_{i-1,j} + U^{n+1}_{i+1,j} + U^{n+1}_{i,j-1} + U^{n+1}_{i,j+1} - 4\,U^{n+1}_{i,j}\right)}{h^2} = U^{n}_{i,j}$$

The formula for density is:

$$D^{n+1}_{i,j} - \kappa\, dt \frac{\left(D^{n+1}_{i-1,j} + D^{n+1}_{i+1,j} + D^{n+1}_{i,j-1} + D^{n+1}_{i,j+1} - 4\,D^{n+1}_{i,j}\right)}{h^2} = D^{n}_{i,j}$$

Our implementation uses the Conjugate Gradient method to solve the linear systems due to its ability to handle internal boundaries. Solving the linear systems results in a strictly data-parallel 5-point stencil data access pattern. Due to the predictable nature of the data access the communication requirements are all statically known. For this reason we can perform all the required data exchanges concurrently at the beginning of the phase then proceed to compute the voxels that do not require subdomain boundary values. At the end, we process the boundary voxels and as a result, completely overlap the data communication of all the cores.

C. Advection phase

The purpose of the Advection phase is to move both density and velocity along the velocity field.

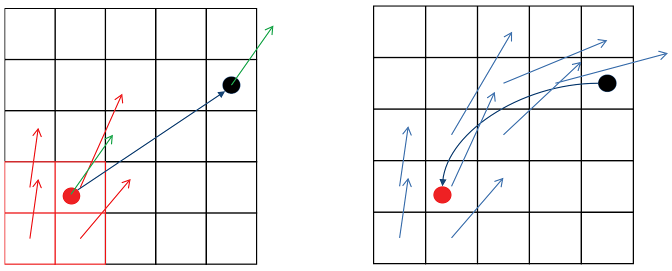


Figure 3. Advection phase backtracking and interpolation

In this phase h represents the size of a voxel as shown in Figure 1, Δt represents the time step, *Interp* represents a 2D linear interpolation function.

The formula for velocity is:

$$U_{i,j}^{n+1} = Interp\left(U_{i,j}^{n}, \begin{pmatrix} i \\ j \end{pmatrix} - \frac{\Delta t}{h} U_{i,j}^{n}\right)$$

The formula for density is:

$$D_{i,j}^{n+1} = Interp\left(D_{i,j}^{n}, \begin{pmatrix} i \\ j \end{pmatrix} - \frac{\Delta t}{h} U_{i,j}^{n}\right)$$

The data access pattern of the Advection phase is unpredictable at compile time, since it is data dependent. In fact, the access pattern depends on the evolution of the velocity field. This model of computation is known as *dynamic stencil* and its efficient parallelization is generally problematic.

Currently our solution involves using an implementation of a request-response protocol that allows one core to request another core for the voxel values of a specific grid. Each core batches its requests into a queue for every other core involved.

The queue data structure is implemented as a collection of fixed size arrays. The queue is initially composed of a single array of requests per destination core. When the space in each array is exhausted it is sent to the target core and a new array is allocated, becoming the current requests storage array. We thus use a data structure that can grow dynamically to accommodate computation requirements. To optimize memory usage, a garbage collection mechanism releases unused requests arrays as required. At the end of the Advection phase, unused arrays in each queue are deallocated in a single operation.

Take for example a queue composed of four arrays, where the three extra arrays have been allocated during a previous Advection phase. If the current Advection phase uses only two arrays, the last two are deallocated at the end of the phase. This strategy is based on the assumption that changes in the velocity field are not abrupt between consecutive executions, thus generating a similar amount of requests. Since we will likely require similar sized sets for the next Advection phase, we don't release all the arrays at the end of the phase.

To compute the Advection phase on a 2D domain, then for each voxel in its local subdomain, each core first computes the global grid indices of its four neighbor voxels (eight in a 3D environment), resulting from the backtracking operation. If all the required voxels are local, the final voxel value is computed. Otherwise a new request is added to the queue of the core which owns the subdomain containing each remote voxel, and the computation of the final voxel value is deferred.

In summary, since the request-response protocol introduces some communication overhead, we use a batching strategy to minimize overhead. Core specific requests are batched and sent at the end of the local computation or when the current request array is full. A communication thread running on each core monitors incoming requests from other cores, then creates messages containing the requested data and enqueues the messages for transmission back to the requesting cores.

On each core when all the required remote data has been successfully received, all the previously deferred voxels can finally be computed.

This approach has proven to be quite efficient due to the low communication latency on the SCC mesh network. However it is important to underline that performance is highly data dependent. For example, small velocities and small time steps imply a small number of voxels with remote dependencies, with the remote voxels likely being stored in the memory of physically neighboring cores on the SCC mesh network. This results in a limited amount of communication between direct physical neighbors, minimizing both the required bandwidth and message routing distance on the mesh, in turn minimizing latency.

In a different scenario, large velocities and/or large time steps introduce large amounts of voxels with remote dependencies, which may involve communicating across larger routing distances on the mesh. This implies additional hops in the communication network, more message collisions/conflicts and in general, higher communication latency.

The implementation of our request-response protocol on the SCC required functionality not available in the RCCE library, which only supports pure send-receive communications. Our protocol requires both asynchronous message passing and data-dependent message destinations. We then extend the RCCE library with additional functions which will be discussed in section V.

D. Projection phase

The Projection phase corrects the velocity field to ensure conservation of mass, and involves computing the current flow gradient field and solving the Poisson equation to force the flow in and out of each voxel to be equivalent.

The current flow gradient field is easily obtained using the current velocity field, and only requires statically known communication of voxel values along borders of the subdomains. The solver then proceeds to solve the following linear system, where P represents the pressure field in the Poisson equation:

$$P_{i+1,j}+P_{i-1,j}+P_{i,j+1}+P_{i,j-1}-4\mathrm{P}_{i,j}=\left(U^x_{i+1,j}-U^x_{i-1,j}+U^y_{i,j+1}-U^y_{i,j-1}\right)h$$

Solving the linear system is accomplished by re-using the Conjugate Gradient method already applied during the Diffusion phase. The data access pattern is the same and we can easily overlap all data communication by using asynchronous communication functions.

V. RCCE EXTENSION

Due to the data-dependent and unpredictable nature of the data access pattern in the Advection phase, the basic RCCE library provided by the SCC SDK is not suitable. The RCCE API does not contain functionality to efficiently listen to incoming messages which can arrive at any time from any core. It also lacks support for asynchronous communication, which is fundamental to implement our request-response protocol.

Some other communication libraries have been developed since the SCC architecture has been released, iRCCE [11] and RCKMPI [10] are the most popular. The former is an extension to the RCCE library while the latter is an implementation of the MPI standard for the SCC.

iRCCE is a promising library, as it adds non-blocking point-to-point communication capabilities to RCCE and introduces a new, smarter version of the "send/receive" functions. This alternative communication scheme is referred to as "pipelined". It splits the Message Passing Buffer (MPB) into two chunks, allowing both the sender and the receiver to access the MPB at the same time, in parallel. While the new features introduced by the iRCCE extension are useful in the context of our work, they are still not sufficient for our purposes. In particular it is not possible to efficiently receive a message without knowing the sender in advance, and mixing of non-blocking communication requests with blocking collective operations is not supported.

In our computation we often need to compute the norm of a vector partitioned among all the cores' address spaces. Without mixing point-to-point communication requests with collective operations, we would require a barrier every time we need to compute a norm. Moreover, the pushing mechanism used by iRCCE to allow the communication to progress leads to a more complicated and less portable application code. One of our purposes is to write the algorithm in a way that minimizes the effort required to port the code to different distributed memory architectures, a cluster, for example. For this reason we decided to isolate the architecture-dependent aspects of the communications in a separate thread that emulates a communication controller, for example a DMA engine, or a hardware thread in a Simultaneous Multithreading system.

Using a dedicated thread for communication management introduces a small amount of overhead due to the context switches between the computation thread and the communication thread. However this solution is more flexible, because the communication management thread waking pattern (and hence the context switch frequency) is configurable. An additional advantage is that the application code is cleaner, as the calls to the functions that allow communication progress is not interleaved with the algorithm code.

RCKMPI is one of several implementations of the MPI standard [8] developed for the SCC architecture, derived from the MPICH2 implementation [9]. Its main advantage is that many parallel applications programmers are familiar with MPI and a parallel application written with RCKMPI only needs to be recompiled with an MPI implementation to be ported to a variety of distributed systems. However, RCKMPI is affected by some of the issues already discussed: in particular the need for a receiver to statically know the rank of the sender and the size of the message.

For these reasons we implement our own extension of the basic RCCE library, reusing most pre-existing data structures to support asynchronous communication and a request-response protocol. Our extension uses an interface similar to the standard TCP/IP "select" function, and introduces a non-blocking operation to quickly identify incoming messages and operate on them.

Our "select" function takes an array of chars as input, which will be filled with the ranks of the cores that are requesting to initiate a communication. Upon completion, our function returns the number of valid entries in the array. Our "select" is based on custom variants of the low-level RCCE "send_general" and "receive_general" primitives.

Our new "send" adds a header to the message containing the type of the message and its size in bytes, so that the new "receive" does not require the size of the message as a parameter.

The type of the message is an additional one-byte field that can be used by the sender program to mark the content of the message, so that the receiver program can perform different tasks according to this information.

We allocate two new sets of communication flags in the MPB, that are used for signaling by all the new functions ("select", size-agnostic "send" and "receive"). This way we can handle both point-to-point and collective communication requests without signaling conflicts. The new flags allocated on the MPB reduce the size of the largest data chunk transferable by 48 bytes (using flags of 1 byte), but we consider this trade-off acceptable.

VI. RESULTS

We tested our solver on domains of different sizes. For each experiment we incremented the size of the domain proportionally to the number of cores involved, which provided a good measure of the impact of communications on the overall performance.

For each domain size, the domain partitions were assigned to neighboring cores in the mesh network by using the logical layout provided by the RCCE library. This ensured neighboring logical domain partitions were assigned to physically adjacent cores.

While our experiment provided a test of both the processor cores and the on-chip mesh communication network, initially we only used the default frequencies for the processor cores and communication mesh.

The focus of the tests was not absolute performance but an analysis of the scalability traits.

TABLE 1. EXECUTION TIMES FOR ONE TIME STEP OF SIMULATION

Domain Size	Cores	Time (seconds)
1024 X 1024	1 X 1	116.76
2048 X 2048	2 X 2	142.63
4096 X 4096	4 X 4	153.00
6144 X 6144	6 X 6	153.45
8192 X 6144	8 X 6	153.85

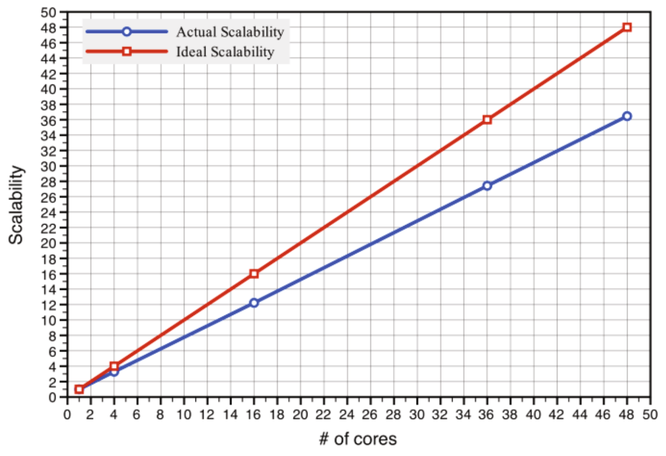


Figure 4. Scalability results

Table 1 reports the execution times for solving one simulation time step using single precision floating point as the basic data type. Fig. 1 represents the actual scalability of the current implementation of the solver on the SCC and compares it with the ideal scalability curve.

The reference single-core solver used for obtaining the baseline timing does not contain any form of communication. The multi-core distributed solver thus introduces a certain amount of overhead even in its minimal 2x2 cores implementation.

The results demonstrate that while communication indeed introduces overhead, the overall scalability traits of the algorithm are good. The overhead is constant beyond 4x4 cores. As a result the solver exhibits a constant execution time for larger domains, up to the maximum size tested. The memory used approached the upper limit of the SCC system used for our tests.

VII. CONCLUSION AND FUTURE WORK

The approach chosen in our implementation exhibited fairly good scalability, with the experimental results being quite promising. We plan to continue the work with the introduction of additional optimizations for performance, communication and synchronization.

This paper focused on simulations performed on 2D domains, but work is already underway on an extension of the solver to 3D domains. The performance optimization work will concentrate on the improvement of memory access, additional exploitation of asynchronous data transfer, and better exploitation of temporal coherence, especially in the Advection phase.

Variations of the cores and mesh frequencies will also be evaluated to understand their effects on power usage, and to find the optimal frequencies that allow the fastest algorithm performance while minimizing power usage. The chosen domain partitioning layout is expected to benefit this experiment by minimizing the average distance messages need to travel on the mesh network.

ACKNOWLEDGMENT

The authors would like to acknowledge Dr. Jos Stam for the precious collaboration and guidance on the algorithm details, and Alex Tessier for his insightful comments.

REFERENCES

[1] W. Zuo and Q. Chen, “Validation of fast fluid dynamics for room airflow ”, IBPSA Building Simulation 2007, Beijing, September 2007

[2] N. Foster and D. Metaxas, “Realistic animation of liquids”, Graphical Models and Image Processing, volume 58, number 5, 1996, pp.471-483

[3] J. Stam, “Stable fluids”, In SIGGRAPH 99 Conference Proceedings, Annual Conference Series, August 1999, pp.121-128

[4] J. Stam, “Real-time fluid dynamics for games”, Proceedings of the Game Developer Corner, March 2003

[5] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, T. Mattson, “A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS”, Proceedings of the International Solid-State Circuits Conference, Feb 2010

[6] T. G. Mattson, R. F. Van Der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, “The 48-core SCC Processor: the programmer's view”, Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, p.1-11, November 13-19, 2010

[7] T. Mattson, R. Van Der Wijngaart, “RCCE: a small library for many-more communication”, Intel Corporation, May 2010, Software 1.0-release

[8] Message Passing Interface Forum, “MPI: a message passing interface standard”, High-Performance Computing Center Stuttgart (HLRS), September 2009, Version 2.2

[9] “MPICH2”, Internet: http://www.mcs.anl.gov/research/projects/mpich2, [June 20, 2011]

[10] I. A. Comprés Urena, “RCKMPI user manual”, Internet: http://communities.intel.com/docs/DOC-6628, January 2011

[11] C. Clauss, S. Lankes, J. Galowicz, T. Bemmerl, “iRCCE: a non-blocking communication extension to the RCCE communication library for the Intel Single-chip Cloud Computer”, Internet: http://communities.intel.com/docs/DOC-6003, February 2011

Task Parallelism on the SCC

Andreas Prell and Thomas Rauber
Department of Computer Science
University of Bayreuth, Germany
{andreas.prell,thomas.rauber}@uni-bayreuth.de

***Abstract*—Task parallel programming has become a popular and effective approach for programming multicore systems. An interesting question is always how to implement the task abstraction on a new architecture. In this short paper, we look at Intel's Single-Chip Cloud Computer (SCC) and propose a tasking environment for the SCC. We compare two different runtime systems with work-sharing and work-stealing schedulers.**

I. Introduction

Task parallel programming allows programmers to identify opportunities for parallelism while leaving the details of parallel execution to the runtime system. Consequently, runtime systems become a major factor in the success of this programming model. As new architectures continue to emerge, much work has to go into building task abstractions that map well to increasingly parallel hardware.

The Single-Chip Cloud Computer (SCC) is a 48-core experimental processor created by Intel Labs to serve as a platform for manycore software research [1], [2]. The SCC is designed to be a message-passing chip where cores communicate through non-cache-coherent shared memory. In this short paper, we describe our approach to task parallel programming on the SCC. We implement a tasking environment and show that efficient runtime schedulers can be built by taking advantage of the processor's on-chip Message Passing Buffers (MBPs).

II. Tasking on the SCC

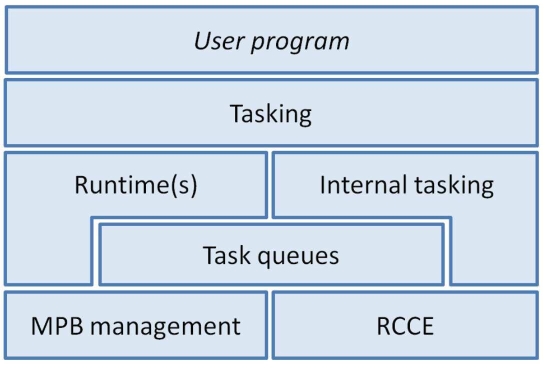


Fig. 1. A tasking environment for the SCC processor.

Figure 1 shows the components of our tasking environment for the SCC processor. At the lowest level, we use some functionality from RCCE [2], as well as our own library routines for accessing MPB memory. Worker threads are based on the RCCE notion of "units of execution" (UE), which are mapped to the cores of the chip. Each UE is assigned a rank from 0 to $N-1$, where N is the number of UEs the program is running on. We use these ranks to define our worker IDs: UE 0 becomes worker 0, UE 1 becomes worker 1, and so on. We choose worker 0 as the (default) master to run the main program. Tasks that are spawned are picked up and executed by workers.

To store tasks in MPB memory, we have implemented a double-ended queue (deque) that is similar to the work-stealing deques of [3], [4], but lock-based instead of lock-free. Locks are the only way to achieve synchronization; no atomic operations, such as CAS, are available that work across the cores of the SCC. On top of that, there are only 48 test-and-set registers (one per core), meaning that at most 48 locks can be used at a time. Tasks are stored in a circular array of fixed size, with *head* pointing to the oldest task. New tasks are inserted at *tail*. A single lock per deque is sufficient to allow deque operations at opposite ends to proceed concurrently most of the time, as long as only one worker inserts tasks. If more than one worker inserts tasks, all accesses require locking. Using two locks per deque would allow for more concurrency between enqueue and dequeue operations [5], but with two workers sharing a deque, inserting a task (a frequent operation!) becomes less efficient.

III. Runtime Systems

The runtime system is the central component that implements the scheduler. The scheduler is responsible for assigning tasks to worker threads and performing load balancing. There are two general approaches to load balancing a computation: work-sharing and work-stealing.

A. Work-Sharing Scheduling

Work-sharing is based on the idea of redistributing tasks to underutilized workers. A trivial implementation of a work-sharing scheduler uses a single queue that is shared among all workers. Because of the scalability problems of centralized task pools, we explore a slightly different implementation. Workers can keep their tasks in private memory, but are required to share some of their work when the need arises. Every worker maintains a queue in private memory (no synchronization is required here because workers have no way to access other workers' private memory). In addition, we allocate a deque in the MPB of the master thread for load balancing. Workers that have enough tasks are responsible for sharing; workers that run out of tasks can pick up new work from the deque.

B. Work-Stealing Scheduling

In work-stealing scheduling, idle workers take the initiative to find new work. Every worker has a local deque on which it operates. Whenever a worker finds its deque empty, it attempts to steal a task from the deque of another worker, which, in the ideal case, is not disrupted by the steal.

We currently support two strategies for selecting victims: randomized and latency-oriented work-stealing. In randomized work-stealing, victims are chosen uniformly at random [6]. In latency-oriented work-stealing, victims are selected based on their distance in terms of on-chip network hops. A thief always tries to steal from the same tile, before it checks for tasks on other tiles, in increasing order of distance, up to a maximum of 8 hops [2].

C. Synchronization

Synchronization is required to coordinate the execution of tasks. We have implemented a task model similar to that of OpenMP 3.0 [7], [8] with two primary synchronization constructs: (1) A *taskbarrier* waits for the completion of all pending tasks. This form of synchronization is coarse-grained and can only be invoked from the master thread. (2) A *taskwait* waits for the completion of all direct descendants of the current task. This form of synchronization allows fine-grained control over parent-child dependencies.

A *taskbarrier* marks a global synchronization point: all tasks must complete before execution can continue past the barrier. Only the master thread is allowed to check into the barrier. Workers keep running their scheduling loop and eventually become idle after finishing all tasks, at which point the master returns from the barrier. The master doesn't simply wait inside the barrier, but helps make progress when work is available.

A *taskwait* keeps track of the children that a parent task is waiting for. In order to provide an efficient implementation, we allocate task counters from MPB memory. A task counter is atomically updated whenever a child is created or terminates. Once the *taskwait* completes by reading a count of zero, the memory for the task counter can be freed or otherwise marked as reusable. A direct consequence of this implementation is that deep recursion may overflow MPB memory. We currently deal with this problem by limiting the runtime's MPB usage. New tasks that would exceed the limit are not created but executed sequentially. Tasks are created again once enough memory has been reclaimed.

In addition to the synchronization constructs derived from OpenMP 3.0, we provide simple explicit *futures* [9]. A future is essentially a placeholder for the result of an asynchronous computation (task). Forcing a future means waiting for the result to arrive, rather than waiting for the task to complete. A worker that forces a future is free to start other tasks while the future is pending. Again, for efficiency reasons, we allocate future objects from MPB memory. A future object contains a synchronization flag that is set once the result is available.

IV. Preliminary Experimental Results

We compare performance on three microbenchmarks that stress the ability of the runtime system to find work and perform load balancing:

- Simple Producer-Consumer (SPC) benchmark: A single producer (worker ID 0) spawns n consumer tasks. Consumer tasks perform some computation for time t.
- Bouncing Producer-Consumer (BPC) benchmark [10]: A variation of the producer-consumer benchmark with two kinds of tasks, producer and consumer tasks. Each producer task creates another producer task followed by n consumer tasks, until a depth of d is reached. Producer tasks do nothing besides spawning tasks. Consumer tasks perform some computation for time t.
- Fibonacci-like tree-recursive benchmark: Modeled after the tree-recursive process of computing Fibonacci numbers. Each task $n \geq 2$ creates two child tasks $n-1$ and $n-2$ and waits for their completion. Leaf tasks $n < 2$, which end the recursion, compute for time t.

The SCC is used in the default configuration, with cores clocked at 533 MHz, routers at 800 MHz, and DDR3-800 memory. Work-sharing and work-stealing runtimes are configured to use a deque size of 10[1]. Work-stealing proceeds by selecting victims at random.

A. Work-Sharing Scheduling

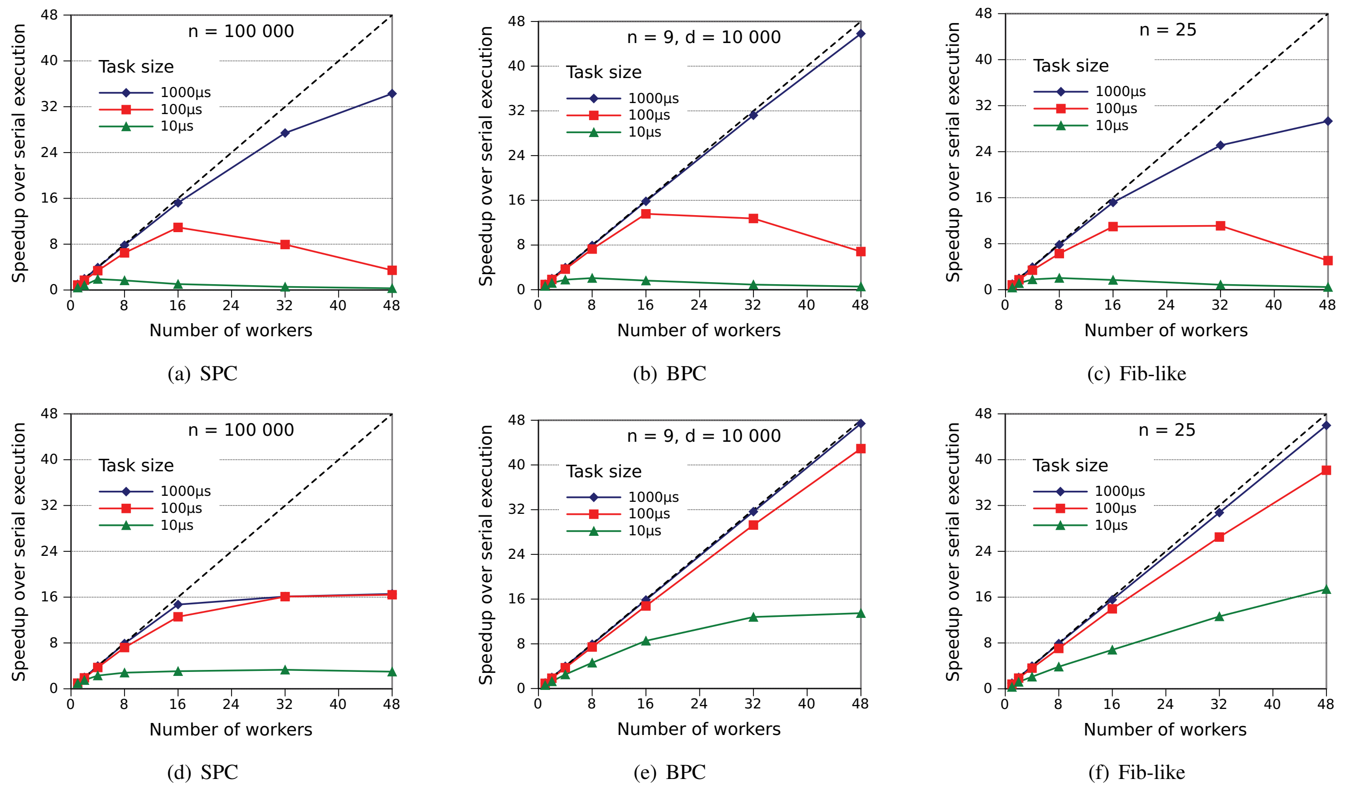
The top row of Figure 2 shows the results for the work-sharing runtime. Work-sharing is only an option for relatively coarse-grained parallelism. Despite our attempt to reduce contention somewhat by setting up private queues, the single shared queue remains a significant bottleneck that quickly limits scalability when workers are frequently searching for tasks. In fact, the contention on the deque grows so high that performance starts to degrade. To work against this, we would have to manage contention explicitly, for example by following a backoff strategy before trying to access the deque again.

B. Work-Stealing Scheduling

The bottom row of Figure 2 shows the results for the work-stealing runtime. Work-stealing delivers good performance on medium to fine-grained parallelism. Very fine-grained parallelism on the order of a few microseconds per task is hard to exploit due to the cost of task creation and scheduling.

We see only one case where work-stealing is outperformed by work-sharing: on the SPC benchmark with large tasks. There are two reasons for this: (1) Our work-stealing implementation uses deques in MPB memory, which are fixed-size, and when full, require that new tasks are executed sequentially. Doing so can cause load imbalance, especially if that worker is the only source of tasks. (2) Unlike work-sharing, work-stealing involves probing deques, often randomly, as in this case, until a deque is found from which it is possible to steal.

[1]A deque of 10 tasks takes up 1504 KB, which is roughly 20% of the available MPB memory of a worker.



(a) SPC (b) BPC (c) Fib-like

(d) SPC (e) BPC (f) Fib-like

Fig. 2. Performance of work-sharing (top row) and work-stealing (bottom row) schedulers on SPC, BPC, and tree-recursive benchmarks. The numbers of tasks are 100 000 for SPC and BPC and 242 784 for Fib-like.

This probing, especially when not strictly needed, can increase the time it takes to find new work.

Several strategies can help improve performance on SPC type of workloads. Increasing the deque size, perhaps dynamically at runtime, or trying to insert tasks in remote deques could effectively prevent producers from inlining large tasks. Adapting work-stealing to be more directed towards the deque of the producer could reduce consumer idle time.

V. Summary and Future Work

Our preliminary results show that tasks can be moved efficiently around the cores of the SCC processor. Work-stealing turns out to be much more practical than work-sharing, especially in the case of fine-grained parallelism.

We consider two directions for future work: (1) simplifying the programming model by adding compiler support for task parallelism and (2) exploring new scheduling approaches that can be adapted for future manycore processors.

The hybrid nature of the SCC makes it an interesting platform for software research. Beyond finding out what works well in the context of the SCC, we think it's important to explore ideas that can be generalized for manycore computing.

Acknowledgment

We thank Intel for granting us access to the SCC as part of the MARC program. Our work is supported by the Deutsche Forschungsgemeinschaft (DFG).

References

[1] J. Howard *et al.*, "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS," in *Proceedings of the 2010 IEEE International Solid-State Circuits Conference*, 2010, pp. 108–109.

[2] T. G. Mattson *et al.*, "The 48-core SCC Processor: the Programmer's View," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread Scheduling for Multiprogrammed Multiprocessors," in *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '98. New York, NY, USA: ACM, 1998, pp. 119–129.

[4] D. Chase and Y. Lev, "Dynamic Circular Work-Stealing Deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '05. New York, NY, USA: ACM, 2005, pp. 21–28.

[5] D. Majeti, "Lightweight Dynamic Task Creation and Scheduling on the Intel Single Chip Cloud (SCC) Processor," in *Proceedings of the Fourth Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, ser. PLACES '11, 2011, pp. 35–42.

[6] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, pp. 720–748, September 1999.

[7] "OpenMP Application Program Interface Version 3.0," http://www.openmp.org/mp-documents/spec30.pdf, May 2008.

[8] E. Ayguadé *et al.*, "The Design of OpenMP Tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, pp. 404–418, March 2009.

[9] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 501–538, October 1985.

[10] J. Dinan *et al.*, "Scalable Work Stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 53:1–53:11.

An Empirical Feedback Provider for Multi-Core Schedulers

Waqaas Munawar, Janmartin Jahn, Artiom Aleinikov, Jian-Jia Chen, Jörg Henkel
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
(munawar, jahn, aleinikov, jchen, henkel)@kit.edu

***Abstract*—Packaging multiple cores per chip is the most viable design methodology of continuing improvement of computational performance. The use of multi-core processors has been challenging due to many issues from both software and hardware aspects. In this paper, we present a practical setup to investigate the effects of design choices in multi-core scheduling algorithms on different parameters of interest. These parameters include the performance overhead, power consumption, thermal effects and cache coherency issues. As a use case, we employ the SCC platform and a robotic head, while the controlling application has been parallelized to reap performance benefits of a large number of cores. This paper presents the current state of our implementation and its design choices, and also outlines the remaining research challenges.**

I. Introduction

Multi-core systems are expected to increasingly employ homogeneous and heterogeneous cores to accommodate performance requirements without significant increase of power consumption and heat dissipation. This presents an additional challenge for programmers, system designers, and researchers. Additionally, multi-core architectures depart from the paradigm of sharing memory resources, as this does not scale well with a large number of cores [1]. Moreover, the access latency and bandwidth of different on and off chip memory banks vary due to varying distance, network load, or link vitality on the network on chip (NOC). Orthogonal to performance considerations, power optimizations can benefit immensely from the ability to dynamically tweak the frequency and voltage of the cores. The presence of these variables obligates non-trivial decisions from the scheduler that manages the system. Additionally, constraints on power consumption, temperature, and different performance requirements, as well as the absence of shared memory, lead to the challenges to design effective schedulers.

Unfortunately, to the best of our knowledge, there exists no setup to profile a novel scheduler's performance under realistic practical settings on large-scale multi-core hardware platform, such as Intel's Single Chip Cloud Computer (SCC) with 48 cores. In this paper we present our setup on such a platform along with its implementation status and the early results. The core idea behind this platform is to provide the necessary infrastructure for task migration and profiling to support the empirical comparison among different scheduling and core-assignment algorithms. As our proposed system serves as a runtime facilitation layer for the scheduler, it faces additional constraints on the performance of its subcomponents. Hence, we keep it transparent in the normal functioning of the overall system. Our platform is hardware agnostic and can be utilized on most multi-core architectures. We use Intel's Single Chip Cloud Computer (SCC) along with the Barrelfish operating system [2] as our first implementation test-bed. The first application we profile is a parallelized robotic software package that incorporates stereo vision, object recognition and robotic control. To increase the level of parallelism, we have transformed the robotic software package to a software pipeline, where each stage runs in a separate thread and communication between stages is implemented using message passing. Stages run in parallel and do not share memory, thus they can be assigned to and migrated among different cores.

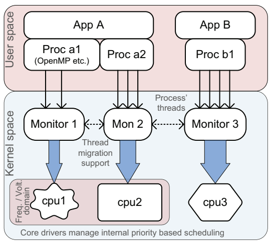
II. Architecture

The central idea of the system is to provide an interface to the scheduler through which it can accomplish two main objectives: *(i)* it can implement its decisions regarding the frequency selection, voltage control and task mapping and remapping on different cores and *(ii)* the scheduler can see the results of its decisions in terms of the specified performance metric such as power savings, temperature control and response time. We adopt a hierarchical scheduler so that the level of abstraction increases with acceptable decrease of control granularity [3]. In order to support such schedulers at different levels of abstraction, we need a hierarchical facilitation system, providing services at appropriate levels.

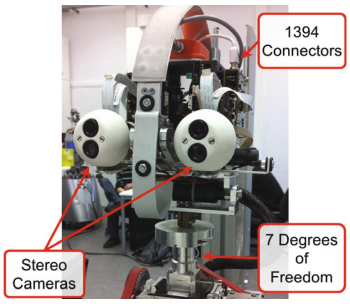
Figure 1a summarizes the available interface for facilitation and feedback services. Considering the first of the two objectives, a hierarchical structure would imply multiple levels of appropriate facilitation operators for the corresponding layer. For an example, let us consider the task migration mechanism. Its objective varies according to the corresponding level in hierarchy. At top-most level it must address the shift of application from one set of cores to another considering the capability and constraints of hardware and its interconnects. At the lower levels it deals with moving a group of threads that belong to a task, from one core to another to actuate, for example, temperature reduction. Similarly, other facilities e.g. frequency and voltage control, memory interconnect and priority reassignment, are accessible via other actuation commands at appropriate level in hierarchy. Like facilitation, the requirements posed by profiling (i.e. the second objective) also demand a layered feedback provider.

III. Implementation

The main principles guiding our implementation of the discussed platform are: *(i)* The developed solution should not be tied to a specific hardware platform and *(ii)* the constant runtime overhead contribution from the framework should be minimal. Therefore, the Barrelfish multikernel OS [4], which satisfies our objective, is adopted as our implementation. The reason for selecting Barrelfish is its ability to

Scheduler structure	Controlled aspect	Facility actuation	Feedback generated
	Application requirements	Hardware capability +Interconnects	QoS achieved + Timeliness + Scheduler fairness + Resource utilization
	Frequency control + Voltage control	Load balancing + Temperature control	Performance/resource
	Priority management	Priority reassignment	Power consumption/core + Core temperature

(a) Architecture of the system in relation to a generic hierarchical multi-core scheduler



(b) Head of the humanoid robot

Fig. 1: Architecture and the case study.

capture and present the underlying hardware's heterogeneity in logic-constructs, thereby easing scheduler's adaptability. However, the architecture design in Figure 1a makes it easy and possible to port to other operating systems, like SMP Linux. It also holds for task migration algorithms. The task migration capability is an important aspect in our system. A number of task migration solutions have been discussed in the literature [5] and any of these, given it fulfills the constraints at corresponding level, can be employed. Our modular architecture facilitates this replacement. The implementation of the presented architecture is still an ongoing task.

IV. A Use Case

The first usage of the system is on the humanoid robotic head's (shown in Figure 1b) software package of the robot ARMAR [6]. It continuously captures stereo images from two FireWire (IEEE 1394) cameras and uses computer vision algorithms to follow a moving object. The source images are first transformed and segmented. Then, moving objects are discovered, stereo matching is performed and the robot performs actions based on their 3D coordinates.

The robotic application exhibits several types of parallelism. As the correct order of the iterations must be maintained, a straightforward parallel execution of loop iterations on different cores would need significant synchronization overhead. However, the filters (FilterHSV2, Erode, Dilate) loop over the image and can be loop-parallelized with OpenMP. Unfortunately such parallelization requires to operate on shared memory, which contradicts to the paradigm of the SCC platform of departing form shared memories, and is thus infeasible.

To generate a reasonably balanced level of parallelism in terms of granularity, we have chosen to create a software pipeline from the main application loop. Each stage operates on private data and forwards its results using message passing. The schematic flow of the application and its stages is depicted in Figure 2. Each box represents a pipeline stage that continuously loops its execution as soon as its input data is available.

By the use of this application we will be able to collect the vital statistics to compare different scheduling algorithms among each other in terms of their fairness, power optimization, responsiveness, effect of hardware interconnects etc.

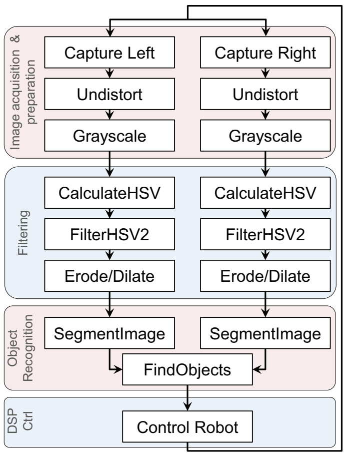


Fig. 2: Robotic Application Pipeline

V. Conclusion

In this paper we have outlined the need for empirical results for next generation schedulers and have presented the architecture of a system to acquire them. We believe this will be helpful for scheduler design, power optimization, temperature management and performance evaluation.

References

[1] J. Howard *et al.*, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *ISSCC'10 IEEE International*, pp. 108 –109.

[2] A. Schpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the barrelfish manycore operating system," in *Proceedings: Managed Many-Core Systems*, 2008.

[3] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe, "Design principles for end-to-end multicore schedulers," ser. HotPar'10, 2010, pp. 10–10.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," ser. SOSP '09, pp. 29–44.

[5] J. Jahn, M. A. Al Faruque, and J. Henkel, "CARAT: Context aware runtime adaptive task migration for multi core architectures," in *DATE '11*, 2011.

[6] T. Asfour, K. Regenstein, P. Azad, J. Schroder, A. Bierbaum, N. Vahrenkamp, and R. Dillmann, "Armar-iii: An integrated humanoid platform for sensory-motor control," in *IEEE-RAS Humanoid Robots*, 2006, pp. 169 –175.

A Scalable and Robust Runtime Environment for SCC Clusters

Björn Saballus, Stephan-Alexander Posselt and Thomas Fuhrmann
Technische Universität München
Boltzmannstrasse 3, 85748 Garching/Munich, Germany
{saballus|posselt|fuhrmann}@in.tum.de

***Abstract*—Today, computing systems – especially high-performance computing systems – become more and more parallel. As a consequence, each system's memory becomes more fragmented across a large network of computing elements, and also across the various different processing cores within a single element. Applications that need to be executed on a whole cluster are most easily implemented when they can pretend to run on a single system. However, providing coherence in hardware is getting more expensive as the number of cores grows. Maintaining memory consistency in software is hard to get right.**

As a possible solution to these problems, we develop a distributed Java virtual machine (VM). The VM's system model respects the memory hierarchy of modern computer architectures as well as their hardware failure rates. Additional features such as transparent thread and object migration help to achieve a high utilization of the available compute resources, while hiding the underlying hardware from the programmer.

I. Introduction

Since about 2005, a fundamental change in software and hardware development is under way. The physical limitations lead the processor design from fast, single core chips to massively parallel, heterogeneous processors. Today, an increase in processor performance can only be achieved by increasing the number of processing cores. However, the growing number of cores also aggravates the problems with concurrency. Hardware-based coherence among the cores is expensive and hard to achieve in many-core processors. Similarly, lock-based consistency between the threads is hard to get right.

Hence, this development imposes a high burden on the software developer, who has to deal with memory consistency, as well as with the distributed and potentially heterogeneous nature of the underlying system.

The J-Cell project[1] aims at hiding the heterogeneous and distributed nature of clusters of many-core processors from the software developer. Additionally, a *software transactional memory* (STM) system shall support the developer to ease the implementation of parallel applications. For this purpose, we develop a scalable and robust runtime environment that provides a *single system image* (SSI) [1] across all cores and all processors in a compute cluster. To this end, we address the following research questions:

- How can we implement a massively scalable STM?
- How can we hide the latencies of the memory hierarchy?
- How can we efficiently protect an application from hardware failures?
- How can we consolidate applications on the cores to save energy?
- How can we efficiently distribute the workload to increase hardware utilization?

We seek to solve these challenges by developing a *virtual machine* (VM) that accesses memory on behalf of the threads. Threads and data can migrate transparently across cores and machines, while our runtime environment ensures the consistency of the data in face of concurrent access. In addition, it can maintain a certain level of redundancy of the data to protect against hardware failures.

II. System Overview

Our research focuses on *unlimited scalability by completely eliminating all centralized components*. The main component of our system is a fully-decentralized virtual machine that scales to vast numbers of cores, processors, and machines in a cluster. On each core runs an individual VM instance in its own thread. Each of these VM threads, again, executes one or more Java threads. Together, all these instances collaborate to create an SSI that allows one or more concurrent applications to run transparently on heterogeneous multi-core machines. The VM instances distribute code, objects, and threads onto the compute resources, which may be added or removed at run-time.

To this end, the system builds an ad-hoc network of processors and cores. A fully decentralized object location and retrieval algorithm facilitates the access to distributed shared objects [2]. A multi-version STM system, the DecentSTM [3], coordinates concurrent access to these objects with the help of a fully decentralized consensus protocol. To deal with node failures, our system uses a decentralized recovery mechanism that works on well-chosen, outdated versions of data to avoid explicit checkpointing [4].

A. Decent Virtual Machine

Our starting point is the *Java virtual machine* (JVM). We are also developing a library for C, but the JVM has a well-defined memory model that lends itself to distribution and STM. Java has gained a lot of interest in engineering and scientific computing in the recent years. Taboada et al. [5] analyze current research projects (as of 2009) that use Java

[1]J-Cell is supported by the German Ministry of Education and Research under grant number 01IH08011.

for High Performance Computing (HPC). They conclude, that Java is well suited for hybrid shared memory (intra-node) and distributed memory (inter-node) architectures because Java threads support shared memory, and Java network support assists distributed memory communication.

Instead of modifying an existing JVM, we decided to develop our own, customized VM. This gives us full control over the VM: We can freely design the interpreter, memory management, and garbage collection. Furthermore, we are free to choose the internal representation of objects and execution contexts to enable easy object and thread migration.

The result is the *Decent Virtual Machine* (DecentVM) [6] design; a fully decentralized, distributed virtual machine with a small code footprint that is compliant with the Java standard. An off-line transcoder converts the Java bytecode of an application to a custom, optimized instruction set. This leads to a less complex VM implementation.

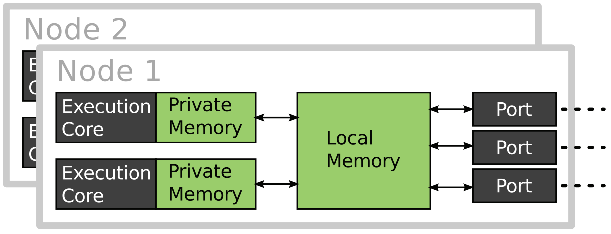


Fig. 1. The DecentVM hardware model

The DecentVM supports the *partitioned global address space* (PGAS) programming model. Its memory model is derived from a *non-uniform memory access* (NUMA) architecture and distinguishes between *logical* (private or global) and *physical* (private, local or remote) memory.

Logically private memory maps to the *physically private* memory. It is only used and accessed by a single VM instance, which may execute multiple Java threads. Nevertheless, the VM itself takes precautions that no Java thread can access the memory of any other thread that is executed by the same VM instance. Physically, this memory is tightly coupled with a single core of the processor, e. g. registers or caches.

Logically global memory is shared among all VMs of a cluster. It does not have a physical representation of its own, but consists of all the *physically local* memories that are littered across different address spaces. From the perspective of a single node, the own *physically local* memory is shared by multiple VM instances that reside in the same address space, e. g. on the cores of the same processor, cf. Fig. 1. On multi-core and multi-processor systems, this usually maps neatly onto the local RAM. Again, from the perspective of a single node, the *physically local* memories of all other nodes are combined into the *physically remote* memory. It is the task of the VM to make all these local memories appear as a single system image, i. e. the logically global memory.

B. Objects

Java and thus the DecentVM work with *objects*, which can be Java language objects and arrays, but also execution contexts and code.

		Physical		
		private	local	remote
Logical	LOC	Mem. Addr.	–	–
	GAO	–	Mem. Addr.	GUID

TABLE I
REFERENCE REPRESENTATION OF GAOS AND LOCS

Due to the differences in memory types and the DecentSTM algorithm, the DecentVM object model distinguishes between *globally accessible objects* (GAOs) and *local object copies* (LOCs). Note that this distinction is only made at the VM level, not at the application level.

LOCs are stored in private memory. They are only accessible by the thread to which they belong. Because the LOC resides in the same address space as the thread, it is directly accessible via a local memory address.

GAOs logically reside in the global memory and can be shared among different threads on different nodes. Therefore, they cannot be referenced with a plain memory address because local memory addresses are only valid within the address space of the respective VM. Instead, we introduce a new type of reference: The *globally unique identifier* (GUID) is location independent and suffices to uniquely identify a GAO within the whole cluster. Additional location information is needed to be able to find the node on which the GAO resides. The GAO's location may change over time due to object migration or recovery from node failures.

A GAO is always stored in the physically local memory of some node. Hence, it is always directly accessible on this node via a local memory address. Internally, the DecentVM uses various tables to map local memory addresses to GUIDs and vice versa. See Table I for the different reference representations of LOCs and GAOs depending on the type of the physical memory that stores it.

Besides by their location, objects are also classified by the way in which a reference to the object is acquired. Thus, we distinguish between *dynamic* and *static* objects.

Dynamic objects are e. g. Java objects (instances of a class), arrays, or execution contexts. These are dynamically instantiated. The access to a dynamic object is only possible via a known reference. This reference may be acquired by reading a reference field of an object. However, the only way to create a new reference is by instantiating an object. There is no way for client code to otherwise make up a valid or invalid reference.

Within the DecentVM, static objects are e. g. Java class variables and code objects. A Java method must be able to access static objects without being passed a reference to the instance. Nevertheless, the VM itself has to take adequate actions, such as loading the class during the first access to a class variable, or the first object instantiation of this class. At this time, the VM creates the reference to the class variables, which afterwards must be globally available to all threads.

In a distributed system, dynamic and static objects require two different location and retrieval mechanisms, especially in face of object migration.

All nodes in the distributed system must be able to access all *static* objects. The DecentVM uses a combination of a distributed *singleton* mechanism together with a *distributed hash table* (DHT). Whenever a node accesses a static object for the first time, it computes the *hash value* of that object's bytecode and uses it as a key to index a DHT. If the key could not be found, the accessing node creates the static object and stores a reference in the DHT. The next node that makes a lookup with the key in the DHT retrieves the reference and can afterwards send its access requests directly to this node. To minimize the number of DHT lookups, the accessing node caches the reference for future use.

In contrast to static objects, which must be accessible by all nodes, typically only a small sub-set of nodes ever needs a reference to any given dynamic object. Therefore, the DHT is too expensive for dynamic objects because a DHT is a global data structure that cannot limit location updates to the sub-set of referencing nodes. Instead, the DecentVM offers two methods [7] to confine reference updates to these referencing nodes: one reactively updates the GAO location information, the other one updates proactively.

With the *reactive update approach*, a migrating GAO creates an intermediate *proxy* on its former home node. After the migration, this proxy forwards all subsequent accesses to the new location of the object. After an access along a single proxy or a chain of proxies, the accessing node is aware of the new location and does not need the proxy (chain) anymore. To prevent proxy chains from getting overly long and eventually exhausting the memory, each garbage collector run updates all invalid location information and afterwards removes all proxies.

Alternatively, a *proactive update approach* uses proxies and additional *incoming references*. These incoming references point to all nodes that hold a reference to the corresponding object. Whenever an object migrates, an update message is sent to all incoming references. After all update messages have been acknowledged, the proxy can be deleted immediately.

It depends on the application which of these two approaches performs better: On the one hand, the reactive update approach does not send update messages, but the access latency will get high if objects migrate frequently and accesses have to traverse a long chain of proxies. On the other hand, the proactive update approach might create a huge number of maintenance messages if many objects are referenced often. Here, the access latency is low because in most cases the access reaches the actual object directly, without having to traverse a chain of proxies.

C. Software Transactional Memory

STM is an alternative to lock-based synchronization. Instead of acquiring and releasing locks when accessing shared data, these accesses are placed in atomic blocks, i. e. ranges of code that run as a *transaction*. With respect to other nodes, STM makes the observable effects of a transaction – reading and writing shared data – appear as if the whole transaction was executed as a single, atomic instruction.

Our STM protocol mediates between GAOs and LOCs, and thereby between (logically) private and global memory. It works with GAOs and GAO versions. A GAO is only a virtual meta structure, whereas the GAO versions are the causally consistent representations of the GAO in the physical memory. So, the GAO determines the GAO version that represents its current value.

Each transaction stores all GAO versions that it created, along with other meta-data, in a so-called *transaction record*. The runtime does not delete transaction records when they become outdated, but retains them for some time as redundancy. Whenever a new transaction commits, our cost model determines which old transaction records to discard and which to keep. This obviates the need for checkpointing in most cases, and still allows the system to tolerate a certain rate of node failures.

III. The DecentVM on the SCC

One target platform of the DecentVM is Intel's *Single-Chip Cloud computer* (SCC), an experimental 48-core processor [8] that Intel Labs have created as a "concept vehicle" for many-core software research. The SCC contains 48 Pentium I cores on a single die. The cores are organized in a 6×4 array of 24 tiles, each hosting two cores. These tiles are connected in a 2D mesh network via 24 routers, which route messages between the different tiles via XY-routing. Each core owns a portion of the external memory as its private memory. All remaining memory is shared among the cores. There is no built-in mechanism to guarantee cache coherency for the shared memory. Consistency among the cores has to be implemented in software. This approach has the advantage that software coherency among multiple caches is dynamically reconfigurable. With this design, each application can define its own memory *domain*, in which the application can guarantee memory coherency.

A. SCC Programming

To support application developers, Intel offers the RCCE native message passing library [9]. RCCE offers a *basic* and a *gory* interface for memory management and an additional interface for power management.

The basic RCCE interface offers send and receive methods to pass messages between cores. It handles all communication details, whereas the gory RCCE interface provides low-level access to the communication layer.

Very recently, the beta version of the *privately owned public shared memory* (POP-SHM) [10] library was released. The POP-SHM library allows a core to allocate private memory and make it available to other cores, which also use the POP-SHM library. The POP-SHM library gathers all the publicly available memory and offers it as one contiguous address space to all cores that also use this library. Due to this feature, a reference to an object in the POP-SHM memory is the same on all cores.

B. The DecentVM on the SCC

The DecentVM is well suited for the SCC: The per-core L1 and L2 caches together with the external private memory represent the logically private memory, whereas the rest of the external shared memory adds up to the physically local memory that is shared by all DecentVM instances.

The new POP-SHM library directly supports the DecentVM on the SCC: Up to now, a committing STM transaction had to copy the LOC data from private memory to local memory to promote the changed LOCs to new globally visible GAO versions. The POP-SHM library simplifies this operation because the LOC can be allocated first in private POP-SHM memory. Only upon a successful commit, the VM makes the POP-SHM memory publicly available without having to copy it.

Thanks to these features, the DecentVM can easily run on a single SCC processor. Together with the described mechanisms for remote memory access via GUIDs and location information, the DecentVM is able to support not only a single SCC, but also a huge cluster of SCCs.

IV. Related Work

Several projects pursue similar objectives:

The Jessica research projects at the University of Hong Kong have been investigating distributed virtual machines since 1996 [11]. The original *Jessica* project developed a distributed Java Virtual Machine based on the Kaffe VM. It offers an SSI on top of the underlying hardware and supports transparent thread migration. *Jessica2* introduced a built-in global heap with lazy release consistency and simple object home migration, as well as a JIT-compilation mode [12]. *Jessica3* had the main objectives to overcome memory space limitations and to solve the problem of global thread scheduling [13]. *Jessica4* aims at new parallel programming paradigms, e.g. the PGAS programming model and transaction-based synchronization with two-way elastic atomic blocks (TWEAK).

Azul Systems Inc. [14] and Terracotta Inc. [15] both offer commercial solutions to distribute Java business applications in clusters.

Azul developed the *Zing* JVM that enables Java application servers to transparently offload threads to a pool of network attached worker nodes. Unlike our approach, Azul's VM follows the client/server paradigm: A single host machine handles class loading, native code execution, file I/O, and network communication. The system relies on efficient hardware support for thread synchronization, rather than on new consistency approaches on the VM level.

The Terracotta system modifies the Java bytecode and instruments it for the use in a global heap. The so modified code runs on unmodified Java VMs, and the application developer has to decide which objects to store in the global heap. Unlike our approach, Terracotta does not allow data and threads to seamlessly migrate between the machines in the cluster.

V. Conclusion

This paper has given a brief introduction into our current research on a fully decentralized runtime system, the DecentVM. The DecentVM is highly scalable and provides an SSI on top of a cluster of heterogeneous computing nodes.

We sketched two fully decentralized algorithms for locating and retrieving objects. The first algorithm relies only on proxies to forward object accesses, whereas the second algorithm features incoming references to proactively update location information. We also briefly introduced the other components of the DecentVM that are vital to scalability: our decentralized STM and our decentralized recovery mechanism.

Altogether, these mechanisms support the offered SSI, e. g. on top of the SCC many-core processor. Thus, developers can implement their applications without the need to deal with the SCC-specific details; they only sees the Java interface and used Java APIs. If multiple many-core processors are connected to form a compute cluster, the very same mechanisms allow the applications to extend across all the nodes in this cluster without any software modifications.

References

[1] R. Buyya, T. Cortes, and H. Jin, "Single System Image," *International Journal of High Performance Computing Applications*, vol. 15, no. 2, pp. 124–135, 2001.

[2] B. Saballus and T. Fuhrmann, "Maintaining Reference Graphs of Globally Accessible Objects in Fully Decentralized Distributed Systems," in *Proc. of the Int. ACM Symposium on High Performance Distributed Computing (HPDC'09)*, Munich, Germany, Jun. 11 – 13, 2009.

[3] A. Bieniusa and T. Fuhrmann, "Consistency in Hindsight, A Fully Decentralized STM Algorithm," in *Proc. of the IEEE Int. Symposium on Parallel Distributed Processing (IPDPS'10)*, Atlanta, Georgia, USA, Apr. 19 – 23, 2010.

[4] S.-A. Posselt, "Design of a Reliable, Fully Decentralized Software Transactional Memory Protocol," Diploma thesis, Technische Universität München, Munich, Germany, Aug. 2010.

[5] G. Taboada, J. Touriño, and R. Doallo, "Java for High Performance Computing: Assessment of Current Research and Practice," in *Proc. of the 7th Int. Conf. on Principles and Practice of Programming in Java (PPPJ'09)*, Calgary, Alberta, Canada, Aug. 27 – 28, 2009.

[6] A. Bieniusa, J. Eickhold, and T. Fuhrmann, "The Architecture of the DecentVM – Towards a Decentralized Virtual Machine for Many-Core Computing," in *Proc. of the 4th Workshop on Virtual Machines and Intermediate Languages (VMIL'10)*, Reno, Nevada, USA, Oct. 17, 2010.

[7] B. Saballus and T. Fuhrmann, "A Decentralized Object Location and Retrieval Algorithm for Distributed Runtime Environments," Technische Universität München, Munich, Germany, Tech. Rep. TUM-I1025, Dec. 2010.

[8] Intel Corporation, "The SCC Platform Overview, Rev. 0.7," May 2010.

[9] T. Mattson and R. van der Wijngaart, "RCCE: a Small Library for Many-Core Communication, Version 0.7," May 03, 2010.

[10] Intel Corporation, "The SccKit 1.4.0 User's Guide, Rev. 1.0," Mar. 2011.

[11] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau, "JESSICA: Java-Enabled Single-System-Image Computing Architecture," *Journal of Parallel and Distributed Computing*, vol. 60, no. 10, pp. 1194–1222, 2000.

[12] W. Zhu, C.-L. Wang, and F. Lau, "JESSICA2: a distributed Java Virtual Machine with transparent thread migration support," in *Proc. of the IEEE Int. Conf. on Cluster Computing (CLUSTER'02)*, Chicago, Illinois, USA, Sep. 23 – 26, 2002.

[13] K. Lam, Y. Luo, and C. Wang, "A Performance Study of Clustering Web Application Servers with Distributed JVM," in *Proc. of the 14th IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS'08)*, Melbourne, Australia, Dec. 08 – 10 2008.

[14] Azul Systems, Inc., "Zing Java Virtual Machine," http://www.azulsystems.com, last visited: May 05, 2011.

[15] A. Zilka and Terracotta Inc., *The Definitive Guide to Terracotta*. New York: Apress Media LLC, Mar. 2011.

Power and performance optimization through MPI supported dynamic voltage and frequency scaling

Florian Thoma[1], Michael Hübner[1], Diana Göhringer[2], Hasan Ümitcan Yilmaz[2], Jürgen Becker[1]
[1]*Karlsruhe Institute of Technology (KIT), Germany*
[2]*Fraunhofer IOSB, Ettlingen, Germany*
{florian.thoma, michael.huebner, becker}@kit.edu, {diana.goehringer, yilmaz}@iosb.fraunhofer.de

Abstract

The Intel Single Chip Cloud Computer (SCC) architecture offers the adjustment of voltage and frequency on individual islands in a certain range and in a certain dependability to each other. This possibility offers a high degree of freedom for workload balancing which can be done statically (at compile time) or dynamically (at run-time). Especially the latter topic, the dynamic voltage and frequency scaling (DVFS) is of high interest for novel multiprocessor systems, if they are deployed in energy efficient systems. It enables to provide computing performance on demand and therefore reduces power consumption. We envision to develop models, methods and cost functions for the DVFS in order to optimize the workload balance of all processor cores at run-time on the SCC.

Keywords: DVFS, Dynamic MPI, workload balancing

1. Introduction

The Intel Single Chip Could Computer (SCC) architecture offers the adjustment of voltage and frequency on individual islands (consisting of 24 frequency and 7 voltage domains) in a certain range and in certain dependability to each other. This possibility offers a high degree of freedom for workload balancing which can be done statically (at compile time) or dynamically (at run-time). Especially the latter topic, the dynamic voltage and frequency scaling (DVFS) is of high interest for novel multiprocessor systems, if they are deployed in energy efficient systems. It enables to provide computing performance on demand and therefore reduces power consumption. We envision to develop models, methods and cost functions for DVFS in order to optimize the workload balance of all processor cores at run-time on the SCC. For this purpose, novel modules for MPI will be developed in order to support the programmer through an autonomous performance / power consumption management. Furthermore, the modules which include the methods and cost functions for the DVFS avoid critical constellations of voltage and frequency which might lead to damage on the SCC. The challenging research topic delivers an extension to the MPI-based programming model through the support of DVFS on the SCC and closes therefore a gap for a novel MPSoC programming methodology. As a result, a reduced power consumption and simultaneously optimized performance increases the attractiveness of the SCC in a variety of application scenario and might open new markets for this and future multiprocessor systems. Especially the embedded high performance applications such as image processing for surveillance of rooms e.g. in an airport, benefit from a dynamic control of the MPSoC architecture in order to optimize power consumption tailored to a current situation. The research, which is proposed, follows the trend of cyber-physical systems, where a close control loop is used to optimize the system characteristic (see [1]). The benefit of the novel method will be measured under real conditions (a realistic application scenario) and compared to the traditional realization. All hardware related methods lead to a novel paradigm called Dynamic MPI. The SCC architecture with its MPI (Message Passing Interface) API is an excellent platform for studying and evaluating new programming paradigms. The research objective of this proposal is to define and implement a novel programming standard together with a designflow for efficiently programming heterogeneous and homogeneous multiprocessor systems. The idea is to extend the existing MPI standard with dynamic aspects. Out of this results the new programming paradigm: Dynamic MPI.

2. Relationship to other research at KIT and IOSB and previous work

The ITIV is working successfully on run-time adaptive systems over 9 years. In the area of reconfigurable hardware the group counts to the leading researchers in the world. The ITIV was responsible for the realization of the network-on-chip, the on-chip integration of the reconfigurable tiles and the run-time system support for the MORPHEUS chip (see [2]). Within this project, funded by the EU, the novel concepts of dynamic workload balancing were investigated successfully for this multicore chip. The proposed research with the SCC differs from the previous work besides MORPHEUS, several DFG funded projects and bilateral research projects in that way, that a fully new concept of a homogeneous processor array which follows the programming model MPI can be used to optimize the performance and power consumption of the MPSoC. The project enables to gain the experience in run-time adaptive systems and enable to develop a standard programming model through an extension of MPI which is then available for the community.

To the best of our knowledge there exists so far no similar research on Dynamic MPI and its designflow as proposed here. The most similar approach is our own designflow (see [3]) from RAMPSoC, which supports the MPI-standard. RAMPSoC is a heterogeneous FPGA-based MPSoC consisting of a variable number of processors closely coupled with hardware accelerators. RAMPSoC exploits dynamic and partial reconfiguration to adapt the hardware structure of the MPSoC at runtime. For the runtime management of RAMPSoC a special purpose operating system called CAP-OS was developed. The designflow so far does not handle the dynamic aspects, which are planned for the here proposed Dynamic MPI. The research strengths at IOSB are the development of the architecture, the designflow and the runtime management system for heterogeneous multiprocessors based on FPGAs. Furthermore, a great research strength of our institute is the development of image processing algorithms, especially as the above mentioned object detection and tracking algorithms.

3. Impact of the research

The result of this research is an extension of the MPI programming model for the SCC chip. It is envisioned to gain the attractiveness of the SCC for a wide area of applications. Since power and performance can be optimized simultaneously through DVFS, but this design space is huge and practically not to handle by the designer, the programming model, the modules and the autonomous methods enable to handle this powerful feature of the new SCC chip. It is envisioned to reduce the power consumption by up to 50% in comparison to an application which is realized with traditional MPI. The application will be selected out of a scenario for image processing in embedded systems (a focus is here homeland security with surveillance cameras). The results will be delivered as an extension for MPI and distributed to the community together with Intel. Furthermore, the output of the research gives a feedback to Intel in terms of the requirements for DVFS. This means that the requirements of the granularity for the adjustment of voltage and frequency were evaluated. A result could be that less stages for the adjustment are sufficient for optimizing the system characteristic which leads to an decrease of chip area and therefore reduced costs. Furthermore, it is planned to extend and create a new programming standard for homogeneous as well as heterogeneous MPSoCs called Dynamic MPI together with a corresponding designflow. With these results, we want to provide a first solution for efficiently programming the SCC architecture and also existing and future architectures consisting of heterogeneous or homogeneous processing elements. Especially, we focus on the modeling of dynamic aspects, e.g. workload balancing depending on inputs from the environment within our programming paradigm. So far, to the best of our knowledge, these dynamic aspects are not handled by any programming paradigm. The envisioned extension of MPI enables to exploit the benefit of a dynamic allocation of processes on an MPSoC through standardized methods included in the MPI library.

4. Preliminary results

In order to explore the performance and power consumption tradeoff of the SCC two image processing applications have been selected and programmed using RCKMPI. The overall execution time and power consumption have been measured by varying the number of processing elements and their clock frequency.

4.1. Image processing applications

The first image processing algorithm is the sobel operator. This is an edge detection algorithm, which calculates the gradient magnitude at each point in a grey level image. It is often used to separate between objects and their background. The operator works with two convolution masks, one for horizontal changes and one for vertical ones. These masks slide over the image and compute the two derivative images. However, the sobel operator is a direction-independent edge detector, so the two results are normed (by the Pythagoras Theorem) and a direction-independent image is created. The characteristics of the sobel algorithm are many multiplications, additions and also square root operations.

The second image processing algorithm is a thinning algorithm. These algorithms are used to reduce two-dimensional objects to single pixel wide branches by removing pixels inside the object shape according to some criteria, but they do not shorten or break the object shape apart. Here a thinning algorithm, developed by Z. Guo and W. R. Hall [6], is used. This algorithm uses two subiterations, the first subiteration deletes the north and east and the second deletes the south and west outline points of the shape. For more information about this algorithm see [6] and [7]. The characteristics of this algorithm are multiple compare and jump operations.

Figure 1 shows the input image used for the exploration and the result images of the Sobel and the thinning algorithm.

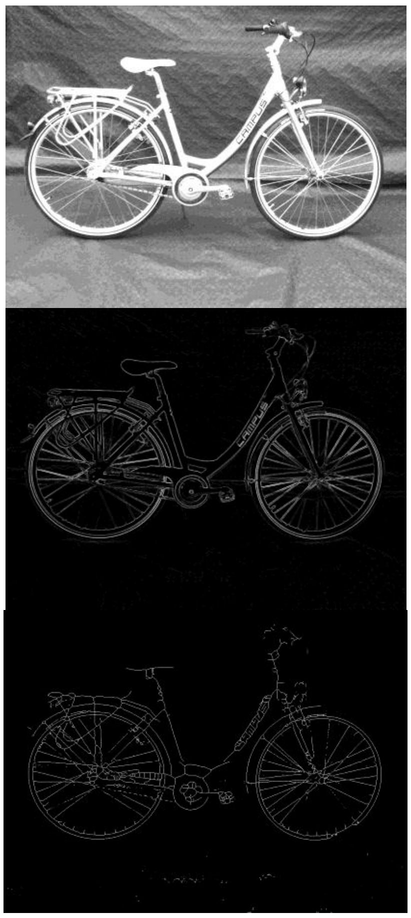


Figure 1. a) input image, b) Sobel output, c) thinning output

Figure 2 shows the number of required clock cycles for the Sobel algorithm. With the increase of the number of cores it can be seen, that the effect of clock cycle reduction comes to saturation. A number of 24 cores is for this implementation of the Sobel algorithm the best choice. Certainly, algorithm and communication optimization can lead to slightly other results.

A other effect can be found with the thinning algorithm where a nearly linear speedup can be found. Therefore the maximum number of cores used to perform this application is of high benefit.

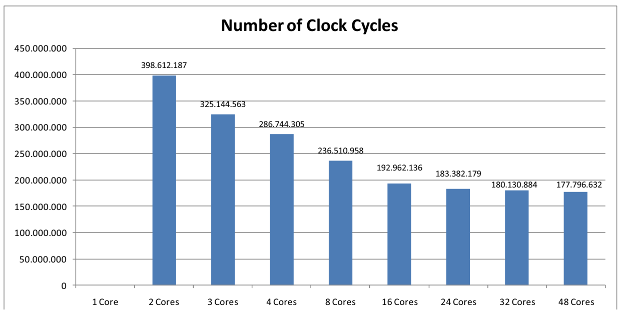


Figure 2. Execution time of the Sobel algorithm by varying the number of cores

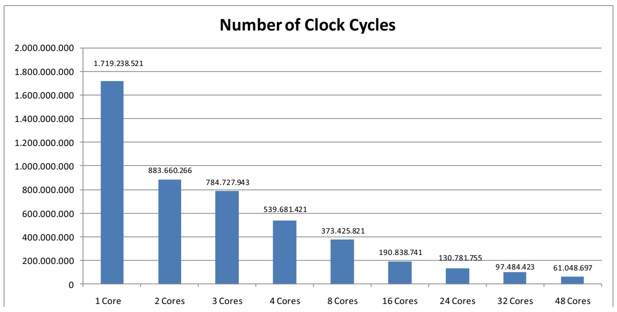


Figure 3. Execution time of the Thinning algorithm by varying the number of cores.

5. Conclusions and outlook

Currently the workgroups work in parallel on hardware related topics like the scaling of the voltage and frequency scaling mechanisms and the extension of the MPI standard as well as required virtualization techniques (see [4]). Here MPI methods for an FPGA-based MPSoC are presented. The groups intend to extract from this work the experience for the SCC related MPI extensions. Furthermore, attractive application scenarios from image processing, bioinformatics as well as from simulation acceleration are the use cases (see [5]).

Next steps of this research work is to include the realized MPI extensions and the control loops for the physical adjustment of voltage and frequency in more application scenarios in order to measure the impact and derive important parameter sets for the control loop equations. Currently an interdisciplinary discussion with experts from control engineering leads to a promising very novel technical experience which can lead to a fully new control mechanism for the SCC and SCC like architectures.

6. References

[1] Becker, Huebner: "Multiprocessor System-on-Chip - Hardware Design and Tool Integration", 1st Edition. 2010, Springer US ISBN 978-1-4419-6459-5

[2] Voros, Rosti, Huebner: "Dynamic System Reconfiguration in Heterogeneous Platforms: The MORPHEUS Approach", ISBN: 9789048124275

[3] D. Göhringer, M. Hübner, M. Benz, J. Becker: "A Design Methodology for Application Partitioning and Architecture Development of Reconfigurable Multiprocessor Systems-on-Chip"; In Proc. of the 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines" (FCCM 2010), Charlotte, USA, May, 2010

[4] D. Göhringer et al. :"RAMPSoCVM: Runtime support and hardware virtualization for a runtime adaptive MPSoC", accepted for the FPL 2011, Chania, Greece

[5] C. Roth et al. "Flexible and Efficient Co-Simulation of Networked Embedded Devices", SBCCI 2011, Brazil

[6] Z. Guo, W.R. Hall: "Parallel Thinning with two-Subiteration Algorithm", CACM, March 1989, vol. 1.32, no.3, pp.359-373.

[7] Y.Y. Zhang, P.S.P. Wang: "A parallel thinning algorithm with two-subiteration that generates one-pixel-wide skeletons", Pattern Recognition, 1996, Proceeding of the 13th International Conference, 1996, vol.4, pp: 457 – 461.

Recent Advances and Future Prospects in iRCCE and SCC-MPICH — Poster Abstract —

Carsten Clauss, Stefan Lankes, Pablo Reble, Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
{clauss,lankes,reble,bemmerl}@lfbs.rwth-aachen.de

***Abstract*—The Single-Chip Cloud Computer (SCC) experimental processor [4] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. Intel provides a customized programming library for the SCC, called RCCE [5], that allows for fast message-passing between the cores. For that purpose, RCCE offers an application programming interface (API) with a semantics that is derived from the well-established MPI standard [7]. However, while the MPI standard offers a very broad range of functions, the RCCE API is consciously kept small [6] and far from implementing all the features of the MPI standard. So, for example, RCCE only provides *blocking* (often also referred to as *synchronous*) send and receive functions, whereas the MPI standard also defines the semantics of *non-blocking* communication functions. For this reason, we have started to extend RCCE by new communication capabilities, as for example by the ability to pass messages *asynchronously*. In doing so, we aim to avoid interfering with the original RCCE library and therefore we have placed our extensions and improvements into an additional library called iRCCE [2]. Moreover, this additional library in turn serves us as low-level communication layer for SCC-MPICH, that is an SCC-customized and full MPI-1 compliant MPI library. In this contribution, we present the recent advances and future prospects for both these SCC-related communication libraries: iRCCE and SCC-MPICH.**

***Keywords*—Many-core, Message-Passing, SCC, RCCE, MPI**

I. iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library

Due to the lack of non-blocking communication functions within the current RCCE library, we have started to extend RCCE by such asynchronous communication capabilities (`iRCCE_isend`/`iRCCE_irecv`). In doing so, we aim to avoid interfering with the original RCCE functions and therefore we have placed our extensions into an additional library with a separated namespace called iRCCE. An obvious way to realize non-blocking communication functions would be to use an additional thread that processes the communication in background. Although this approach seems to be quite convenient, it is not applicable in *bare-metal* environments where a program runs without any operating system and thread support. And since RCCE has been designed to support also such bare-metal environments, we had to waive this thread-based approach for realizing non-blocking functions. Therefore, we have followed another approach where the application must drive on the communication progress by itself. For this purpose, the non-blocking communication functions return *request handles* which can then be used by the application to trigger the progress by means of additional *push*, *test* or *wait* functions (`iRCCE_push`, `iRCCE_test`, `iRCCE_wait`). [2]

A recent improvement of iRCCE is the feature that one can use a wildcard (`iRCCE_ANY_SOURCE`) instead of a definite source rank when calling the receive function. That means that this wildcard can be used to receive *any* incoming message regardless from its actual sender. However, the application programmer still has to ensure that at least the stated message length matches between receiver and sender.

Currently, we are developing a mailbox system on top of iRCCE that can be used to exchange small (cache-line-sized) datagrams between the cores. Since this mailbox system works without interference with the common send and receive functions, it can be used to pass additional signaling information alongside with normal RCCE/iRCCE messages. Therefore, such a mailbox datagram is well structured in terms of data items that are quite similar to that of message headers: `source`, `size`, `tag` and embedded `payload`.

Our aim is to use this mailbox system to extend the current semantics of the send and receive functions. So, for example, we plan to introduce a further wildcard mechanism also for the message length (`iRCCE_ANY_LENGTH`). That means that the information about the actual message size has then just to be provided by the sender, while the receiver merely has to ensure that the receive buffer is large enough to store the message. Moreover, by introducing additional message *tags*, as known from the MPI standard, even a message prioritization and reordering by means of these tags would become possible.

For this purpose, a sender would initially post a mailbox datagram to the respective receiver, indicating that a payload message of a certain size and with a certain tag will follow. Therefore, the local mailbox on the receiver side needs to be checked frequently in order to detect such incoming messages. However, it is entirely possible that the receiver detects a message that is yet still unexpected. This is for example the case when the message tags on sender and receiver side do not yet match and thus a message reordering becomes necessary. In such a case, the receiver can either copy the incoming message into a temporary buffer or the receiving of the actual payload must be delayed by sending a corresponding response datagram. The choice for one of these two approaches depends on the message size: for short and midsize messages, a temporary buffering seems to be acceptable, whereas long messages should be delayed because the additional copy procedure would impact the communication performance. Besides this, very small messages could be embedded *into* a datagram, so that there is no need for an additional payload message in such a case.

II. SCC-MPICH: Yet Another MPI-compliant Message-Passing Library for the Intel SCC

Although the semantics of RCCE's communication functions are obviously derived from the MPI standard, the RCCE API is far from implementing all MPI-related features. And even though iRCCE extends the range of supported functions (and thus the provided communication semantics), a lot of users are familiar with MPI and hence want to use its well-known functions also on the SCC. A very simple way to use MPI functions on the SCC is just to port an existing TCP/IP-capable MPI library to this new target platform. However, since the TCP/IP driver of the Linux operating system image for the SCC does not utilize the fast on-die message-passing buffers (MPBs), the achievable communication performance of such a ported TCP/IP-based MPI library lags far behind the MPB-based communication performance of RCCE and iRCCE.

For this reason, we have started in the last year to implement an SCC-optimized MPI library, called SCC-MPICH, which in turn is based upon our iRCCE extensions of the original RCCE communication library. At about the same time, Intel also started to implement an SCC-customized MPI library, called RCKMPI [8]. While RCKMPI has already been released by Intel, we have not yet published SCC-MPICH despite the fact that it is already fully operational, too. The reason for this is that we think that our human resources are too limited to provide sufficient user support for this project in case of an official software release. However, we use SCC-MPICH as basis for our future message-passing related research on the SCC and we have launched several student projects that in turn are also based on SCC-MPICH.

A major advantage of SCC-MPICH compared to RCKMPI is that it can be installed and used as easy as RCCE. That means that one can use the `mpirun` script just instead of the known `rccerun` directly from the Management Console PC (MCPC) without installing any additional libraries or startup environments on the SCC cores. Moreover, even the cores of the MCPC can easily be involved into an SCC-MPICH session. That means that one can start x MPI processes on the SCC cores and additionally y MPI processes on the cores of the MCPC.

In doing so, SCC-MPICH does not use just TCP/IP (as the lowest common dominator) for all the communication, but rather offers *hierarchy-awareness* in such a way that always the fastest communication mode is being used. That means that MPI processes running on the SCC cores use the message-passing buffers (MPBs) to communicate among each other, while processes running on the MCPC communicate via shared memory. The communication between the MCPC processes and the SCC cores is then eventually conducted via TCP/IP. In order to start such a mixed MPI session, one just needs to issue `mpirun -nue x -mcpc y` in a console on the MCPC.

A further advantage of SCC-MPICH is that it offers SCC-optimized collective communication routines. This is because SCC-MPICH is directly based upon RCCE (with iRCCE extensions) and due to the fact that RCCE can in turn be extended by the customized collective functions of the so-called *RCCE_comm* library [1]. That way, an easy mapping of MPI collective function calls onto the optimized RCCE_comm functions becomes possible

A more detailed description of SCC-MPICH together with some performance results can be found in [3].

References

[1] Ernie Chan. *RCCE_comm: a Collective Library for the Intel Single-chip Cloud Computer.* Intel Corporation, September 2010.

[2] C. Clauss, S. Lankes, J. Galowicz, and T. Bemmerl. *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer.* Chair for Operating Systems, RWTH Aachen University, December 2010. Users' Guide and API Manual.

[3] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor (accepted for publication). In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011) – to appear*, Istanbul, Turkey, July 2011. accepted for publication.

[4] Intel Corporation. *SCC External Architecture Specification (EAS)*, July 2010. Revision 0.98.

[5] T. Mattson and R. van der Wijngaart. *RCCE: a Small Library for Many-Core Communication.* Intel Corporation, May 2010. Software 1.0-release.

[6] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: The Programmer's View. In *Proceedings of the 2010 ACM/IEEE Conference on Supercomputing (SC10)*, New Orleans, LA, USA, November 2010.

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* High-Performance Computing Center Stuttgart (HLRS), September 2009. Version 2.2.

[8] Isaias A. Compres Urena. *RCKMPI User Manual.* Intel Braunschweig, January 2011.

Exploring Database Workloads on Future Clustered Many-Core Architectures

Panayiotis Petrides
Department of Computer Science
University of Cyprus
Email: csp7pp5@cs.ucy.ac.cy

Andreas Diavastos
Department of Computer Science
University of Cyprus
Email: cs06da1@cs.ucy.ac.cy

Pedro Trancoso
Department of Computer Science
University of Cyprus
Email: pedro@cs.ucy.ac.cy

***Abstract*—Decision Support System (DSS) workloads are known to be one of the most time-consuming database workloads that process large data sets. Traditionally, DSS queries have been accelerated using large-scale multiprocessor. In this work we analyze the benefits of using future many-core architectures, more specifically on-chip clustered many-core architectures, for such workloads for accelerating DSS query execution and study their performance behavior. To achieve this goal we propose data-parallel versions of the original database scan and join algorithms. In our experiments we study the behavior of three queries from the standard DSS benchmark TPC-H executing on the Intel Single Chip Cloud Computing experimental processor (Intel SCC). The results show that parallelism can be well exploited by such architectures and also how the computational workload compared to the data size of each executed query can influence performance. Our results show linear scalability for queries where the computation to data size ratio is balanced.**

I. Introduction

The *de-facto* standard in processor design is the multi-core architecture. This architecture offers the benefit of an increased degree of parallelism to provide better performance, without the drawbacks of previous monolithic designs, such as high power consumption and complex design. As technology improves, the integration level increases leading to an increase in the number of cores per chip. While this results in a further increase of the degree of parallelism, it may not necessarily lead to improved performance, even when considering highly parallel applications. The increasing number of processing units per chip results in a higher demand for “feeding” those units with both instructions and data. At the same time, neither the number of pins on the chip, nor the links to memory improve at the same rate as the number of cores. Moreover the complexity of the interconnection network of large scale multi-core architectures increases with the number of cores, the above mentioned multi-core issues result in limiting the scalability in terms of number or cores of these architectures. The proposed large scale many-core architecture by Intel, also known as the Intel Single Chip Cloud Computing experimental processor (Intel SCC) [1] addresses the above limitations.

Database applications are of the most demanding workloads. More specifically, Decision Support Systems (DSS) database applications combines the processing of large data sets along with the computation of statistical information extracted from data. The purpose of this paper is first to understand the benefits of the use of a future clustered many-core architecture, like Intel’s Single Chip Cloud Computing experimental processor [1], in a large scale data center which handles DSS applications.

In order to achieve our goal we have analyzed the performance of the basic database algorithms parallelized using the available toolchain of the Intel SCC. The algorithms are the basis for the execution of standard representatives DSS queries taken from the TPC-H benchmark suite [3].

The Single-Chip Cloud Computer (SCC) experimental processor [1] is a 48-core concept vehicle created by Intel Labs as a platform for many-core software research. Intel SCC processor implements on-chip clustered many-core architecture where cores are organized in tiles of 2 cores, 24 tiles in total, and the communication is based on a 2D mesh interconnection network. This modern architecture includes 48 Intel Pentium cores (P54C architecture) that are served by four on-chip memory controllers. The memory controllers use the technology DDR3-800 and -1066 speed grades.

II. Mapping Data-Parallel Database Queries to Intel SCC experimental processor

Queries submitted for execution in a Database Management System (DBMS) are described in a high-level language (SQL) where four basic operations can be performed: *Scan*, *Join*, *Order* and *Aggregate*. The execution of these operations is done by a set of algorithms provided by the DBMS.

For example, *Scan* can be performed using the *Sequential Scan* or the *Index Scan* algorithm. The DBMS *Query Optimizer* decides which algorithm will be used for the submitted operations in order to achieve the best execution time.

We have formatted the processing data as data streams resembling data arrays from regular high-level languages. For the purpose of our work the data are stored column-wise, *i.e.* all values of a particular attribute belonging to different records, are stored in the same *data stream*. More details about the algorithms used and how data are mapped can be found in [2]. Let’s consider a table (Table A) which is composed of records containing 3 attributes: attr1, attr2, and attr3. For each attribute a new stream is created to store all data as is shown in Fig 1 (b) instead of the traditional way to store the data as is shown in Fig 1 (a).

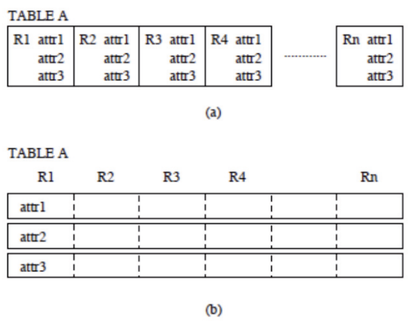


Fig. 1. Table A: (a) logical and (b) physical data organization.

Given the data layout as presented above, for this work, we use the simple sequential scan algorithm as to exploit both load balancing and locality while traversing the data. In our algorithm, all records are traversed and the record's attributes are checked against a certain condition. If the condition is satisfied then the record is copied to the result stream. The condition may be a simple attribute comparison or a complex boolean function. We have mapped this operation to the Intel SCC by implementing the condition to be tested and by sending to each core the input parameters which are the data streams that are used to evaluate the scan condition.

The data-parallel nested loop Join is also implemented in order to exploit the streaming model. More specifically if we want to join Table A with key *ka* and Table B with foreign key *ka*, then the key value over the foreign key are compared. Join operation is performed by checking a certain key value from one table against all the key values of the other table. The checks may be performed in a loop and the results that satisfy the checks are then passed to the next condition.

III. Experimental Setup

For this work we have used the Intel Single Chip Cloud Computing (Intel SCC) experimental processor, RockyLake version. The operating system used for the Intel SCC cores is the default Linux kernel provided by the RCCE SCC Kit 1.3.0. The host PC, responsible for controlling the applications execution on the Intel SCC processor, is configured with Intel Core i7 processor 3.7GHz and 4GB memory. The connection of the host PC to the Intel SCC is through a PCIe Expansion Card and a PCIe x4 Cable. For porting and executing the applications on the SCC we have used RCCE 1.3.0 toolchain.

For our work we focused on the execution of the basic database algorithms and their parallelization. As such, the queries analyzed in this work were implemented as programs that executed the operations determined by the queries and their results were validated. We have ported 3 different queries from TPC-H benchmark suite of different complexity and demands. More specifically we have ported Queries 3, 6 and 12, from now on referenced as Q3, Q6 and Q12. Different input sizes were used for our evaluation in order to study their performance scalability. The input data sets were generated using the dbgen tool. The input sizes and the number of tables used for each query execution are depicted in Table I.

```
select
      sum(l_extendedprice*l_discount)as revenue
from
      lineitem
where
      l_shipdate  >= date `[DATE]'
      and l_shipdate < date `[DATE] + interval `1' year
      and l_orderkey = o_rderkey
      and l_discount between [DISCOUNT] - 0.01 and + 0.01
      and l_quantity < [QUANTITY];
```

Fig. 2. TPC-H Q6. The parameters used were: DATE=2005, DISCOUNT=10, and QUANTITY= 1000000.

```
select
      sum(case when o_orderpriority = '1-URGENT' or
      o_orderpriority = `2-HIGH'
         then 1
         else 0 end) as high_line_count
from
      orders, lineitem
where
      o_rderkey = l_orderkey
      and l_shipmode in (`[SHIPMODE1]', `[SHIPMODE1]')
      and l_commitdate < l_receiptdate
      and l_shipdate < l_commitdate
      and l_receiptdate > date `[DATE]'
      and l_receiptdate > date `[DATE]' + interval '1' year;
```

Fig. 3. Simplified version of TPC-H Q12. The parameters used were: SHIPMODE1=1, SHIPMODE2=2 and DATE=2009.

```
select
      l_orderkey,
from
      customer, orders, lineitem
where
      c_mktsegment = `[SEGMENT]'
      and c_custkey = o_custkey
      and l_orderkey = o_rderkey
      and o_orderdate < date `[DATE]'
      and l_shipdate > date `[DATE]';
```

Fig. 4. Simplified version of TPC-H Q3. The parameters used were: DATE=2007 and SEGMENT=3.

IV. Results

We have monitored the execution of the three queries scaling them from 1 to 48 cores on the Intel SCC processor.

The first investigation for our workloads was to monitor the time taken for reading the input data to the different cores for the different input sizes. The results were obtained by measuring the time taken for all cores to read the same amount of input data simultaneously, i.e. creating a copy of the input data locally at each core. It is important to notice that for

TABLE I
TPC-H QUERIES INPUT SIZE.

Query	Tables	Input Size 0.01	Input Size 0.1
Q3	3	4.24MB	93.56MB
Q6	1	3.71MB	74.24MB
Q12	2	3.74MB	91.14MB

both input sizes the time taken to read input data is stable and does not show high deviation between the different cores neither when the number of cores is increased. This can be explained from the high bandwidth available to the cores from the network-on-chip and the integrated memory controllers.

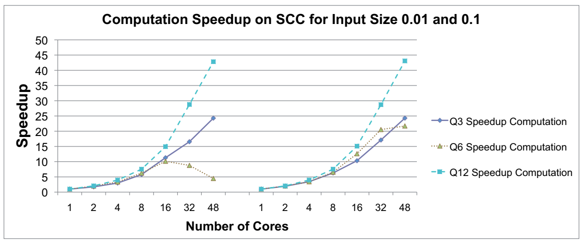


Fig. 5. Computation Speedup for TPC-H Queries 3, 6 and 12 for input size 0.01 and 0.1.

Secondly we wanted to investigate the scalability of the algorithms in the section where computation is done. In Figure 5 we can observe the speedup results for the three queries for the two different input data, 0.01 and 0.1, for the computational part of each algorithm. From our results we can observe the good speedup scalability of Q12. For Q6 with input data size 0.01, we can observe that the speedup reaches a maximum of 10x for 16 cores. For 32 and 48 cores we can observe a degradation of the speedup. This is caused due to the low computational complexity and the small input data set of Q6. Even though in scale factor 0.1 the workload increases significantly we can also observe that the speedup does not increase linearly from 32 to 48 cores, but instead it remains stable. This can also be explained due to the low computation complexity of the specific query. For Q3 we can observe a performance improvement for both input data sizes as the number of cores increases, although not in a linear way. This is caused by the high computation complexity in contrast to the other queries which results in slow performance increase as the number of cores increases.

In Figure 6, we show the total speedup for the three queries including both the time for reading the input data and the computational time until the completion of the queries. We can observe from our results that even though we can achieve a relatively good speedup scalability for Q6 in terms of computation, when it comes to the total time spend for the execution of the specific query the results are dominated by the time taken for reading the input data.

Q12 offers very good scalability since it combines well the data transfers and computation resulting to almost linear

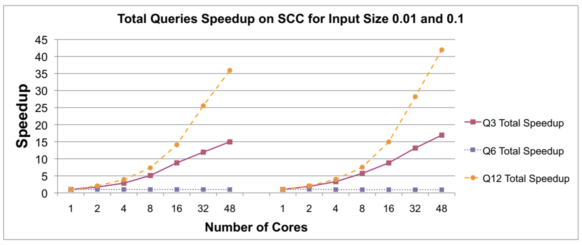


Fig. 6. Total Speedup for TPC-H Queries 3, 6 and 12 for input size 0.01 and 0.1.

speedup for both input sizes, up to 36x and 42x respectively for the two input sizes for 48 cores. Q3, which has the highest complexity of the three queries, offers relatively good speedup but lower compared to the Q12 for the two different input sizes, 15x and 17x respectively for the 48 core setup. Even though there is a similar behavior to the other queries, the impact to the performance is the computational part of the query. As described earlier, this query makes a join of three tables and consequently this can impact performance due to data transfers from the memory to the local cache of the cores and/or conflict misses to the local cache from the different data. These factors affect performance of Q3 even though the performance improves as the number of cores is increased.

V. Conclusion

From our experiments we have observed that for queries algorithms we have different performance behavior. In order to achieve good performance the algorithms' complexity and input data ratio must be well balanced otherwise the performance is dominated by the data transfers. More specifically, our results shows that a medium complexity algorithm with large input data set, Q12, achieves linear speedup up to 42x for 48 cores setup in contrast to low complexity algorithms, Q6, which are dominated from the data transfers. Our future work will be focused on concurrently execute the different queries on the Intel SCC. The target of our future work is to determine how we can split the Intel SCC resources for the most optimal execution of the different queries. Also we plan to investigate how we can overcome the data transfers overhead in order to improve the performance of algorithms that are dominated by them.

Acknowledgment

The authors would like to thank Intel Labs for lending the Intel SCC research processor.

References

[1] J. Howard, et al, *A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS*, In Proceedings of the International Solid-State Circuits Conference, Feb 2010.

[2] P. Trancoso and D. Othonos and A. Artemiou, *Data parallel acceleration of decision support queries using Cell/BE and GPUs*, In Proceedings of the 6th ACM conference on Computing frontiers (CF'09), pages 117-126, 2009.

[3] Transaction Processing Council, *TPC Benchmark H (Decision Support) Standard Specification Revision 2.6.1*, June 2006.

[4] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, *Gputerasort: High performance graphics o-processor sorting for large database management*, In SIGMOD 06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pages 325336, 2006.

A Fast Fourier Transformation Algorithm for Single-Chip Cloud Computers Using RCCE

Wasuwee Sodsong and Bernd Burgstaller
Department of Computer Science
Yonsei University
Seoul, Korea

***Abstract*— Multimedia applications, spectrum analyses and data compression algorithms employ Fourier transformations as one of their main components to transform series from the time to the spectral domain and vice versa. Effective fast Fourier transformation (FFT) algorithms imply performance enhancements in related applications. Although architecture-specific programs achieve better performance, many FFT implementations were designed to be hardware independent and unaware of the underlining architecture. In this paper we introduce a novel FFT algorithm based on the RCCE native message passing library for the single-chip cloud computer (SCC). We parallelized the recursive (divide-and-conquer) radix-2 FFT such that the inputs for all processing units are independent. Private memories are used to avoid cache coherence issues, and the algorithm was designed to minimize the message passing overhead. Preliminary experimental results were conducted using the RCCE emulator on an Intel Xeon 2 CPU quad-core computer. The emulator results showed promising scalability and speed-ups over the sequential implementation. Based on hardware availability, we plan to run the experiments on real SCC hardware for the final version of this paper.**

Keywords-component; Fourier transform; Single-Chip Cloud Computer; Radix-2 DIT FFT; RCCE;

I. Introduction

Fourier analysis is widely used across a large number of applications involving time series and waveform analysis [6], including audio compression, image processing, and spectrum analysis. Many sequential and parallel FFT sources are currently available, as well as architecture-specific FFTs such as FFTW for the Cell B/E architecture [9, 13]. However, to the best of our knowledge, an application for the SCC processor has not yet been devised.

Intel's SCC consists of 48 cores arranged in a grid of 4x6 tiles with two cores per tile [1, 2, 7]. Communication costs between cores on the same tile are invariably smaller than communication costs between cores farther away on the mesh. Architecture-independent FFT algorithms cannot minimize those communication overheads leading to degraded overall performance. Besides communication-aware mapping of processes to cores, the choice of memory also affects the performance of the SCC. The SCC architecture does not support cache coherence between cores [3]. The SCC memory hierarchy comprises three types: off-chip private memory, off-chip shared memory and on-chip shared memory. The characteristics of those memory-types are discussed in Section II. This paper focuses on SCC properties provided through the RCCE library to develop parallel FFT algorithms.

The remainder of this paper is organized as follows. Section II describes background material on SCCs, RCCE and Fourier analysis. Section III discusses different approaches on FFT algorithms. Section IV contains our experimental evaluation; we draw our conclusions and outline future work in Section V.

II. Background

A. *Single-Chip Cloud Computer (SCC)*

The SCC architecture was introduced by Intel's Tera-Scale research team based on the idea of implementing a cloud computer system on a single die. The architecture contains a mesh of 24 tiles with two cores and one router per tile [2]. Each core consists of 16KB L1 cache, 256KB L2 cache and 8 KB-SRAM Message Passing Buffer (MPB). By default, 64 GB off-chip DRAM is divided into 48 private memory regions, plus a memory-region shared by all cores. The off-chip private memory, served as a local memory, is accessible by only one core. Hence, normal cache rules are applied. Data is cached through L1 and L2 accordingly. On the other hand, depending on user's specifications, shared memory can be set up as cacheable and non-cacheable memory [3]. If cacheable shared memory is used, cache coherence must be explicitly handled by the user programs. In addition to DRAM, the SCC provides the MPB which consists of fast on-chip SRAM shared memory. The MPB is mainly utilized for message passing purposes. Although RCCE logically partitions the MPB for each core, the MPB is accessible by any core [12].

Each type of memory was designed to serve different purposes. The RCCE library provides users an ability to manage memory allocations. Applied to FFT, frequently used variables can be stored in the off-chip shared memory. However, accessing off-chip memory consumes many clock-cycles compared to accessing the MPB. Alternatively, the variables can be stored in the private memory. Private memory is cached by L1 and L2 caches, but because private memory is exclusive to a core, message passing must be used for communication between cores. A *hosts* file in RCCE contains the number of cores in the system in descending rank order. Since FFT algorithms have a fixed data flow and communication pattern, an optimal partitioning of processes

Research partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0005234), and the OKAWA Foundation Research Grant (2009).

onto cores can be achieved through the hosts file. Hence, in this paper we applied private memory.

B. *Fast Fourier Transform (FFT)*

In 1965, Cooley and Tukey [10] introduced an algorithm commonly known as the fast Fourier transform, which effectively computes a DFT in $\Theta(n \log n)$. Several FFT algorithms exist. In this paper, we use one-dimensional radix-2 FFT. An FFT equation of length N is rearranged as the sum of the following two parts: the DFT of even indices and the DFT of odd indices [6] as shown in Equation 1. The reduced equations can be recursively broken down into two half-size DFT problems.

$$X(k) = DFT_{N/2}[x(0),x(2),\ldots,x(N-2)] + W_N^k * DFT_{N/2}[x(1),x(3),\ldots,x(N-1)] \quad (1)$$

Each small DFT takes different inputs, which makes it suitable for parallel programming.

III. APPROACHES

Our SCC FFT algorithms are based on radix-2 FFT. As follows from Equation (1), inputs of sub-DFTs are independent of each other; hence, each computation can be easily mapped onto a different core. We develop two parallel FFT algorithms based on the RCCE library:

1. a synchronous message passing FFT algorithm, and
2. a pipelined one-sided message passing FFT algorithm.

The first algorithm uses synchronous message passing based on recursive FFT. The algorithm comprises three steps: initialization, private FFT and combining of results. Using P units of execution (UE), the top-ranked UE loads input sequences of length N and distributes N/P elements to all working UEs in the initial step. Process i internally performs a DFT on x[i, i+P, i+2P, … , N-P+i]. After this computation, half of the active cores pass their results to the other half for further DFT and combining of results. As shown in Figure 1, UE0 must combine results from UE2 and UE1 in respective order. To guarantee this order, synchronous message passing is chosen to prevent deadlocks as UE1 may compute faster and try to pass messages to UE0 before UE2.

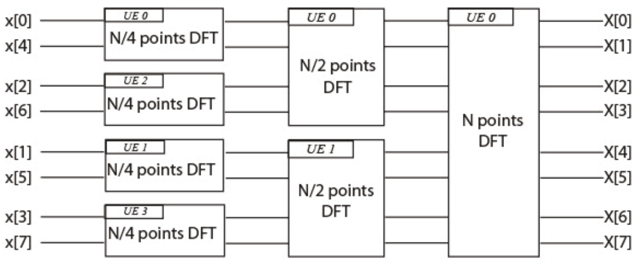


Figure 1: Synchronous message passing Radix-2 FFT algorithm for length-8 data array using four cores. The Core ID indicates distribution of work where DFTs are performed.

In practice, an application performs an FFT on a sequence of input data. Instead of blocking processes to wait for messages, FFT can operate in a pipelined fashion. To avoid the deadlock problem from the first design and enable pipelined operation, the second approach uses 2P-1 UEs. In the case of Figure 1, the total of seven cores are needed for the pipelined method, where four cores perform the lowest level, N/4 points DFT, two cores are responsible for N/2 points DFT and at the highest level, the last core produces the final results by combining N points. As all cores always get messages from a lower level and send to a higher level, incorrect message passing from the first design is prevented. The pipeline algorithm works on more than one FFT at a time and finishes one FFT per step. As the lowest level finishes its computation, it then immediately loads the next set of inputs. The pipelined version was implemented using one-sided message passing (gory mode). The MPB has a limited size of 8 KB per core to hold both communication flags and messages. The algorithm behaves as asynchronous message passing as long as the sizes of messages do not exceed the MPB's capacity. However gory mode on RCCE does not handle messages exceeding the MPB's capacity. In the initialization step, the top-ranked UE divides input into chunks of 4KB and distributes the chunks to appropriate MPB addresses on the receivers' tiles in a round-robin fashion. Once a message is read, the receiver sets an acknowledged-flag on the sender and waits for the next chuck of the message. When the sender sees an acknowledge flag, it resumes and sends the next chunk of code. In the combining step, two UEs pass their results to the same receiver. Both senders put data on their MPB and set sent-flags on the receiver's side. The receiver gets data from the source locations in a round-robin fashion. One-sided communication allows the sender to continue its computation after placing the last chunk of data on MPB without waiting for an acknowledged signal from the receiver.

All UEs have fixed sources and targets. From observations, the most communication is located between the two lowest levels. Because the RCCE hosts file allows to order the ranks of all cores, we arrange such that UEs from the two lowest levels are on the same tile. For instance, according to Figure 1, the first design has UE0 andUE2 on the same tile next to a tile consisting of UE1 and UE2.

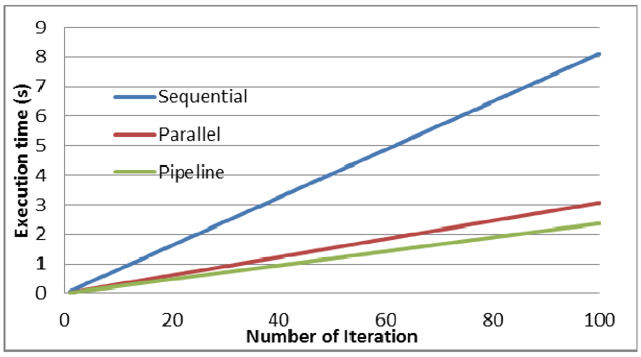


Figure 2: Total execution times for FFT algorithms with input size of 65536 elements.

IV. RESULTS

Our experimental evaluation was conducted using the RCCE V1.0.13 emulator with the g++ compiler version 4.1.2 on an Intel Xeon 2 CPU quad-core computer. For simplicity the input size and number of active UEs were chosen to be a power of two. Each method performs FFT with one, fifty and one hundred input sets. The execution times are shown in Figure 2. In the experiment, parallel synchronous message passing and the pipelined algorithm use four and seven cores respectively, and the input sizes vary from 256 to 1024 elements. The performance improvements are shown in Table 2 and Table 3. For one iteration and input size of 65536, the pipelined version and synchronous message passing achieve on average a 2.58x improvement over the sequential algorithm. However when the input size is 256, a performance loss is shown in the synchronous algorithm and a small improvement in the pipelined version. The main reason is that computation is relatively small with respect to communication time. The pipelined method is faster in this case because the input size is smaller than the MPB's capacity, and it employs asynchronous message passing. With 1000 iterations, the pipelined algorithm achieves up to 3.39x speedup while the parallel algorithm levels off at 2.60x for input of 65536 elements. The algorithms produce correct results on the RCCE emulator. It should be noted that the performance results may differ from a real SCC environment, because the emulator cannot represent the message passing cost of the SCC hardware.

Table1: Speed up of the parallel synchronous message passing version in comparison to the sequential algorithm.

Parallel	Iterations				
Input Size	1	5	10	100	1000
256	0.84	1.57	1.25	1.05	1.02
1024	1.56	1.72	1.66	1.69	1.64
4096	2.03	2.12	2.13	2.12	2.10
16384	2.30	2.40	2.35	2.41	2.36
65536	2.58	2.62	2.61	2.61	2.60

Table2: Speed up of the pipelined version in comparison to the sequential algorithm.

Pipeline	Iterations				
Input Size	1	5	10	100	1000
256	1.33	1.53	1.41	1.37	1.28
1024	1.72	2.39	2.45	2.31	2.38
4096	2.03	2.68	2.90	2.99	3.01
16384	2.36	3.08	3.13	3.23	3.20
65536	2.58	3.22	3.29	3.41	3.39

V. Conclusion

In this paper, we have proposed two fast FFT algorithms for the SCC processor, using the RCCE library. The algorithms aim to minimize communication overhead by distributing UEs such that distances between cores are minimized. The results on the RCCE emulator establish the correctness of the algorithms and show a promising 2.24x speedup for 7 cores with the pipelined method. As for future work, we want to evaluate our algorithms on real SCC hardware and investigate none-recursive FFT algorithms.

References

[1] R. Van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. ACM Operating Systems Review, 2011. in press.

[2] T. G. Mattson et al. The 48-core SCC processor: The programmer's view. In SC'10: Proceedings of the 2010 ACM/IEEE Conference on Supercomputing, New Orleans, LA, USA, 2010.

[3] "SCC External Architecture Specification (EAS)", Intel Coopration, November 2010

[4] Intel's Many-core applications reseach community website. Available: http://communities.intel.com/community/marc/

[5] R. Fisher, S. Perkins, A. Walker and E. Wolfart, "Fourier Transform", Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm, 2003

[6] A. Grama, A.Gupta, G. Karypis, V. Kumar. "Fast Fourier Transform," in Introduction to Parallel Computation, 2nd ed., Edinburgh Gate: Pearson Education Limited, pp.537-548, 2003.

[7] J. Howard et al. A 48-core ia-32 message-passing processor with DVFS in 45nm CMOS. In ISSCC '10: Proceedings of the International SolidState Circuits Conference, 2010.

[8] J. R. Breitenbach, Sequential FFT Source Code in C language, Available: http://courseware.ee.calpoly.edu/~jbreiten/C/

[9] Parallel FFT for Cell Broadband Engine processor, Available: http://www.fftw.org/cell/index.html

[10] J. Cooley, J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Math. Comput. 19, 297–301 (1965).

[11] R. Van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. ACM Operating Systems Review, 2011. in press.

[12] "The SCC Platform Overview", Intel Labs, May 2010.

[13] Matteo Frigo and Steven G. Johnson, "The design and implementation of FFTW3", Proceedings of the IEEE, 2005, pp. 216-231.

Meta-programming Many-Core Systems

Alexander Arlt, Jan H. Schönherr, Jan Richling
arlteini@mailbox.tu-berlin.de, {schnhrr|richling}@cs.tu-berlin.de
Communication and Operating Systems Group
Technische Universität Berlin, Germany

***Abstract*—In this paper we present a novel approach for a highly customizable operating system for modern many-core architectures such as the Single-Chip Cloud Computer (SCC), an experimental processor created by Intel Labs. The customizability is realized by advanced template meta-programming techniques. We employ these techniques at runtime and exploit hardware features not present in today's multi-core systems to create a versatile general purpose operating system.**

I. Introduction

Over the last decade, processor manufacturing made considerable progress, and the mass market turned from single-core computer systems to multi-core systems. Currently, it is expected that this trend will continue and that many-core systems will emerge. While the designers of operating systems were able to adapt their products towards the multi-core era, it is unlikely that these operating systems will be able to handle many-core processors adequately. On the one hand, there are scalability issues within the operating systems itself. On the other hand, future many-core processors may differ from current architectures substantially, invalidating core assumptions current operating systems rely on. An example for this is the Single-Chip Cloud Computer (SCC) experimental processor [1] created by Intel Labs which is a 48-core *concept vehicle* as a platform for many-core software research: while it supports shared memory, it no longer provides cache-coherent shared memory. This makes current multi-core operating systems unusable on this architecture.

Instead, Intel's SCC can be configured and used like a cluster: each core is handled as a single node with distinct memory. While such configurations basically work, they are not able to unleash the full potential of many-core architectures. Furthermore, they greatly restrict the possibilities of application design to a small subset. Therefore, a different approach for many-core architectures is needed. The *multikernel model* [2] addresses this by considering the cores within many-core architectures to be independent from each other but still closely coupled: while the multiple kernels have to use message passing in order to interact, they are – compared to cluster operating systems – lightweight and applications are conceptually allowed to use architectural features, such as shared memory.

A vastly different view on operating system design exists within the area of embedded systems: different techniques were developed to realize highly customizable operating systems fitting a multitude of embedded devices without causing overhead at runtime. This is realized by synthesizing an instance of the operating system in an application-aware step at compile time including only relevant parts and aspects. Examples of this kind of operating system are PURE, CiAO, and EPOS. They use a meta-description of the target system, a *system configuration*. While PURE [3] and CiAO [4] use a set of tools and languages to realize the synthesis out of the system configuration, EPOS [5] achieves this with template meta-programming techniques. Running many-core systems with such highly tailored operating systems allows applications to exploit the full potential of the system. However, this comes at the disadvantage that the tailoring is done offline. This makes it either unsuitable for general purpose systems, or, if general purpose aspects are included in the synthesis, the desired goal of specialization is not achieved.

Therefore, we propose a novel approach for many-core operating systems: synthesizing the operating system at runtime on a subsystem level as an extension of the traditional offline synthesis. This keeps the efficiency of approaches like EPOS, while adding the flexibility of dynamic reconfigurations. Additionally, this concept allows to utilize the unique features of Intel's SCC in order to target a general purpose operating system that specializes on demand while still guaranteeing protection between different subsystems.

The remainder of the paper is structured as follows: In Section II, we present our template-based approach in more detail followed by a description how our concept can be applied to Intel's SCC in Section III. Finally, the paper is wrapped up in Section IV.

II. Approach

Our approach generates optimized, application-specific kernels for individual cores at runtime. This is made feasible by many-core architectures for two reasons: First, there are enough cores available so that sufficient computational resources can be reserved for kernel creation without causing too much distress. Second, modern many-core architectures provide us with hardware support to protect cores from each other. This is necessary as the specialized kernels may drop the classical distinction between user and kernel mode for performance reasons. On many-core architectures we are able to regain that distinction on core level. These ideas perfectly match the SCC architecture as described in more detail in Section III. For the purpose of system management and system reconfiguration, we designate a dynamically changing number of cores as *system cores* which have full access to the system. They generate individual, application-specific kernels

```
namespace config {
typedef model::Hardware<
  architecture::scc::Architecture,
  architecture::scc::Cores<
      Off,            Off, Off, Off, AppA<0>, AppA<2>,
      Off,            Off, Off, Off, AppA<1>, AppA<3>,
      BareMetal<0>, Off, Off, Off, Off,      AppB,
      BareMetal<1>, Off, Off, Off, Off,      AppC,
      Off,            Off, Off, Off, System,  Linux,
      Off,            Off, Off, Off, System,  Linux,
      Off,            Off, Off, Off, Off,      Linux,
      Off,            Off, Off, Off, Off,      Linux
    >
  > Hardware;
} // namespace config
```

Listing 1. Excerpt of an example configuration of an SCC system

for the *application cores* and enforce that application cores are confined to their respective partitions with only restricted interfaces to the remaining system.

To synthesize such highly customized kernels, we use C++ template meta-programming techniques. C++ template meta-programming is Turing-complete and can be interpreted as a functional language directly embedded in C++ that is evaluated at compile time. Our system is mainly driven by Alexandrescu's proposed *policy-based design* [6] with the addition of *traits* [7], [8] and the *curiously recurring template pattern* [9]. In general, these techniques allow the movement of calculations and decisions from runtime to compile-time resulting in a reduced amount of code with increased performance. Policies and static polymorphism enable a high abstraction level in software engineering without sacrificing performance in otherwise low-level kernel development.

System and application cores are described by a concrete system configuration. The model of this system configuration is realized as a collection of template-classes. Different parameters of this model are represented as template arguments. As template arguments can be templates themselves, it is possible to describe configurations of different subsystems recursively. Finally, if all template arguments are specified, the system configuration is complete. From the viewpoint of the compiler, the resulting configuration-model is just a type. Therefore, meta-programming techniques can be used to retrieve information from the model, analyze it, and finally synthesize the desired behavior.

An example configuration of an SCC system is shown in Listing 1 describing the different cores of the SCC system. It contains two system cores and many different application cores. There are no restrictions regarding the type of applications: even other operating systems or bare metal applications are supported. For every application, further configurations exist that describe their interfaces. For example, a parallel application like `AppA` might need to exchange messages. Thus, the synthesized kernels for `AppA` will support message passing between their cores, while this functionality is excluded from the kernels for `AppB` and `AppC`. On the other hand, `AppC` might consist of multiple threads resulting in a scheduler to be included in the corresponding kernel. Individual kernels are reduced exactly to the functionality required by their applications with respect to the underlying architecture. This reduces the complexity and therefore the memory footprint of the resulting kernels drastically. Case studies based on template meta-programming, e. g., [10], [11], exemplarily show a code size reduction of more than 50%. We expect similar results for our approach.

The possibility to execute other operating systems, such as Linux, allows us to seamlessly integrate legacy applications. Applications with known requirements on the other hand, are subject to our individual kernel generation.

A system configuration as described above is sufficient to run a static set of applications. However, in order to realize a general purpose operating system, we include the ability to change the system configuration at runtime by applying differential configurations. The differential configuration serves two purposes: (i) we can derive the next system configuration to generate the necessary core-specific kernels, and (ii) we can create an operating system component that is able to carry out the necessary changes to transform the system from the previous configuration to the next without affecting running but untouched applications.

III. SCC Integration

The SCC architecture [12] provides the necessary hardware features to realize our approach: On the one hand, the complete management of the SCC can be done from within the SCC itself. On the other hand, the SCC allows us to create isolated partitions where bare-metal applications can be executed without them being able to compromise the system. This is due to the fact that all configuration registers are accessed with regular memory accesses which – in turn – undergo a translation via per-core lookup tables (LUTs). Thus, it is possible to remove access to the configuration registers by modifying the LUTs appropriately.

Currently, the initial bootstrapping must be done by the management console PC (MCPC). However, due to the nature of our approach, this can be done using the same code base with slightly adapted policies. Based on an initial system configuration, a boot loader and one or more kernels will be compiled on the MCPC. Template meta-programming techniques create the necessary code within the boot loader, so that it sets up LUTs and other system parameters according to the configuration, stores the generated core images at the desired memory locations within the SCC and initiates the boot process. Depending on the actual use case, the initial system configuration may already contain an arbitrary mix of application and system cores. For minimal bootstrapping, a single system core would be sufficient: after bootup, the SCC can operate without the MCPC by modifying the system configuration, building further kernels and compiling and starting the boot loader for other cores itself.

IV. Conclusion

In this paper we introduced a novel approach for an operating system for many-core systems. Our approach is based on the principle of static offline configuration. However, we apply this idea not on system scale but at subsystem level allowing arbitrary reconfigurations of system partitions at runtime. The configuration itself is based on the concept of template metaprogramming and is used to create operating system images for individual cores that perfectly fit the needs of the software to be executed. This way, we avoid unnecessary overhead while still retaining universality. Furthermore, the approach supports different levels of heterogeneity up to the point where some cores execute existing operating systems while other cores execute applications with highly specialized kernels – either with interactions through shared memory or message passing, or with guaranteed freedom of interference from other applications.

The next step of our research is to transfer this concept into reality. We plan to do this by implementing a prototype that runs on Intel's SCC and demonstrates the feasibility of our approach and allows to evaluate its usefulness compared to more traditional concepts. Ideally, it will be self-contained and turn the SCC into a general purpose system without sacrificing parts of its potential.

Overall, we consider static configurations at core level with dynamic reconfigurations at runtime as a promising way to address the challenges of upcoming many-core systems.

References

[1] J. Howard *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb. 2010, pp. 108–109.

[2] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new OS architecture for scalable multicore systems," in *Proceedings of the 22nd ACM SIGOPS Symposium on Operating systems principles*, 2009, pp. 29–44.

[3] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk, "Design rationale of the PURE object-oriented embedded operating system," in *Proceedings of the International Workshop on Distributed and Parallel Embedded Systems*, 1999, pp. 231–240.

[4] D. Lohmann, F. Scheler, W. Schröder-Preikschat, and O. Spinczyk, "PURE embedded operating systems – CiAO," in *Proceedings of the International Workshop on Operating System Platforms for Embedded Real-Time Applications*, Jun. 2006.

[5] A. A. M. Fröhlich, "Application-oriented operating systems," Ph.D. dissertation, Technische Universität Berlin, 2001.

[6] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, Feb. 2001.

[7] N. C. Myers, "Traits: a new and useful template technique," *C++ Report*, Jun. 1995.

[8] A. Alexandrescu, "Traits: the else-if-then of types," *C++ Report*, Apr. 2000.

[9] J. O. Coplien, "Curiously recurring template patterns," *C++ Report*, Feb. 1995.

[10] C. Steup, M. Schulze, and J. Kaiser, "Exploiting template-metaprogramming for highly adaptable device drivers – a case study on CANARY an AVR CAN-driver," in *Proceedings of the 12th Brazilian Workshop on Real-Time and Embedded Systems*, 2010, pp. 51–62.

[11] R. Klemm and G. Fettweis, "Bitstream processing for embedded systems using C++ metaprogramming," in *Proceedings of the Conference on Design, Automation, and Test in Europe*, 2010, pp. 909–913.

[12] *SCC External Architecture Specification (EAS)*, Intel Labs, 2011, rev. 1.1.

Performance of RCCE Broadcast Algorithm in SCC

Hayder Al-Khalissi
Institute of Computer and Network Engineering
TU Braunschweig
Braunschweig, Germany
h.al-khalissi@tu-braunschweig.de

Mladen Berekovic
Institute of Computer and Network Engineering
TU Braunschweig
Braunschweig, Germany
berekovic@ida.ing.tu-bs.de

Abstract—**RCCE is a small library for many-core communication created for the Single-Chip Cloud Computer (SCC) processors. RCCE has two basic communication primitives, which are point-to-point communication and broadcast. Collective communication are an important aspect of most of the message-passing programming. The Broadcast function is the most heavily used collective operation for the widely used message programming paradigm. In this work, we have implemented, optimized, and compressed RCCE_bcast performance on the SCC. This paper looks at the broadcast function of RCCE and explores some alternative implementations for the SCC architecture. It then compares these implementations to the build-in broadcast RCCE implementation and propose a new implementation for the optimization of one-to-many operation. This proposed algorithm improves 95% of communication time over current broadcast algorithm with large number of cores and message size.**

Index Terms—**RCCE (Message Passing library on many-core system), SCC (Single Chip Cloud Computer), pipeline, collective operations.**

I. Introduction

The Single-Chip Cloud Computer SCC [1], has been designed to explore the future of many-core computing by Intel. It contains 48 P54C Pentium cores connected with a 4x6 2D-mesh. The architecture of the SCC resembles a small cluster or "cloud" of computers. The SCC has 24 dual-core tiles, one off-chip private memory per core, shared off-chip memory, and a shared on-chip message passing buffer (MPB). The MPB is a small fast local memory located in each tile. There are also four DDR3 memory controllers on the chip, which are connected to the 2D-mesh as well. The SCC does not use any cache coherency between the cores, but rather offers a special hardware in terms of these MPBs for explicit message-passing between cores. Each core can boot its own operating system and software stack. The cores are connected via a mesh network with low latency and high bandwidth (256 gigabytes per second). RCCE [2,3] is a Message Passing API has been designed for the SCC and used for programming these SCC features. RCCE is a simple message passing environment based on a simple one-sided (put/get) communication system. The RCCE has two basic communication primitives which are point-to-point communication and collective operations. iRCCE [4], is a non-blocking communication extension to the RCCE communication library that has been developed at RWTH Aachen university. iRCCE extended RCCE by adding asynchronous message-passing functions and improved the performance of some existing RCCE functions. As an example for the blocking send and receive operations is applying of an assembler-coded and SCC-customized memory copy routine. RCCE uses a static Single Program Multiple Data (SPMD) model familiar to message passing programmers [3].

As the number of cores grows the need for cores communication also grows, and since core communication can be one or more orders of magnitude slower than local memory. The communication time can quickly come to dominate the time it takes to complete a computation. Thus, improvement in communication speed can significantly improve the overall performance of an application.

The broadcast function is one of several sub routines defined by RCCE for collective communication operations [3]. Its purpose is to distribute data from one processor to all of the other processors in a communications group. Broadcast is used in situations where, for example, partial results from one processor must be shared among all the processors in a group in order for the computation to continue.

This paper focuses on the algorithms for one-to-all communication and implements several possible implementations of broadcast algorithms in RCCE and compares their performance. Broadcast algorithms implementations are based on the blocking and the pipelining blocking communication. Based on the results of the empirical testing and on analysis of the algorithms, we propose a new implementation.

II. Definition

The following parameters are required for the analytical algorithms presented in the next sections:

Root : the root core is the process that is broadcasting the data.

P : number of cores which take part in the collective communication:

D : the size of the data.

C : the number of (equal-sized) parts into which the data is split by an algorithm. C is also the number of transmissions that take place to transmit all of the data from one processor to another.

T : the communication time that is defined as the time it takes for one core to transfer data of size D to another core in one operation. It consists of time it takes to start up sending (T_s) plus the time it takes to transmit the data.

T_d : the time required for the broadcast function to complete transmitting all data to all processors.

Saturation : for ease of discussion, we define saturation to be, the network state in which every processor is either sending to or receiving from (or both sending to and receiving from) another processor. This is the point at which network throughput is at its maximum.

III. SCC Hardware and RCCE

A. Architectures of the SCC

Fig 1 shows the features of the SCC chip. The SCC is a (mostly) distributed memory, tiled, and many-core processor. Each tile is containing two cores, a Mesh Interface Unit (MIU), a 16 KB Message Passing Buffer (MPB), and two test-and-set registers. The SCC processors P54C cores are second generation Pentium processors. Each core has a 16 KB L1 instruction cache and a 16 KB L1 data cache. The L1 caches are on the core. Each core also has a 256 KB L2 cache that is on the tile. The MPB shared among all the cores on the chip. Each tile is connected to a router. This router works with the MIU to integrate the tiles into a mesh. The MIU packetizes data onto the mesh and de-packetizes data from the mesh using a round-robin scheme to arbitrate between the two cores on the tile [1, 2].

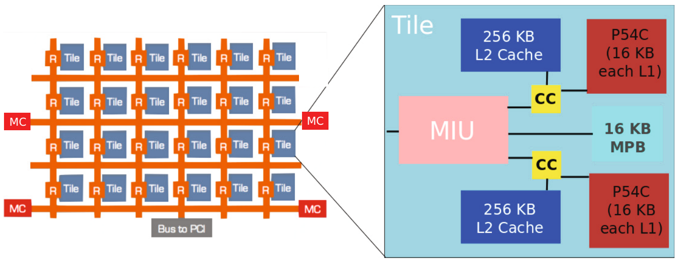


Fig. 1. Layout and tile architecture for the SCC

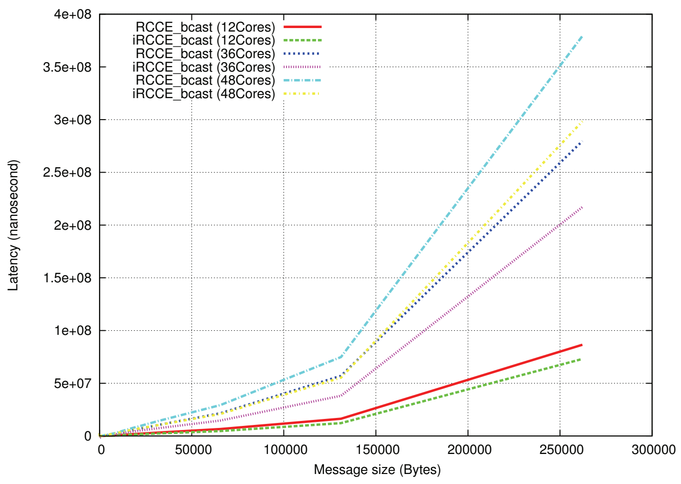


Fig. 2. Communication Latency vs. Message size of RCCE_bcast and iRCCE_bcast

B. SCC Communication Environment (RCCE)

With regards to the architecture in previous section, the SCC supports a variety of parallel programming models. RCCE (pronounced rockey) is a message passing programming model provided with SCC. It is a small library for message passing tuned to the needs of many core chips such as SCC. The communication between cores occurs by transferring data from the private memory through the L1 cache of the sending core to the message passing buffer (MPB) and then to the L1 cache of the receiving core. The MPB allows L1 cache lines to move between cores without having to use the off-chip memory. RCCE has functions that perform the actions as for initialize and shut down the environment, send/receive message among the environment, and synchronize core programs with barriers and fences. It also has functions that manage data between private memory and the MPBs with simple put/get routines, and synchronize core programs using flags [5].

iRCCE is a non-blocking communication extension to the well-known RCCE communication library that extends RCCE by asynchronous message-passing functions. The iRCCE library implements a queuing mechanism in terms of single-linked lists with one send and one receive queue per core. The pipeline approach is implemented within iRCCE in terms of the blocking send and receive functions[4]. The pipelining blocking approach is based on dividing MPB into two smaller chunks, since in this case sender and receiver can work on the MPB simultaneously in a pipelined and parallelized manner.

IV. RCCE Broadcast

The primary problem with broadcast line is how to take advantage of all the available network bandwidth. The fastest algorithms have two characteristics: (1) it gets the saturation (as defined above), and (2) how it is accessing to saturation quicker than the other algorithms. For performance timing analysis, we used the rdtc CPU instruction [6], and Time measurement is only performed and printed on core P_0. This assembly instruction returns the number of clock cycles elapsed since the last reboot of the machine and due to its machines clock precision and minimal execution time, it gives better execution time information when compared to RCCE_wtime( ) of the RCCE library that gives only second precision. We have used 19 data points to draw all graphs in each section which are represented average time for 100 runs.

A. Simple Algorithm

The simplest algorithm of broadcast is for P_0 (root) to send its data to all participating cores. RCCE_bcast is a blocking collective operation which is already implemented according to this algorithm in RCCE library. RCCE_bcast is build on the elementary RCCE _send/recv functions. We implemented iRCCE_bcast by replacing the RCCE_send/recv by pipelining blocking routines iRCCE_send/recv to improve the performance of this algorithm. The total time equal to T times the number of processors minus one: T_d = T $(P-1)$. On the SCC, iRCCE_bcast performs better than RCCE_bcast

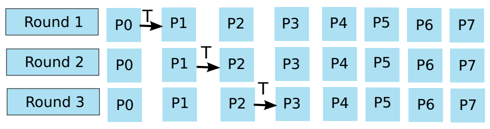


Fig. 3. Chain broadcast running on 8 cores

as the amount of data increases because the T_s divided by C. Both algorithms performance for transferring various D for small (1B) and large (262144 B) are shown in Fig 2. Fig 2 shows the broadcast duration (RCCE_bcast and iRCCE_bcast) which are giving a real average duration of run 100 for each algorithm in the same number of cores. The iRCCE_bcast algorithm was the fastest on all the test with different number of cores when D is larger than 1024 B.

B. Chain Algorithm

Another algorithm with a similar performance lets each core sends and receives at most one message. This effectively creates a kind of ring topology where each core has one predecessor from which it receives the message, and one successor to which it sends the message (for that reason it is also sometimes called ring algorithm). Since the root core does not need to receive a message, this is illustrated in Fig 3, the ring is reduced to a chain where the last core skips the sending part. We implemented RCCE_bcast_chain and iRCCE_bcast_chain based on the point-to-point blocking communication and pipelining blocking communication respectively. Theoretically, these implementations should perform identically, with total time T_d = T $(P - 1)$ in the previous section, but in iRCCE_bcast_chain, T is different because the T_s divided by C. The P_0 completes the broadcast after a single send, and the last core in the chain needs to wait $(P - 1)$ rounds until it receives the message. This gives the following extreme performance numbers : $T_s \leq$T$_s$(simple algorithm). Fig 4, shows the variation of communication time for different message sizes and different number of cores for two implementations of this algorithm. We obvious the broadcast duration in Fig 4 is less than in Fig 2. Also, the Chain algorithm that is implemented by using pipeline blocking communication, it gets better performance with different number of cores and various message sizes.

C. Binary Tree Algorithm

A binary tree is a well-known data structure in computer science. Nodes, which represent RCCE processes, are connected by directed edges, which indicate the direction of the message transfer. To get a good performance. we require that each parent node has two children - except the leave nodes which are allowed to have only a single or no children (Fig 5). This algorithm reduces the number of transmission steps from (P-1) to ($log_2(P+1)$ -1) and T$_d$ = (($log_2(P+1)$)-1)* T. In Fig 5, for an 8-core network, that translates from 7 steps to 4, a reduction of 43%. We also implemented this algorithm based on two routines for blocking and pipelining blocking communications

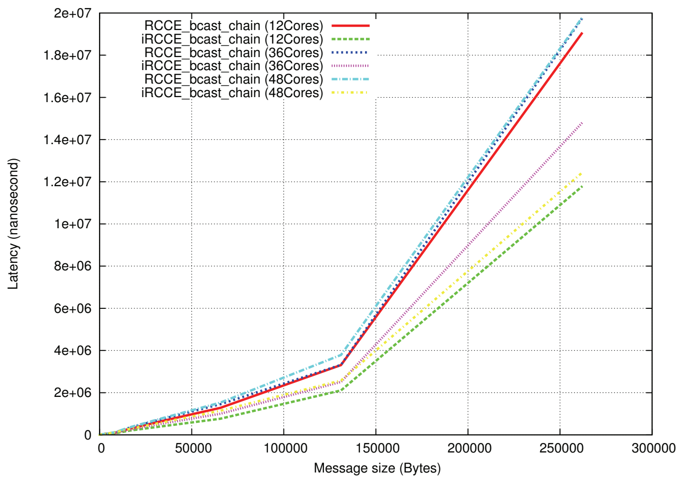


Fig. 4. Communication Latency vs. Message size of RCCE_bcast_chain and iRCCE_bcast_chain

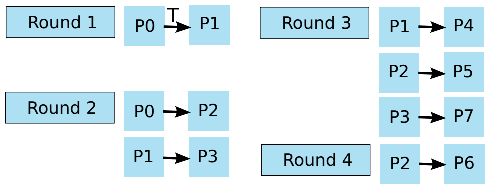


Fig. 5. Binary Tree broadcast running on 8 cores

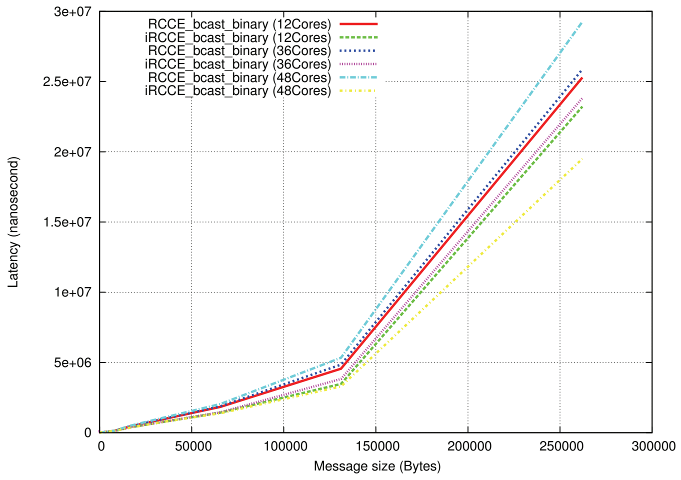


Fig. 6. Communication Latency vs. Message size of RCCE_bcast_binary and iRCCE_bcast_binary

as (RCCE_bcast_binary and iRCCE_bcast_binary). We can see the performance of those implementations in Fig 6.

D. Two Tree Algorithm

We propose a Two Tree algorithm by considering k cores, (P_0,P_1,...,P_{k-1}), where P_0 is the root core. In the first round, P_0 sends a message to P_1. In the second round, P_0 sends to P_5 and P_1 sends to P_2. In the third round, processors 2-5 send to processors 3-6, respectively, and so on, with the number of cores sending and receiving being D at each round (Fig 7). Basically, the number of cores is divided into subtrees:

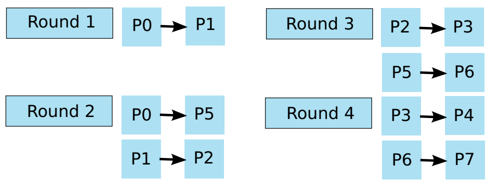


Fig. 7. Two Tree broadcast running on 8 cores

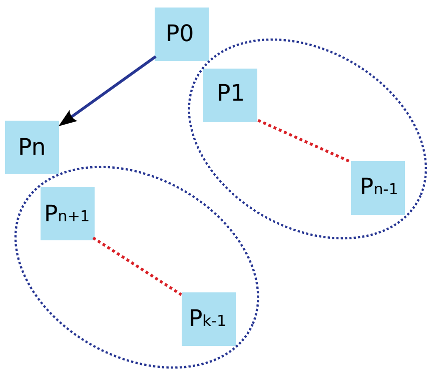


Fig. 8. A Two Tree broadcast with K cores consists of two sub-tree

one is an n-core tree (P_0,P_1,...,P_{n-1}) rooted at P_0 and the other is a $(K-n)$-core tree rooted at P_n as shown in Fig 8. In order to construct a two tree, two requirements must be satisfied. First, the core P_n must be chosen such that the generated subtree is optimal (P_n = $((P/2)+1)$ when P is even, $P_n = (P/2)$ when P is odd) . Second, the two subtrees (P_0,P_1,...,P_{n-1}) and (P_n,P_{n+1},...,P_{k-1}) must apply the same algorithm to each subtree. We chose Chain algorithm to be implemented in each subtree because it has better performance. Thus, this algorithm improves the performance by reducing the number of transmission steps to ($log_2(P+1)$ -1). The time measurement for this algorithm is T$_d$ = ($log_2(P+1)$-1)* T that is identical to T$_d$ in section (C) but the difference in time transmission between cores as illustrated in Fig 8. As mentioned in [7], that shows the latency comparative for transfer messages between same tile, neighboring tile, and farthest tile. T_s between cores in the same tile or neighboring tile is less than T_s between the farthest tile. As result, T$_d$ in section (C) is greater than T$_d$ in this section because the Two Tree algorithm are exploiting neighboring core and tile to save broadcasting duration. Fig 9 presents the performance results of RCCE_bcast_2tree and iRCCE_bcast_2tree which are implemented by using point-to-point communication blocking and pipelining blocking respectively.

V. Performance Comparison of various algorithms

The goal of this section is to compare the performance of RCCE broadcast and iRCCE broadcast operations on the SCC. The purpose of this comparison is to determine the best performance of sending message from one processor to all processors using two-sided routines (blocking and pipelining blocking). For all measurements the cores were running at 533

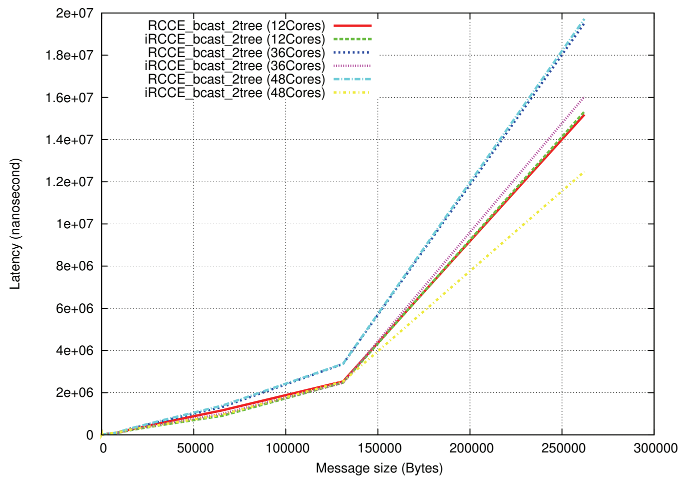


Fig. 9. Communication Latency vs. Message size of RCCE_bcastt_2tree and iRCCE_bcastt_2tree

MHz and the mesh at 800 MHz.

To study the performance of RCCE and iRCCE broadcast communications commands on SCC, we have run the blocking RCCE version of our code for transferring various block sizes starting (1B) and up to (262144 B) of data to different numbers of processor. After several runs for each case, we noticed the time in Fig 10 and 11 which made clear the differences between algorithms. Fig 10 shows the performance of four algorithms which are based on blocking communication sending. While, Fig 11 depicts the performance of four algorithms which are based on pipeline blocking communication routine. On the SCC, for 1-Byte and 262144-Byte message, the RCCE/iRCCE broadcast algorithms implementation is significantly slower than all the other algorithms implementation as shown in Fig 10 and Fig 11.

This study is analogous to the previous sections; the communication time required for transferring the same data block size is less in pipelining blocking broadcast than in broadcast blocking. The difference increases with larger data block size and with more number of processors. We observe the difference in performance of RCCE and iRCCE operations in each implementation separately as illustrated in Table 1.

Fig 12 explains chart of time performance variation for each broadcast function for transferring two different message size (8192 B, 262144 B) to 48 processors. It clearly shows that iRCCE_bcast_2tree outperforms all the other algorithms for different data size sets on the SCC. This proposes that the overall performance of RCCE broadcast improved by used pipelining blocking routines. For the small D on a small number of P, iRCCE_bcast (simple algorithm) performs the best. For large D, iRCCE_bcast_2tree works the best.

Finally, Table 1 summarizes the percentage improvement in completion time between RCCE_bcast function and various broadcast algorithms which are implemented on SCC. Table 1 shows that for messages sizes 8192 B and 262144 B, RCCE_bcast_2tree saves 96% and 95% respectively over RCCE_bcast.

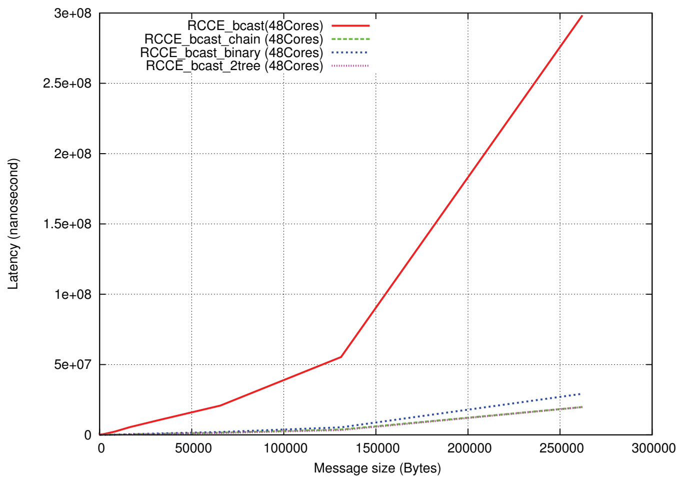


Fig. 10. Performance of RCCE broadcast algorithms

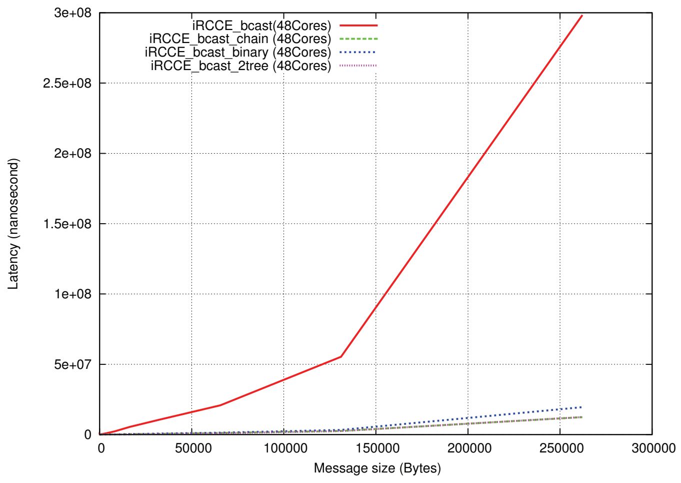


Fig. 11. Performance of iRCCE broadcast algorithms

However, the formula for the T_d is identical to the RCCE_bcast_binary (as explained in section (4.C)), but RCCE_bcast_2tree is vast because it reduces the time for transfer messages between cores by sending message to neighbor. Therefore, we reduce the number of transmission steps and time of transmission. Table 1 also showed that RCCE_bcast_chain gets better performance, as compared to the RCCE_bcast.

We compare the broadcast algorithms communication timing which are improved by pipelining blocking operation with iRCCE_bcast as reported in Table 2. According to Table 2, those algorithms which are improved by pipelining blocking send functions, the communication time saved more than 93% over iRCCE_bcast with number of cores = 48. Where iRCCE_bcast_2tree saved time 96% over iRCCE_bcast time and 1% over iRCCE_bcast_chain when is sending data with size 262144 B. It is obvious, for broadcast improvement algorithms, processors can better utilize the waiting time in sent data by using pipelining blocking send, showing better performance compared to the blocking send.

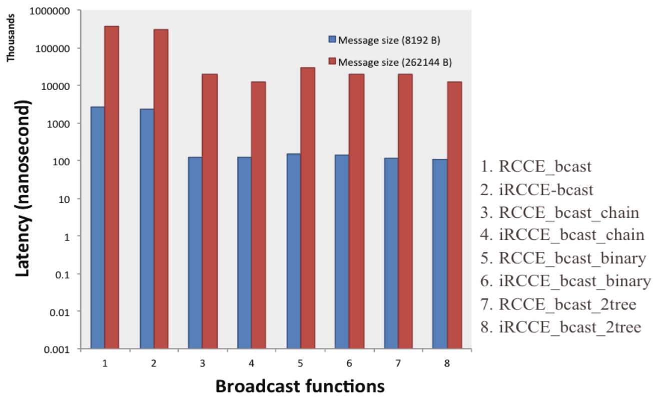


Fig. 12. Comparison RCCE broadcast algorithms performance on 48-cores

TABLE I
COMPARISON BETWEEN RCCE_BCAST VARIOUS BROADCAST ALGORITHM WITH % IMPROVEMENT IN TIME

Message size	iRCCE_bcast	bcast_chain	bcast_binary	bcast_2tree
8192	13%	95%	94%	96%
262144	21%	95%	93%	95%

TABLE II
COMPARISON BETWEEN IRCCE_BCAST VARIOUS BROADCAST ALGORITHM WITH % IMPROVEMENT IN TIME

Message size	bcast_chain	bcast_binary	bcast_2tree
8192	95%	94%	95%
262144	96%	93%	96%

VI. CONCLUSION

From the above experimental study on the SCC architecture, it is interesting to note that with different number of cores, the broadcasting can be implemented using different algorithms to save time (in nanosecond) appreciably. The performance of broadcast in RCCE can be improved on SCC for data sets by changing its implementation. The broadcast implementation on RCCE uses simple algorithm. We could switch to RCCE_bcast_2tree or iRCCE_bcast_2tree which have better performance for different message sizes would work. So, the experimental results reveal the fact that for different sizes of message and number of cores, it is better to implement broadcasting in terms of pipelining blocking point-to-point communication in parallel programming by using RCCE library on the SCC system.

ACKNOWLEDGMENT

We wish to express my sincere gratitude to Intel for providing SCC system and software development platform. We also thank Intel Lab staff in TU Braunschweig and Adrian Knoth (Jena University) for helping.

REFERENCES

[1] Intel Corporation. *SCC External Architecture Specification (EAS).* July 2010. Revision 0.99.

[2] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. *The 48-core SCC Processor: The Programmers View*. In Proceedings of the 2010 ACM/IEEE Conference on Supercomputing (SC10), New Orleans, LA, USA, November 2010.

[3] Ernie Chan. *RCCE comm: A Collective Communication Library for the Intel Single-chip Cloud Computer*. 2010.

[4] Carsten Clauss, Stefan Lankes, Jacek Galowicz, Thomas Bemmerl. *iR-CCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer*. February 22, 2011.

[5] T. Mattson and R. van der Wijngaart. *RCCE: a Small Library for Many-Core Communication*. Intel Corporation, May 2010. Software 1.0-release.

[6] Intel Corporation. *Intel Application Notes - Using the RDTSC Instruction for Performance Monitoring*. Technical report, Intel, 1997.

[7] Aparna Chandramowlishwaran, Richard Vuduc, and Kamesh Madduri. *Performance Evaluation of the 48-core SCC Processor*. LBNL ICCS 2011 Workshop.

ARGOS - a software framework to facilitate user transparent multi-threading

Nils Petersen
DFKI GmbH
Email: nils.petersen@dfki.de

Julian Pastarmov
Google Germany GmbH
Email: pastarmovj@google.com

Didier Stricker
DFKI GmbH
Email: didier.stricker@dfki.de

***Abstract*—In this paper we present a software framework called ARGOS designed for auto-parallelizing algorithms and distributing its parts among physical machines.**

The core approach is to split an algorithm into an abstract, a target specific static, and a run-time part. The abstract part is an abstract graph representation of the distinct steps of an algorithm with all data dependencies explicitly resolved.

The target specific part is statically distributing graph-parallel parts of the algorithm among threads and physical machines and in addition associates each algorithm step with an optimal implementation for the respective target hardware.

The run-time part covers load-balancing and - in presence of streaming data - parallelizes sequential branches of the graph by stage-parallel execution.

The underlying programming model allows for optimal partitioning of the graph in terms of computational load and memory resp. bandwidth footprint. All thread synchronization and memory management is hereby carried out by the framework making the multi-threaded execution completely transparent to the user.

The software is in daily use within our research group and has already been deployed to several industrial projects.

I. Introduction

Developing efficient software for current multi-core and future many-core hardware is becoming an increasingly difficult task. While some constraints, like the demand on memory efficiency could be relaxed to some extent due to cheap memory, the paradigms for writing fast code has switched to making extensive use of multi-threaded execution.

This puts higher demands on the algorithmic design and thus the skills of the developer and provides an additional source of hard-to-catch code issues. To alleviate the demands on the programmer, we have designed a software framework that is able to perform many parallelization steps either completely automatically or with only little interaction.

The framework hereby implements a constrained but sufficient programming model that implicitly favors algorithmic architectures that afford parallelization.

The general optimization strategies of Argos are threefold:

- Optimize by separating the implementation from the declaration of an algorithm. This allows to choose an optimal implementation for a given target hardware without changing the algorithm design.
- Optimize by graph-parallel execution and distribution of fragments to different threads and physical machines. The goal is to partition the graph into balanced fragments in terms of computational load with minimal bandwidth footprint in between.
- Optimize by stage parallel execution, accelerating graph-sequential parts. This technique used by most modern processors allows to exploit hardware parallelism also for explicitly sequential algorithms.

Although the software framework is still under heavy development it is already in daily use in our research group and has proved its applicability to many use cases in the fields of real-time computer vision, sensor fusion, as well as offline processing and ”number crunching”. We call our framework ARGOS after the 100-eyed giant from greek mythology in reference to its principal field of application, the parallelization of computer vision algorithms.

In the remainder of this paper, after shortly discussing related work, we will show the means of optimization and the underlying architecture in more detail.

II. Related work

Since there seems to be a consensus that the current programming paradigms have to - at least - be extended to facilitate writing highly parallel applications, new languages or programming models evolve.

One successful member of this fraction is nVidia CUDA [3] that allows the development of general purpose software on the GPU without the inconvenience of requiring to disguise the implementation as a graphical shader program. The CUDA concept extends C and recently C++ by simple directives to advise the compiler on the number of dedicated processing cores and blocks as well as to influence data locality. Unfortunately CUDA is limited to nVidia GPUs. A more comprehensive change in comparison to classical programming languages is OpenCL [4]. OpenCL is a so called stream processing language, thus having a very similar goal as CUDA but is a common standard and multi-platform targeted.

While it is very easy to create a working implementation using both of these languages it is quite hard to achieve an actual speed up. The developer needs considerably more insight into the underlying hardware as well as the preferred data locality compared to writing code for *e.g.* modern x86 hardware with its automatic multi-level caches.

To that end there is the class of declarative languages, *e.g.* the Microsoft Accelerator concept [5] where the parallel programming is achieved through merely declaring the data parallelism rather than explicitly creating and dispatching threads. Through the declarative nature, the concept can even synthesize into FPGA circuitry [1]. This concept is the closest to ARGOS as our program representation is a declaration of the steps necessary to solve a problem as well as the fully resolved data dependencies for each of these steps.

III. System description

We have given careful thoughts of a how to design a system that

- implicitly influences the developer to favor architectures that facilitate parallel execution
- hides the complexity and the synchronization overhead of multi-threaded execution
- is able to automatically parallelize and optimize parts of the provided algorithm
- allows to make use of hardware like GPUs without requiring a certain system configuration
- can distribute code among heterogeneous physical machines and clusters
- still allows the developer to formally override all these mechanisms to provide specifically optimized implementations

In the following we explain the distinct concepts we have implemented to meet these requirements. The enumeration is in didactic order and not reflecting the utility or priority of concepts.

A. Component-based graphs with full data dependency resolution

The component paradigm has several natural benefits for the purpose of parallelization. The fact, that the developer provides a component- (or filter-)graph that describes the algorithm affords extracting static information about run-time behavior.

The nodes in our graph are placeholders for what we call modules. We will explain this in detail in the following section.

We require that all data dependencies are made explicit in the graph. Either by ”wiring” a consumer to all associated data providers in the graph or by referencing constant default values, see figure 1 for an example graph. A module is allowed to opt out this requirement by declaring that it maintains internal state. This affects the scheduler to treat these modules differently as for instance the order of execution is important. From a performance perspective the biggest drawback is that the framework cannot automatically replicate these modules to use them within a single instruction multiple data (SIMD) configuration. Examples for modules that maintain internal state would be a module providing camera images (the reading of the sensor provides the internal state) but also probabilistic modules where the seed for the random number generation is not made explicit.

To facilitate global loops that maintain an internal state, we allow for cycles in the graph. To initialize cyclic graphs, ARGOS provides dedicated trigger policies for boot-strapping and reinitialization.

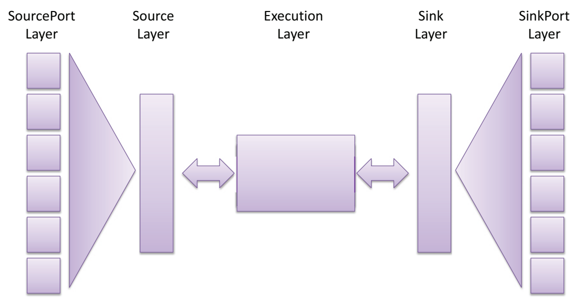


Fig. 2. The internal structure of a module using 5 distinct layers.

After all, this representation gives the scheduler a lot of information that is either hard or incompletely acquirable through code analysis. The downside is that the approach depends on how well the developer has divided the algorithm into smaller pieces, as each component is treated as a black-box by the scheduler.

B. Strong separation between algorithm and implementation

Binary code that runs fast on a given target hardware might not be optimal for another one. Especially since one of the goals is to facilitate cluster processing where not all machines might have the same hardware configuration. For example a general purpose GPU (GPGPU) might be available on one machine and not on the other. Additionally an optimal algorithm is dependent on the nature and the amount of data that has to be processed.
Since Argos has the focus to distribute code among several machines abstraction from actual implementations becomes a necessity. Therefore, when constructing complex algorithms the building blocks never directly refer to concrete implementations but rather algorithms or even classes of algorithms.

Since the graphs resolve all data dependency, each implementation has to be able to cope with the provided set of information. This does not mean, that two implementations of the same algorithms have to provide the same interface but that both implementations have to comply with the set of required data as defined by the abstract algorithm declaration.

Due to this, ARGOS can distribute parts of an application before actually associating any implementations at all.

The combination of signature and actual implementation for an abstract algorithm is called module. An arbitrary amount of modules can be registered to an algorithm.

C. Language agnostic implementations

A module is comprised of five different layers, see figure 2. The SourcePort and SinkPort layers are doing the actual data synchronization that will be explained in the following subsection. The Source and Sink layers subsume the information from the respective ports and communicate with the so called execution layer.
The execution layer contains the actual implementation and is the only part that has to be provided by the developer. The source code in algorithm 1 lists the complete execution

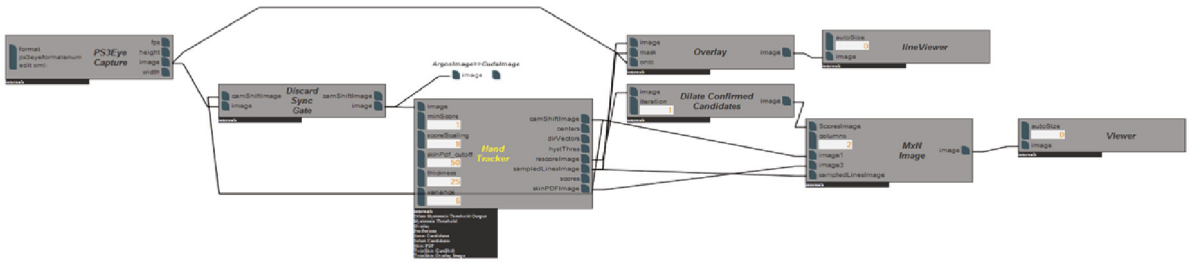


Fig. 1. An example graph visualized in the web based user interface. The components in the graph are cascadable. The "Hand Tracker" component in the middle of the graph for example comprises several sub-components which allows for internal parallelism.

layer implementation in C++ for a simple module that outputs the width and height of an image. Since the data input and output of an actual implementation is not required to be exactly identical with its associated abstract algorithm, the developer declares this in the implementation. ARGOS then creates all other module layers from this definition.

To make the syntax as convenient and unobtrusive as possible, the input and output buffers can be mapped to shadow variables that are managed by ARGOS.

The execution layer is completely language agnostic, besides C++ and Cuda we have provided implementations in Matlab, Python, and Lua.

D. Asynchronous execution of single-threaded implementations

Each module is entirely self-dependent besides its specified data dependencies. This property allows to execute data parallel (which is exactly equivalent to graph-parallel) modules in parallel. Additionally ARGOS is able to parallelize sequential branches of the graph in presence of streaming data by using stage parallel execution: While the second module in the pipeline processes the output of the first module, the first module can already start processing the next frame of data.
This becomes possible by thread-safe buffers maintained in the sink ports. Figure 3 illustrates a simplified sketch of the asynchronous data handover between two modules. Once a module gets triggered, *i.e.* all data that is necessary for the implemented algorithm to run is available, ARGOS selects the correct input and output buffers for this iteration (in case of stage-parallel execution) and invokes the run-method of the module.

Within the run-method, the module has exclusive writing rights on its output buffers and guaranteed reading rights on the input buffers.
The module developer does not need to care about thread and data synchronization. The code within the run method can be written without any additional constraints compared to single threaded applications. Thus, the developer is also free to provide a multi-thread implementation within a single module.

The single-thread nature of each module is also not violated when Argos parallelizes SIMD operations as every worker thread will be represented by its own module.
This has two advantages. First, the scheduled graph is a sufficient representation of a parallel schedule for an algorithm. Second, this constraint is the prerequisite to maintain and update internal state.

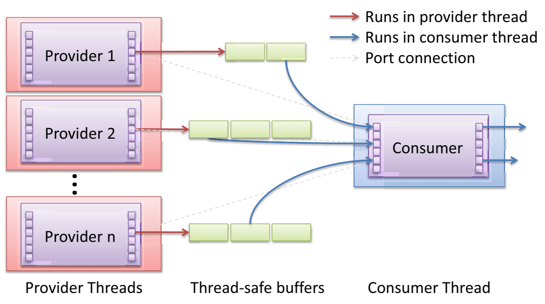


Fig. 3. Asynchronous data handover between two modules.

E. Instrumentation and optimal cuts

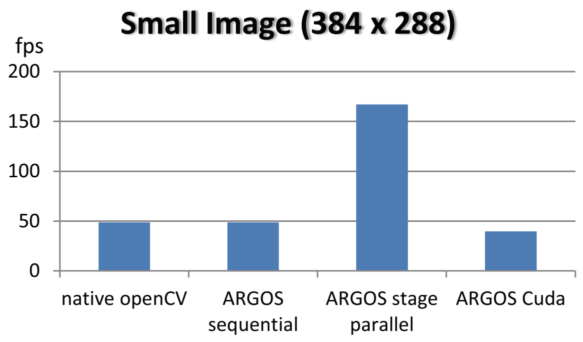
As the topology is static and all data dependencies are explicit, a scheduler can easily measure the computational and bandwidth footprint of each single module. This allows to compute optimal cuts through the graph leading to load-balanced fragments that can eventually be distributed among physical machines.

As already stated above, the graph is a sufficient representation of the data parallel schedule which means this load-balancing can be performed in a nicely structured way.

F. Delayed scheduling

When establishing the graph representing an application or algorithm, everything is solely defined abstractly. The process of scheduling associates implementations, dispatches worker modules for SIMD tasks and distributes among several physical machines.

When not doing a profile guided scheduling this is performed in less than a second and happens in the moment when a graph gets started. Hereby only the most performing basic optimizations are performed:
Every algorithm is scheduled to run in its own thread which allows for full stage-parallel and graph-parallel execution but leads to severe overhead problems with larger graphs due to the high amount of threads.
For every algorithm its default implementation is selected and no distribution to different physical machines is performed.



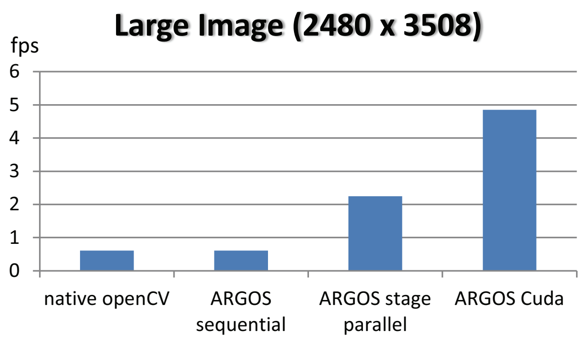


Fig. 4. Performance comparison for small (left) and large (right) input cardinality. The experiment was performed on a Intel Core2Quad 2.5 GHz with nVidia 9800 GTX

A profile guided schedule exploits information about computational and bandwidth footprint of each module acquired through instrumentation. Currently this schedule is in large part computed through brute force, trying out every possible combination of implementation and measuring the resulting common performance.

Algorithm 1 An example module implementation. The module receives an input image and outputs its width and height.

```
class ModuleExample: public Module
{
private:
  const argos::Image *_inImg;
  argos::Int         *_outWidth;
  argos::Int         *_outHeight;

protected:
  ModuleExample() {
    createSource("image", &_inImg);
    createSink("width", &_outWidth);
    createSink("height", &_outHeight);
  }

  void run() {
    *_outWidth  = _inImg->getWidth();
    *_outHeight = _inImg->getHeight();
  }
};
```

IV. Performance Evaluation

To give an impression of possible speed ups we evaluated a simple sequential algorithm. We implemented chose closing, which is simply a number of subsequent image dilate and erode operations. We chose a total of 4 erode/dilate steps to provide enough stages to saturate a quad-core. First we did a ”native” implementation using OpenCV [2]. For testing with ARGOS we implemented the two trivial modules that simply wrap the OpenCV calls for erode and dilate and have built the equivalent pipeline: image reader - dilate - dilate - erode - erode.
Additionally we provided ARGOS with a CUDA implementation of erode and dilate.

The results of the experiment can be reviewed in figure 4. Both for small and big input images the overhead due to ARGOS is below the measurement accuracy. When ARGOS was allowed to perform stage parallel execution, the application performed at about 3.3 times the speed of the single threaded execution resp. at 80% of the theoretically achievable maximum on a quad core. Forcing the ARGOS scheduler to dispatch erode/dilate to CUDA implementations, performance was slightly worse on small input cardinality. On bigger images, the CUDA implementation could exceed the CPU implementation. This underscores the utility of a loose association between algorithm and implementation. The optimal choice of algorithm is depending on the target platform as well as on the nature of data to process. The possibility to delay the actual scheduling to the time information about both is available is a desirable property.

V. Summary and Future Work

We have presented ARGOS, a component based software framework to design, implement, and execute algorithms on possibly heterogeneous hardware. By abstracting from implementation details, algorithms can quickly be adapted to accommodate a specific target hardware.

Currently we adapt the software to run on Intel Single-Chip Cloud (SCC) many-core CPU. Future work includes improving our scheduling that exploits computational and bandwidth footprints also optimizing schedules for the SCC. Of particular interest for SCC schedules is to meet topological conditions like router distance or thermal considerations.

Acknowledgment. This work has been partially funded by the project CAPTURE (01IW09001) and the German BMBF project AVILUSplus (01M08002).

References

[1] B. Bond, K. Hammil, L. Litchev, and S. Singh. Fpga circuit synthesis of accelerator data-parallel programs. In *FCCM*, 2010.
[2] G. Bradski and A. Kaehler. *Learning opencv*. O'Reilly, 2008.
[3] C. Nvidia. Programming guide, 2008.
[4] J. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010.
[5] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data-parallelsim to program gpus for genral purpose uses. In *ASPLOS*, 2006.

Power-aware Dense Linear Algebra Implementations on Multi-core and Many-core Processors

Pedro Alonso*, Manuel F. Dolz†, Francisco D. Igual†, Bryan Marker‡,
Rafael Mayo†, Enrique S. Quintana-Ortí† and Robert A. van de Geijn‡
*Univ. Politécnica de Valencia, 46.022 - Valencia, Spain
†Univ. Jaume I de Castellón, 12.071 - Castellón, Spain
‡The University of Texas at Austin, TX 78712

***Abstract*—This paper outlines our research on the application of power-control techniques to the execution of dense linear algebra operations on modern multi-core processors and hybrid CPU-GPU architectures. The framework is based on the SuperMatrix runtime system which exploits the inherent task-parallelism present in most blocked dense linear algebra algorithms. As part of the on-going work, we analyze the possibility of extending the power-aware techniques to novel many-core architectures, such as the Intel SCC processor.**

***Index Terms*—Dense linear algebra, power consumption, multi-core processors, DVFS, Intel SCC processor.**

I. Introduction

During the last decades, the combined pressure from the mobile and embedded market segments has forced hardware manufacturers to improve the power efficiency of their designs. Large-scale high-performance computing (HPC) facilities have greatly benefited from this shift towards greener information technologies hardware. However, most current scientific, engineering and industrial applications running in the HPC centers are quite oblivious to the possibilities offered by the underlying hardware, in spite of the significant assets it can yield [1]. This is no longer possible: power consumption has a direct impact on the operation and maintenance costs of these centers, compromising their existence and impairing the deployment of new facilities [2], [3].

A major part of the computations that are needed to solve many of these HPC applications can be cast in terms of a reduced number of well-known linear algebra problems like, e.g., basic matrix algebra operations, linear systems of equations, or eigenvalue and singular value problems. Recent work has demonstrated the advantage of exploiting task-level parallelism present in these dense linear algebra (DLA) operations when targeting multi-core processors [4]. In these projects, (blocked) DLA algorithms are decomposed into a collection of tasks (or kernel operations), forming a directed acyclic graph (DAG) which captures the parallelism information implicit in the algorithm. This can then be used by a scheduler to optimize the degree of parallelism during the execution of the algorithm.

In this paper we briefly describe how task-level parallelism, and the potential scheduling information stored in the DAG, can be effectively used to reduce power consumption of task-parallel codes for linear algebra operations on multi-core processors and hybrid CPU-GPU architectures, and how the same strategy can be effortlessly transferred to a drastically different architecture: the Intel SCC experimental chip. We take benefit from the separation of concerns advocated by the FLAME framework, more specifically by the versatility of the SuperMatrix runtime system [4], to port our power-aware techniques to different architectures, namely multi-core processors, hybrid CPU-GPU architectures, and novel many-core architectures such as the Intel SCC chip.

While there exist a number of related investigations on the trade-off between energy and computational performance [5], and the design of power-aware schedulers [6], none of these works explicitly tackles the exploitation of task-level parallelism present in DLA codes on multi-core and accelerator-based and novel many-core architectures.

II. The SuperMatrix runtime as a framework for power-aware computing

The conventional approach to extract performance in the linear algebra domain focused on exploiting parallelism at the BLAS level. Therefore, the efficient concurrent execution of scientific applications written on top of these kernels heavily depended on the expertise of the BLAS developers, usually armed with a deep knowledge of the architectural details.

More recently, the FLAME project (as well as several other projects in this and other more general domains) has shown the benefits of a different approach, one that extracts parallelism at a higher level, so that only a sequential tuned implementation of the BLAS routines is necessary. In this approach, extracting parallelism is left in the hands of a runtime system, SuperMatrix in the case of the FLAME project and the associated `libflame` library [4] for DLA. Precisely this runtime is the tool we plan to leverage in order to extract parallelism and tune power consumption on a fully novel architecture like the Intel SCC chip.

The SuperMatrix runtime adapts `libflame` to modern multi-core architectures. It provides automatic parallelization of dense and banded linear algebra algorithms. The framework exploits the benefits of storing and indexing matrices by blocks and algorithms-by-blocks in general, applying techniques for dynamic scheduling and out-of-order execution (common in superscalar processors) and a systematic handling of data dependencies at execution time.

Our efforts towards power conservation start with the DAG of tasks that need to be performed to compute a given DLA operation. It is based on two key observations. First, if tasks are all run at full speed (i.e., highest frequency), some cores will experience idle periods during the execution of the DAG. Second, present processors are quite good at adjusting frequency/voltage (DVFS) dynamically and hence the energy consumed. In previous works, we have investigated and demonstrated the theoretical possibilities of two different power reduction policies, the *slack reduction algorithm* (SRA) and the *race-to-idle* approach, which precisely aim at saving energy by controlling the operation frequency of the cores [7]. The SRA algorithm detects and avoids idle periods in the operation of cores by reducing the frequency at which certain tasks are performed (in particular, those which do not increase the global execution time if slowed down). The race-to-idle algorithm (RIA), on the other hand, operates at the maximum frequency, so as to generate longer idle periods which can be exploited by promoting the cores to a low consuming state. Our experiments using a simulator determined the superiority of the RIA for the execution of DLA operations based on level-3 BLAS kernels. Our aim next is to experimentally validate these results for realistic DLA problems on general-purpose multi-core architectures, and simultaneously extend them to hybrid CPU-GPU architectures, and the SCC processor, by controlling which resources are idle at each time of the parallel execution, reducing their frequency as they become idle.

Specifically, the task dependency information generated at runtime and stored in the DAG can be effectively used to design novel power-saving strategies. For example, given a list of ready tasks, inactive threads can be blocked until new ready tasks are generated. In the meanwhile, the frequency of the corresponding cores is adjusted accordingly to the chosen power-aware heuristic. These idle times can be controlled via busy-wait or, better from the energy savings viewpoint, estimating the execution time of each task. Factors like cache affinity, reuse distance, or number of pending tasks to be released upon the completion of a given one, can then be exploited in the design of new power-aware scheduling policies.

III. Accommodating Power-Aware Techniques into SuperMatrix

Reducing the frequency/voltage operation in CPU-bounded DLA operations (as those in the level-3 BLAS as, e.g., `gemm`) entails an increase in the execution time that is directly translated into higher energy consumption. Despite being also CPU-bounded kernels, the behavior of DLA operations like the LU factorization with partial pivoting can be potentially different. Specifically, the existence of task dependencies in algorithms-by-blocks when task-parallelism is exposed yields idle periods during the computation (depending on a large variety of factors like, e.g., the problem dimension, efficiency of the computational kernels, number of cores, scheduling algorithm, etc.). Modern operating systems offer mechanisms (*governors* in the Linux kernel) to set idle threads into power-hungry/power-save modes by increasing/reducing their operation frequency and voltage scaling. While idle periods in

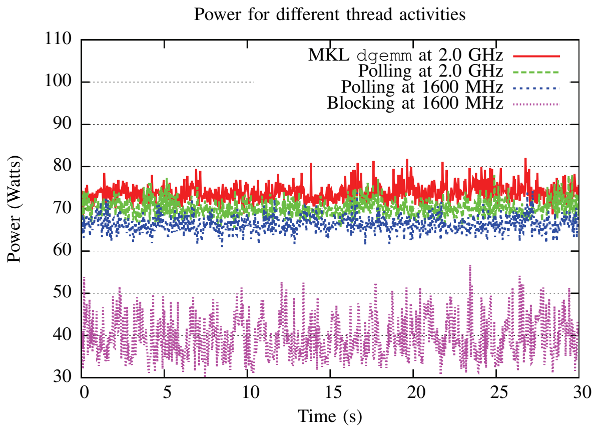


Fig. 1. Power consumption of different actions performed by threads.

DLA implementations can be exploited by selecting a given governor for the entire application, our aim is to integrate this mechanism into the runtime level. This approach allows the application of more sophisticated techniques, specifically tailored for each DLA operation and system configuration. In addition, the designed techniques can be seamlessly migrated to other architectures where the runtime is already available, in our case, hybrid CPU-GPU architectures and the SCC chip, to name only two.

In this paper, we propose two different complementary approaches to integrate power-aware policies into the SuperMatrix runtime. The first energy-saving technique scales the frequency operation of the cores when the corresponding threads are idle. Proceeding in this manner, when a thread finds no pending tasks in the list of ready jobs, the runtime immediately scales the operation frequency of the associated core to the lowest possible (system call `cpufreq`). On the other hand, as soon as the poll for new ready task is successful, the frequency is raised back to the highest, in preparation for the execution of the corresponding job. The main benefit of this operation mode is a reduction in the polling rate, which is beneficial from the point of view of energy saving. To illustrate the impact of the frequency reduction in the polling phase, Figure 1 illustrates the difference in energy consumption between a thread that performs polling at 2.0 GHz and one that does the same at 1600 MHz on a dual socket Quad-core Intel E5504 processor (when all remaining cores are idle); in this case, energy consumption is decreased from around 75 Watts to less than 70 Watts. Note also the power consumption of a thread performing polling at the highest frequency is only slightly smaller than that of one performing useful work like, e.g. a matrix-matrix product (MKL `dgemm`).

These results also reveal an interesting insight that conforms the basis of our second technique. Observe that a thread performing the busy-wait corresponding to polling, even at 1600 MHz, still employs a considerable amount of energy. However, when the same thread is blocked, the consumption is decreased significantly, to 40–50 Watts.

Taking into account this observation, our second power saving technique replaces the polling strategy for new ready tasks by a blocking policy. In our implementation we employ

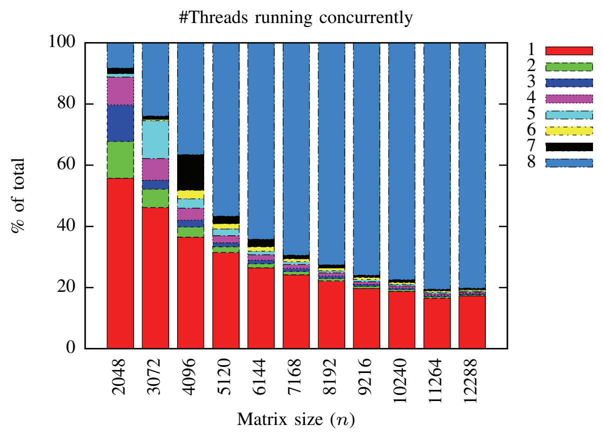


Fig. 2. Thread activity during the execution of the LU factorization with partial pivoting.

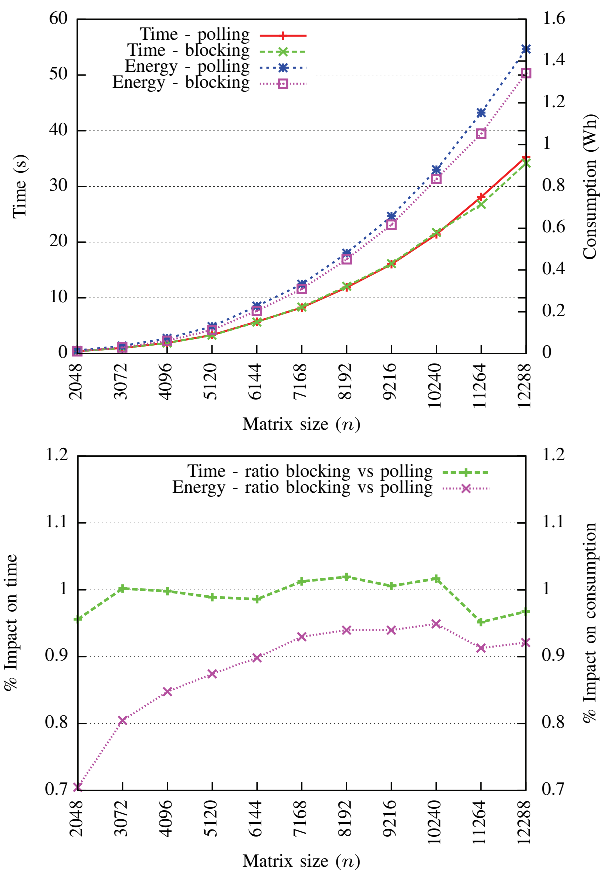


Fig. 3. Evaluation of the impact on time and energy of the power-saving strategies.

POSIX semaphores to control the active threads. Now, when a thread finds no ready tasks in the corresponding list, it blocks itself instead of keep on polling for new jobs. Also, upon the completion of the execution of a task, the corresponding thread updates the dependencies of tasks in the pending list. In case that this requires moving k tasks from the pending list to the ready list, using our strategy, this thread will also enforce that (at least) there exist k active threads, awakening blocked threads in case it is necessary. The basic effect of this mechanism is that there is basically one active thread per task in the ready list and, key to power conservation, that no continuous polling is being done on an empty list, with the corresponding theoretical energy savings. (Whether this technique yields an actual gain ultimately depends on the existence and duration of idle periods during the parallel execution of the algorithm and the overhead of blocking/activating a thread.)

IV. Experimental Results

All experiments reported next were obtained using double-precision arithmetic on a dual socket Quad-core Intel E5504 processor (2.0 GHz) with 32 Gbytes of DDR3 RAM. The system runs a Linux Ubuntu 10.04 distribution. We use the Intel `icc` compiler (version 11.1), and highly tuned implementations of BLAS and LAPACK provided by MKL 10.2.4. A modified version of SuperMatrix runtime in `libflame` version 5.0–r5587 was designed to leverage the two power-saving techniques described in the previous section. We evaluate execution times/power measurements of routine `FLASH_LU_piv` (blocked right-looking variant of the LU factorization with partial pivoting) from this library, linked to the original and power-aware implementations of the runtime. Our evaluation includes a variety of (square) matrix dimensions, ranging from 2048 to 12288, and the block size $b = 512$.

Power was measured using an internal DC powermeter. This is an ASIC operating with a sampling frequency of 25 Hz, directly attached to the lines connecting the power supply unit and the motherboard (chipset plus processors). All tests were repeated 30 times and average values are reported.

Our first experiment evaluates the existence and length of idle periods during the computation of the LU factorization with partial pivoting using 8 threads on the Intel processor, with parallelism extracted by the SuperMatrix runtime. Figure 2 reports the results from this evaluation. When the problem size is (n=)2048, during 54% of the time there is a single active thread and only 6% of the time all the available threads are effectively performing work (that is, executing tasks). On the other hand, when the problem size is much larger, e.g. n=10240, about 18% of the time there is one active and most of the remaining period all 8 threads are running. As a conclusion, there actually exists the opportunity of saving energy by carefully controlling the level of activity of idle threads as described in the previous section.

This observation is not effective unless our energy-aware approach actually yields energy gains. In our second experiment, we compare the original SuperMatrix runtime with a modified variant that employs semaphores to block idle threads (following the second technique described in the previous section). We leave the manipulation of core frequencies in the hands of the OS by setting the governor to `ondemand`, with the default policies to raise/lower frequency (namely, when the CPU load exceeds/falls below 95%, its frequency is set to the highest/lowest possible; the OS samples CPU activity with a frequency of 10 ms.). In this mode, a polling thread is active and, thus, the corresponding core/CPU remains at 2.0 GHz; a thread blocked in a semaphore, instead, is detected by the OS which lowers the operation frequency of the associated

CPU to the minimum available frequency (in this case, 1600 MHz). We will refer to these two versions of SuperMatrix as "polling" and "blocking".

Figure 3 illustrates the effect of the power-saving strategy from two perspectives: execution time and energy consumption. The impact on the execution time is small, with a maximum increase of 2% at most for some problem sizes while, for others, there is no appreciable difference between the two strategies or the blocking strategy is even more efficient. On the other hand, the effect on power efficiency is much more relevant. For the smallest problem sizes, the number of tasks is relatively low compared with the number of threads, which results in idle periods during the parallel execution; this translates into significant energy savings. These inactive periods are reduced as the problem dimension grows, and the power savings tend to stabilize around 6%.

Preliminary results for well-known DLA operations on multi-core and hybrid CPU-GPU architectures have demonstrated nonnegligible energy savings using naive scheduling policies. On multi-core processors, alternative combinations of waiting policies for idle threads (busy-wait or blocking) and frequency governors have been explored. On the SCC, similar qualitative results are expected. In addition, the power control capabilities of this processor open new challenges for novel scheduling policies and fine-grained power saving techniques. Our aim is to propose techniques general enough to cover the full functionality of the `libflame` library, attaining power savings with no impact on the programmability of the solution.

V. Retargeting the framework to the SCC chip

We have recently demonstrated how the techniques used in SuperMatrix (and thus, the whole runtime) can be effortlessly ported to a radically different architecture: the Intel SCC processor. The separation of concerns proposed by FLAME enables the application of both the methodology and the runtime implementation to this many-core chip, abstracting the programmer from the underlying architecture, and thus reusing the same codes independently from the target architecture. Two different approaches were adopted to port this tool to the SCC:

- View the SCC chip as a purely distributed memory architecture: we have developed an alternative implementation of the runtime targeting distributed-memory clusters, and successfully migrated it to the SCC processor using the RCCE communication library [8].
- View the SCC chip as a shared memory architecture: the FLAME team has developed a port of the SuperMatrix runtime targeting the many-core processor. In this approach, all matrix data are allocated using `RCCE_shmalloc` with uncacheable hijacked shared memory on SCC. The DAG is generated and tasks are scheduled to the different cores using the SuperMatrix scheduling algorithms [9].

The interesting point is that, in both cases, all the research performed for multi-core and hybrid CPU-GPU architectures was directly applicable to the SCC processor, without major conceptual changes. The capabilities of the SCC processor, in terms of power consumption control, make it the ideal testbed in which the combination of the runtime-based implementations of `libflame` and DVFS can be exploited to gain new insights to economize power on present multi-core and future many-core architectures.

VI. Conclusions

While CPU-bounded computations like, e.g., the matrix-matrix product should be run at the highest frequency so as to reduce execution time and, therefore, energy consumption, this paper addresses this issue for complex dense linear algebra operations, where idle periods appear during the execution of the corresponding algorithm due to data dependencies. In particular, we address the LU factorization with partial pivoting, and a parallel data-flow runtime-assisted (SuperMatrix) from a production library like `libflame`, to analyze the trade-off between performance and energy on a multi-core platform. Our results show that, for large problem sizes, it is possible to leverage these inactive periods, reducing energy consumption around 6% with a negligible impact on the execution time.

We consider the Intel SCC chip as an ideal framework for consolidating and extending the power-aware scheduling policies and programming techniques already developed and under investigation on multi-core and hybrid accelerated architectures. The port of the SuperMatrix framework to the SCC offers a rich variety benchmark of dense linear algebra implementations. We expect to demonstrate significant energy benefits in a transparent manner for the library developer.

Acknowledgments

The authors from the Universidad Jaume I were supported by project CICYT TIN2008-06570-C04-01 and FEDER.

References

[1] Susanne Albers. Energy-efficient algorithms. *Commun. ACM*, 53:86–96, May 2010.

[2] Wu-chun Feng, Xizhou Feng, and Rong Ce. Green supercomputing comes of age. *IT Professional*, 10(1):17 –23, jan.-feb. 2008.

[3] Ralf Gruber and Vincent Keller. One Joule per GFlop for BLAS2 Now! In Simos Theodore E., Psihoyios George, and Tsitouras Ch, editors, *AIP Conf. Proceedings*, volume 1281, pages 1321–1324. American Institute of Physics, 2010.

[4] Field G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.

[5] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah, Rob Springer, Barry L. Rountree, and Mark E. Femal. Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Trans. Parallel Distrib. Syst.*, 18:835–848, June 2007.

[6] D. King, I. Ahmad, and H.F. Sheikh. Stretch and compress based re-scheduling techniques for minimizing the execution times of DAGs on multi-core processors under energy constraints. In *International Conference on Green Computing*, pages 49–60. IEEE, 2010.

[7] P. Alonso, M.F. Dolz, R. Mayo, and E.S. Quintana-Ort. Improving power efficiency of dense linear algebra algorithms on multi-core processors via slack control. In *Workshop on Optimization Issues in Energy Efficient Distributed Systems (OPTIM), part of the International Conference on High Performance Computing & Simulation (HPCS), 2011, Istanbul, Turkey*, page To appear, 2011.

[8] Francisco Igual and Gregorio Quintana-Ortí. Solving linear algebra problems on distributed-memory computers using serial codes. Technical Report DICC 2010-07-01, Depto. de Ingeniería y Ciencia de Computadores. University Jaume I, July 2010.

[9] FLAME. http://z.cs.utexas.edu/wiki/flame.wiki/.

Investigation of Main Memory Bandwidth on Intel Single-Chip Cloud Computer

Nicolas Melot, Kenan Avdic and Christoph Kessler
Linköpings Universitet
Dept. of Computer and Inf. Science
58183 Linköping
Sweden

Jörg Keller
FernUniversität in Hagen
Fac. of Math. and Computer Science
58084 Hagen
Germany

***Abstract*—The Single-Chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. It comprises 48 x86 cores linked by an on-chip high performance network, as well as four DDR3 memory controllers to access an off-chip main memory of up to 64GiB. This work evaluates the performance of the SCC when accessing the off-chip memory. The focus of this study is not on taxing the bare hardware. Instead, we are interested in the performance of applications that run on the Linux operating system and use the SCC as it is provided. We see that the per-core read memory bandwidth is largely independent of the number of cores accessing the memory simultaneously, but that the write memory access performance drops when more cores write simultaneously to the memory. In addition, the global and per-core memory bandwidth, both writing and reading, depends strongly on the memory access pattern.**

I. Introduction

The Single-Chip Cloud Computer (SCC) experimental processor [1] is a 48-core “concept-vehicle” created by Intel Labs as a platform for many-core software research. Its 48 cores communicate and access main memory through a 2D mesh on-chip network attached to four memory controllers (see Figure 1).

Algorithm implementations usually make a more or less heavy use of main memory to load data and to store intermediate or final results. Accesses to main memory represent a bottleneck in some algorithms’ performance [2], despite the use of caches to reduce the penalty due to limited bandwidth to main memory. Caches are high-speed memories, close to processing units but are rather small and their effect is less visible when a program manipulates a larger amount of data. This leads to the design of other optimizations such as on-chip pipelining for multicore processors [2].

This work investigates the actual memory access bandwidth limits of SCC from the perspective of applications that run on the Linux operating system and use the SCC as it is provided to them. As thus, the focus is not what the bare hardware is capable of, but what the system, i.e. the ensemble of hardware, operating system and programming system (compiler, communication library, etc) achieves. Our approach is to use microbenchmarking to create different sets of patterns to access the memory controllers. Our experience indicates that the memory controllers can support all cores reading data from their private memory, but that the cores experience a significant performance drop when writing to main memory. For both read and write accesses, the available bandwidth is strongly dependent on the memory access pattern.

Section II introduces the SCC, then Section III describes the method used for stressing the main memory interface and discusses the results obtained. Finally Section IV concludes.

II. The Single Chip Cloud computer

The SCC provides 48 independent x86 cores, organized in 24 tiles. Figure 1 provides a global schematic view of the chip. Tiles are linked together through a 6×4 mesh on-chip network. Each tile embeds two cores with their cache and a message passing buffer (MPB) of 16KiB (8KiB for each core); the MPB supports direct core-to-core communication.

The cores are IA-32 x86 (P54C) cores which are provided with individual L1 and L2 caches of size 32KiB and 256KiB, respectively, but no SIMD instructions. Each link of the mesh network is 16 bytes wide and exhibits a 4 cycles crossing latency, including the routing activity.

The overall system admits a maximum of 64GiB of main memory accessible through 4 DDR3 memory controllers evenly distributed around the mesh. Each core is attributed a private domain in this main memory whose size depends on the total memory available (682 MiB in the system used here). Six tiles (12 cores) share one of the four memory controllers to access their private memory. Furthermore, a part of the main memory is shared between all cores; its size can vary up to several hundred megabytes. Note that private memory is cached on cores’ L2 cache but caching for shared memory is disabled by default in Intel’s framework RCCE. When caching is activated, the SCC offers no coherency among cores’ caches to the programmer. This coherency must be implemented through software methods, by flushing caches for instance.

The SCC can be programmed in two ways: a baremetal version for OS development, and using Linux. In the latter setting, the cores run an individual Linux kernel on top of which any Linux program can be loaded. Also, Intel provides the RCCE library which contains MPI-like routines to synchronize cores and allow them to communicate data to each other. RCCE also allows the management of voltage and frequency scaling.

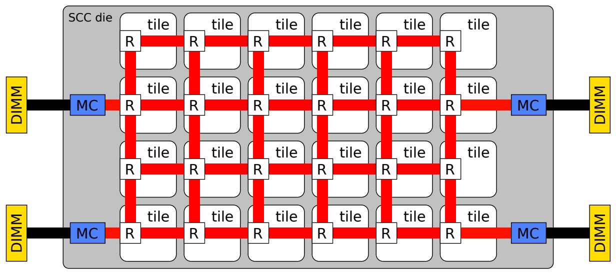


Figure 1. A schematic view of the SCC die. Each box labeled DIMM represents 2 DIMMs.

```
/* SIZE is a power of two
 * strictly bigger than L2 cache
 */
int array[SIZE];

void memaccess ( int stride )
{
  int i, j, tmp;

  for (j = 0; j < SIZE; j += stride)
    tmp = array[j];
}
```

Figure 2. Pseudo-code of the microbenchmark for reading access. For writing, the order of the variables in the assignments is exchanged.

III. Experimental evaluation

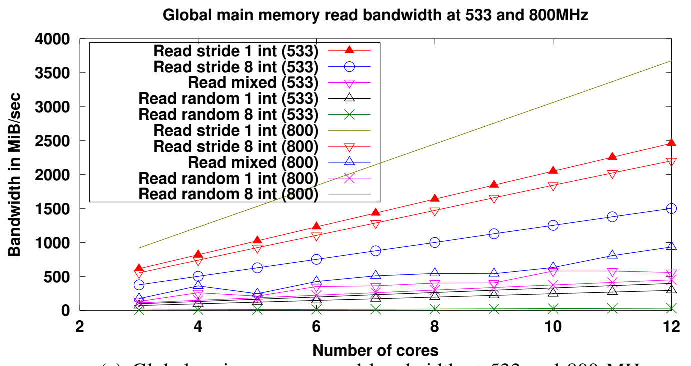
The goal of our experiments consists in the measurement of the bandwidth available to an application that runs on top of the Linux operating system in standard operating conditions (cores at 533 MHz, on-chip network at 800 MHz, memory controllers at 800 MHz). Furthermore, we are interested in how this bandwidth varies with the number of cores performing memory operations and the nature of the operations themselves, read or write. This is achieved by consecutively reading respectively writing the elements of a large array of integers, aligned by 32 bytes which is the size of a cache line. Thus, consecutive access to all integers (1-int-stride, 4-byte-stride) yields perfect spatial locality whereas 8-int-strided access (4 out of 32 bytes) to the data always results in a cache miss. Each participating core runs a process that executes a program as depicted in Fig. 2, where each array is located in the respective core's private memory and through which the cores iterate exactly once.

While the 1-int-strided and 8-int-strided memory accesses stresses the bandwidth difference due to cache hits and cache misses, the *random* access pattern stresses the memory controllers' throughput using a random access, making helpless its hardware optimizations that parallelize or cache read or write accesses, such as using a plurality of open rows in the attached SDRAMs. To simulate random access, the array is accessed through a function $pi(j)$ that is bijective in $\{0,\ldots,SIZE-1\}$, where j is same index (strided 1 int or 8 ints) used to access the array in the strided access described above. In practice, we use $pi(j) = (a \cdot j) \mod SIZE$ for a large, odd constant a where *SIZE* is a power of two and the size of the array to be read. The random access pattern also applies the 1-int-strided, 8-int-strided and mixed patterns described above to the index j.

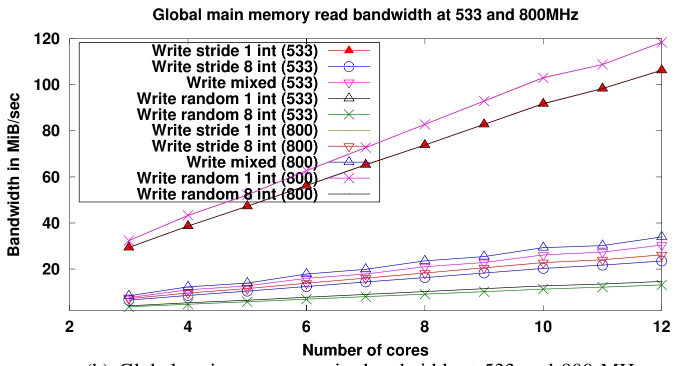
Finally strided, mixed and random access make all the cores read or write at the same time, along the different access patterns they define. All these patterns also combine read and write operations, one half of processors performing reads, and the second half performing writes. This is denoted as the *combined* access pattern.

In this experiment, a varying number of cores synchronize, then iterate through the array to read or write as described above. Since every memory operation leads to a cache miss in the 8-int-strided access and random access reduces the memory controllers' performance, such memory operations generate traffic and the time necessary to read the targeted amount of data allows the calculation of the actual bandwidth that was globally available to all cores. The amount of data to be read or written by each core is fixed to 200MiB. 3 to 12 cores are used, as up to twelve cores share the same memory controller. Cores run at 533 MHz and 800 MHz in two different experiments, while the mesh network and memory controllers remain both at 800MHz. The global bandwidth and the bandwidth per core are measured: the global bandwidth represents the bandwidth a memory controller provides to all the cores. The bandwidth per core is the bandwidth a core gets when it shares the global bandwidth with all other running cores. Figures 3, 4 and 5 show the global and per core bandwidth measured in our experiments.

Figure 3 indicates that both read and write bandwidth are linearly growing with the number of cores. Since the SCC provides no hardware mechanism to manage and share the memory bandwidth served to cores, this shows that all cores together still fail to saturate the read memory bandwidth available. The random access pattern offers a much lower read throughput around 250MiB/sec with 12 cores running at both 533 and 800 MHz. The write throughput for random stride 1 shows the same performance as write stride 1 (up to 105 and 120MHz respectively at 533 and 800MHz) and other write patterns do not exceed 20MiB/sec nor about 7MiB/sec for random stride 8 access pattern. This shows that memory controllers struggle to serve irregular main memory request patterns. The absolute numbers of read bandwidth per core in the 1-int-stride experiment are stable around 205 MiB/s and around 125 MiB/s with the 8-int-stride access pattern with cores running at 533 MHz and respectively 305 and 235 MiB/sec at 800 MHz, as shown in Fig. 4(a). However, the bandwidth per core with the write accesses (Fig. 4(b)) drops with the number of cores from 10 MiB/sec with 3 cores to 9 MiB/sec using 12 cores at 533 MHz and from 11 MiB/sec to 10 MiB/sec at 800MHz. The P54C's L1 cache no-allocate-on-write-miss behavior may explain this performance drop: as write cache misses do not lead to a cache line allocation, every consecutive write results in a write request addressed to the memory controller. In both cases, the low difference

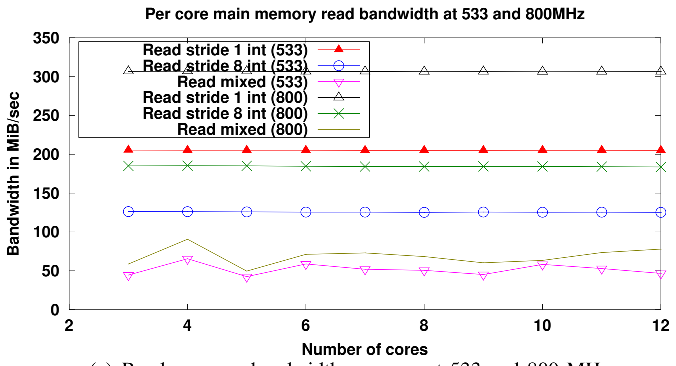


(a) Global main memory read bandwidth at 533 and 800 MHz.

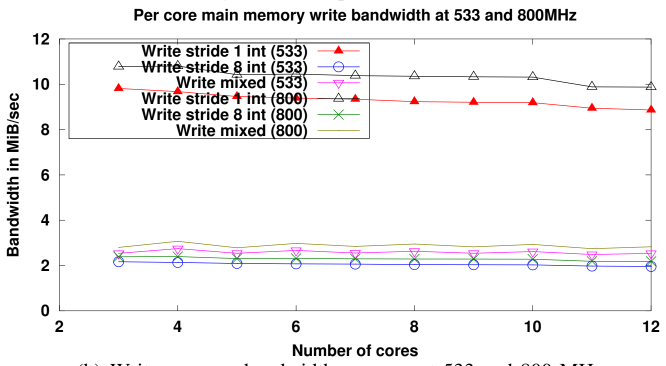


(b) Global main memory write bandwidth at 533 and 800 MHz.

Figure 3. Measured global memory read and write bandwidth as a function of the number of cores involved, at 533 and 800 MHz.



(a) Read memory bandwidth per core at 533 and 800 MHz.



(b) Write memory bandwidth per core at 533 and 800 MHz.

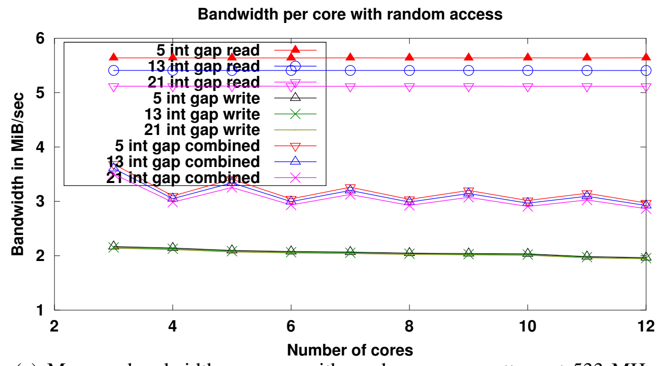
Figure 4. Measured per-core memory bandwidth as a function of the number of cores involved, for strided access patterns, at 533 and 800 MHz.

in performance of 1-int-stride and 8-int-stride access patterns shows that the high performance memory controllers are able to compensate efficiently the performance losses due to cache misses. However the mixed access pattern, with one half of the cores reading memory with a 1-int-stride and the second half with 8-int-stride, exhibits lower performance, which shows again the limited capabilities of memory controllers to serve irregular access patterns.

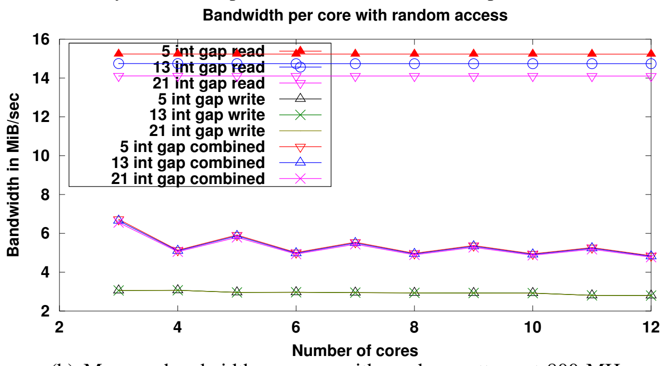
The bandwidth measured per core for the random access pattern reveals better performance with faster cores.

IV. Conclusion

The memory wall represents an important performance limiting issue still present in multicore processors, and implementations of parallel algorithms are still heavily penalized when accessing main memory frequently [2]. This work enlightens the available memory bandwidth on Intel's Single Chip Cloud Computer when several processors perform concurrent read and write operations. The measurements obtained here and the difficulty we experience to actually saturate the read memory bandwidth show that the cores embedded in the SCC cannot saturate all together the read memory bandwidth available: for read access patterns behave regularly, the cores cannot saturate. However, the measurements obtained from the write access patterns demonstrate a much smaller write bandwidth available. Also, we can note that the available bandwidth for both read and write strongly depends on the memory access pattern, as the low bandwidth on random



(a) Memory bandwidth per core with random access pattern at 533 MHz.



(b) Memory bandwidth per core with random pattern at 800 MHz.

Figure 5. Measured per-core memory access bandwidth as a function of the number of cores, for random access patterns, at 533 and 800 MHz.

access patterns indicates. Thus, there is no point in reducing the degree of parallelism in order to increase the available bandwidth for tasks requiring a high main memory bandwidth. The measurements shown in the paper show a behavior possibly adapted to program restructuring techniques such as on-chip pipelining and our previous implementation of on-chip pipelined mergesort [2]. In this implementation, many tasks mapped to several cores fetch input data in parallel from main memory, and a unique task running on a unique core writes the final result back to main memory, therefore limiting expensive main memory accesses. However, the gap between the memory bandwidth available and the limited capabilities of cores to saturate it shows that there is room to add more cores, run them at higher frequency or add SIMD ISA extensions. Without such improvements in the cores' processing speed and accordingly higher demands on memory bandwidth, our ongoing research on program restructuring techniques such as on-chip pipelining is, for SCC, limited to implementation studies leading to predictions of their theoretical speed-up potential, rather than demonstrating concrete speed-up on the current SCC platform. Such techniques could speed up memory-access intensive computations such as sorting [2], [3] on SCC-like future many-core architectures that are more memory bandwidth constrained.

Acknowledgments

The authors are thankful to Intel for providing the opportunity to experiment with the "concept-vehicle" many-core processor "Single-Chip Cloud Computer". We also thank the anonymous reviewers for their helpful comments on an earlier version of this paper.

This research is partly funded by the Swedish Research Council (Vetenskapsrådet), project *Integrated Software Pipelining*.

References

[1] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart, "A 48-Core IA-32 message-passing processor in 45nm CMOS using on-die message passing and DVFS for performance and power scaling," *IEEE J. of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[2] R. Hultén, J. Keller, and C. Kessler, "Optimized on-chip-pipelined mergesort on the Cell/B.E." in *Proceedings of Euro-Par 2010*, vol. 6272, 2010, pp. 187–198.

[3] K. Avdic, N. Melot, J. Keller, and C. Kessler, "Parallel sorting on Intel Single-Chip Cloud Computer," in *Proc. A4MMC workshop on applications for multi- and many-core processors at ISCA-2011*, 2011.

Towards Resource-Aware Programming on Intel's Single-Chip Cloud Computer Processor

Georgia Kouveli, Frank Hannig, Jan-Hugo Lupp, and Jürgen Teich
Hardware/Software Co-Design,
Department of Computer Science,
University of Erlangen-Nuremberg, Germany.

***Abstract*—In this paper, we apply a new programming paradigm called resource-aware programming to the Single-Chip Cloud Computer. According to this paradigm, an application may change at certain points of execution its allocation of resources. This gives application engineers the opportunity to dynamically adapt an algorithm's behavior and parallelism to the work load and state of the underlying resources (e. g., availability, clock frequency, temperature). Resource-aware programming can provide a self-organizing behavior to conventional programs for being able to not only tolerate certain types of faults and cope with feature variations, but also to provide scalability, higher resource utilization, as well as performance and power gains by managing voltage/frequency islands and adjusting the amount of allocated resources to the temporal needs of a running application. We discuss the details of resource-aware programming as well as three alternative implementation concepts that we intend to evaluate. Finally, we present the results of initial experiments, we conducted using a centralized resource management framework on the Single-Chip Cloud Computer.**

I. Introduction

As transistor feature sizes keep shrinking, billions of transistors can be implemented in a single chip, which has led to a shift of processor architecture trends towards massively parallel processors. Multi-core and many-core architectures are now mainstream, either in the form of desktop and server processors, or multiprocessor System-on-Chip (MPSoC) designs used in embedded devices. By 2020, packing more than 1000 cores in a single chip will be possible due to technology scaling.

With the availability of these many-core architectures, a number of challenges arise for the application programmers to face. Among these challenges is the mapping of algorithms and programs to the numerous available processors. Current system support for distribution of resources among the different applications executing on a chip is expected not to scale well for thousands of processors. Applications will need to become adaptive to run efficiently without change on a variable set of processors, depending on their availability during the application's run-time. Applications will also face transient and permanent faults, which will become more evident as the number of cores on a chip increases. All these requirements call for new programming paradigms, which will allow us to efficiently exploit available resources and increase programmer's productivity.

The concept we envision as an answer to the aforementioned challenges is a new programming paradigm based on resource-aware programming. The core idea of resource-aware programming, as the name suggests, is making the application aware of the status and the availability of the underlying resources and giving it control over resource allocation and deallocation. In the resource-aware programming paradigm, a given program is able to explore the available resources and distribute its computations to neighboring processors, thus executing code sections with high degrees of parallelism in parallel on the number of available processing elements. The program can dynamically claim or release resources to adjust to changes in its degree of parallelism or to the status and availability of resources on the chip.

Resource-aware programming can enable us to tackle the challenges of many-core programming. The self-organizing behavior of resource-aware applications can lead to optimal utilization of resources, as their availability varies during run-time. The performance of individual applications can be boosted by discovering and allocating additional available resources while the application is running, in order to adjust to its varying degrees of parallelism. The overall processor utilization is also greatly improved by the fact that applications explicitly release resources they no longer need. Resource-aware programming can also result in improved power consumption, by managing voltage/frequency of the processing elements to meet application computational needs. Moreover, monitoring temperature statistics and adjusting workload accordingly can lead to a decrease of faults to overheating. Keeping track of resource status can also enhance the fault tolerance of applications.

We believe the concept of resource-aware programming will be of high importance in the design of future programming languages and programming environments for the development of parallel programs. Providing the programmer with an interface to manage resources explicitly will in the future be of high importance for algorithms running on many-core platforms, since it gives algorithm designers the opportunity to dynamically adapt an algorithm's behavior and parallelism to the dynamic features of the platform. Even though "forcing" the programmer to keep details about the underlying platform in mind when programming seems counter-intuitive, experience from parallel programming so far has shown that the best implementations of parallel algorithms consist of low-level, hand-optimized code. One can also argue that the concepts of resource-aware programming do not need to be exposed directly to the programmer, but can be exploited by operating systems and parallelizing compilers to automatically generate good quality parallel code.

The Single-Chip Cloud Computer (SCC) experimental processor [1] is a 48-core "concept vehicle" created by Intel Labs as a platform for many-core software research, and as such it provides the ideal platform for our experimentation with resource-aware programming. The large number of available cores and provided capabilities such as dynamic voltage/frequency scaling and temperature monitoring provide us with the opportunity to explore the benefits of resource-aware programming for future many-core platforms.

In this paper, we present our first thoughts and initial experimentation with resource-aware programming on the SCC. In Section II, we present previous work related to the concept of resource-aware programming, in Section III we discuss the application of resource-aware programming to the SCC and the necessary changes that are required in the supplied programming models in order to support it and in Section IV we describe our initial experimentation with the concepts of resource-aware programming on the SCC. Finally, Section V concludes the paper with our plans for future work.

II. Related work

In the CAPSULE project [2], the authors describe a component-based programming paradigm combined with hardware support for processors with simultaneous multi-threading in order to handle the parallelism in irregular programs. Here, an application is dynamically parallelized at run-time. A pure software version of CAPSULE, demonstrated on an Intel Core 2 Duo processor is presented in [3]. In the TRIPS project [4], an array of small processors is used for the flexible allocation of resources dynamically to different types of concurrency, ranging from running a single thread on a logical processor composed of many distributed cores to running many threads on separate physical cores. However, these approaches do not touch the major problems of algorithmic design. In [5], Henkel and others proposed a run-time agent-based mapping approach that minimizes thermal hotspots through power distribution for general multi-core architectures. Power management on a concrete tiled architecture, namely the SCC, is part of the current research by Cintra and others [6].

Our ideas presented in this paper are strongly related to the concepts of *invasive computing* [7], a new paradigm for resource-aware computing that integrates research on MPSoC architectures, compilers, simulation, applications, and run-time support. A first framework for resource-aware programming and the simulation of MPSoC architectures through extensions of the programming language X10 [8] has been recently proposed in [9]. X10 is also currently ported to the SCC by Hosking and others [10].

III. Resource-Aware Programming on Intel SCC

The SCC provides an ideal platform for experimentation with the concepts of resource-aware programming. It consists of 48 cores, each one of which is fully capable of running a full operating system, and provides hardware support for message passing, as well as shared memory. It is, therefore, very well suited to many-core software research. A resource-aware program running on the SCC has the possibility to monitor dynamic state information such as processor utilization, temperature (via the temperature sensors located on each tile), voltage and frequency, therefore adapting its resource usage on these statistics, and use frequency/voltage scaling, to optimize power consumption according to the variable needs of the application.

A. Resource-aware program phases

The execution of any resource-aware program can be broken down to phases that can be roughly categorized in three types: resource allocation, execution, and resource deallocation (see Fig. 1). Usually, a program allocates resources, executes its code using those resources and at the end of execution deallocates them. This is represented by solid lines. In a resource-aware

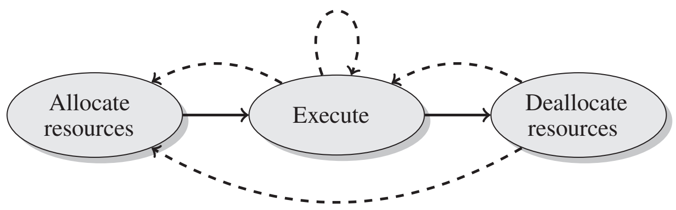


Fig. 1: Phases of execution of a resource-aware program.

program, however, there can exist more than one of each type of phase. For example, a program that needs a lot of computational resources can allocate more resources as these become available, or allocate more resources only for one particular part of its code with high degree of parallelism and then deallocate them before continuing the execution of less demanding code. These cases where the execution of a resource-aware program might differ from a simple parallel program are represented by dashed lines in Fig. 1. An example of what a simple resource-aware application might look like is given in pseudocode in Listing 1.

Listing 1: Example pseudocode for a simple resource-aware application

```
/* Request resources */
claim = request_resources(type, quantity, constraints);

if (successful(claim)) {
        /* Execute on allocated resources */
        execute(claim, code, data);

        /* Release resources */
        release(claim);
} else {
        execute_sequentially();
}
```

1) Resource allocation: In this phase, a program claims the resources it needs for the following execution phase, and depending on their status and availability, allocates them. The simplest form of request a program can make is for a given number of cores for exclusive use. For this type of request, the only information that needs to be maintained by our resource management system is one bit of information for each core, denoting whether it is idle or occupied. Building on this, more complicated requests can be formed, for example for non-exclusive use of a number of cores, given that their workload remains under a given percentage. Other possibilities for constraints the program might impose on the requested resources can be related to the temperature of the cores, or to their frequency, e.g., for the execution of computationally intensive parts of the program, it might request cores of the highest available frequency.

2) Execution phase: After allocating the necessary resources, the program needs to execute parts of its code utilizing those resources. This phase is the most straightforward one, however, it poses some significant implementation challenges. The publicly available programming models for the SCC provide no dynamic process management, which is essential to the concept of resource-aware programming. The RCCE library does not allow the dynamic creation of processes to execute on other cores. As a matter of fact, it does not even allow the execution of two RCCE processes on the same core, since each one of them uses the entirety of the on-chip shared Message Passing Buffer (MPB) memory that corresponds to each core [11]. In order to apply the concept of resource-aware programming to

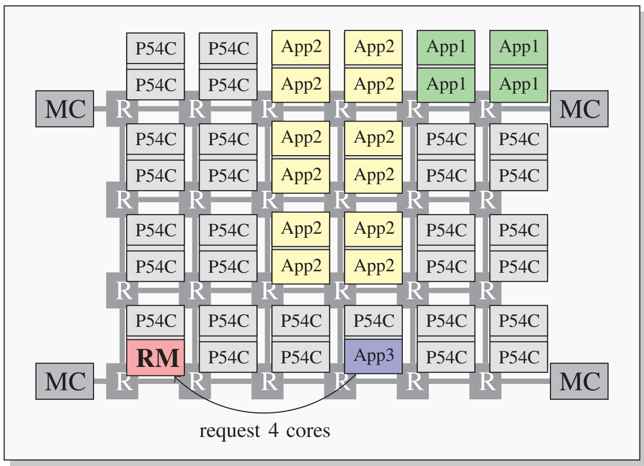


Fig. 2: Centralized resource management: Resource allocation/deallocation requests from applications are forwarded to and serviced by the Resource Manager (RM).

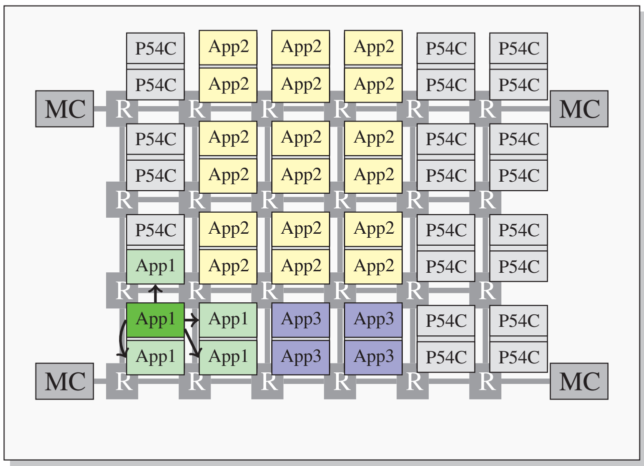


Fig. 3: Distributed resource management: Resource allocation/deallocation requests from applications are immediately directed to the requested cores.

the SCC, there needs to be, therefore, a major design change in the RCCE library.

3) Resource deallocation: In this phase, the program deallocates resources it does not need anymore, due to a decrease in its degree of parallelism. This phase, although straightforward, is very important, as releasing resources that do not need to be allocated to the program can lead to significant improvements in resource utilization, in particular when applications are competing for exclusive use of resources.

Our ultimate goal is to provide a library implementation that supports these basic constructs of resource-aware programming for programs executing on the SCC.

B. Centralized vs. distributed resource management

In order to enable resource-aware programming on the SCC, a resource management framework has to be implemented that provides the basic functionality described in the previous subsection. We intend to experiment with two main different approaches for resource management, one centralized and one distributed approach, as well as combinations of these two.

1) Centralized approach: In the case of centralized resource management, there is one "master" process that manages resource availability and status information and services all application requests for resource allocation/deallocation (see Fig. 2). Whenever an application needs to allocate or deallocate resources, it forwards a request to the "master" process (central resource manager), which receives it and composes a so-called claim of assigned resources as a response, depending on the status and availability of resources on the chip. This is the approach towards which our initial experiments are directed, since a centralized approach is simpler and, therefore, easier to implement. Moreover, since the "master" process can access all information on resource status and availability, it can make better choices for the allocation of resources than if it only had access to information regarding its neighborhood. However, a centralized approach is expected to suffer from bottlenecks and is more prone to faults. For example, if all applications running on the SCC keep making frequent requests to the central resource manager, performance will be seriously degraded. Still, for research purposes, it is interesting to implement a centralized approach, which can later serve as a reference implementation to compare our more advanced implementations with.

2) Distributed approach: In contrast to the centralized approach, there exists no central resource manager to service requests for resource allocation/deallocation, instead these requests are directly forwarded to the requested resources. Various alternatives can be implemented here. For example, an application executing on one core might claim its neighboring cores in a given direction (e.g., "north"), or it can be allowed to request usage of particular cores identified by their coordinates in the processor grid. Fig. 3 shows how an application (App1) could request the allocation of its neighboring cores to it directly.

To make the best out of a distributed approach, information should also be maintained in a distributed way. Thus, the bottlenecks that arise in a centralized approach will be eliminated. A distributed approach also offers better fault tolerance, since in case one processor fails, the rest of the system can still function, as opposed to the case of failure of a centralized resource manager. However, a distributed approach is more complicated to implement. Information is maintained non-locally and an application lacks a general view of status and availability of resources. Therefore, more time will be needed to access necessary information and make decisions on resource allocation, which will tend to be suboptimal.

3) Hybrid approach: The previous two approaches can also be combined in a hybrid approach, where there exist more than one resource manager process, each one managing an "island" of tiles/cores. One can experiment with varying granularity for these "islands". One resource manager per core gives us the fully distributed approach, whereas regarding all the cores as one "island" gives us the centralized approach. Thus we can combine the best characteristics of the two approaches, namely the simplicity and optimal decisions on resource allocation that a centralized approach provides, and fault tolerance and absence of bottlenecks, as in the case of a distributed approach.

IV. Initial Experimentation

For our initial experiments, we chose to explore the implementation of a centralized resource manager to enable resource-aware programming on the SCC. In our first experiments, the resource management framework consists of a "manager" thread and a pool of "worker" threads, one per core, which communicate with each other using the RCCE library, which implements message passing over the on-chip shared Message Passing Buffer (MPB) memory. The "manager" distributes pieces of work, which initially are restricted to single-processor programs, to the "workers", by sending them an appropriate message. Information about the utilization of tiles (idle/occupied) is also stored in the on-chip shared MPB memory, provided by the gory version of the RCCE API. Once a "worker" has finished execution of the work assigned to it, it updates its flag to denote that it is no longer occupied. In this sense, utilization information is stored and updated in a distributed way, despite the presence of a central resource manager that allocates tasks to cores.

Our initial experiments helped us to realize the restrictions and weaknesses of the currently available programming models for the SCC and determine our next steps. One of the most important restrictions of the RCCE library was that there was no way to start the execution of a parallel RCCE application from within our resource management framework, since the framework is also using the RCCE library, which makes the assumption that the whole MPB memory is available to the application and overwrites all previously stored data [11], including the status of the resources and the buffers we use to communicate with the "workers". For this reason, we modified the original RCCE library to reserve a part of the on-chip shared Message Passing Buffer memory for our resource management purposes and use the rest for normal message passing between applications. We provide special functions to access the dedicated parts of the MPB memory, as well as to initialize these parts of the memory (see Listing 2).

Listing 2: Interface for resource management functions

```
/* Initialize resource management framework */
int RCCE_resource_aware_init(int *argc, char***argv);

/* Functions to modify busy flag */
int RCCE_read_busy_flag(int *value, int rank);
int RCCE_set_busy_flag(int rank);
int RCCE_clear_busy_flag(int rank);

/* Function to wait until work is assigned to
   the worker */
void RCCE_wait_for_work();

/* Functions to access the buffer containing the command
   to be executed by the worker */
char *RCCE_read_cmd_buffer(char *str, int rank);
int RCCE_write_cmd_buffer(char *str, int rank);
```

Another significant change is the integration of the shared memory initialization process in the initialization of the RCCE library. This was previously done separately, by executing a completely different application to initialize the MPB memory to a known state, before the execution of the RCCE application. We modified this to facilitate the execution of RCCE applications from the "workers". For this purpose, we implemented a tournament algorithm to initialize shared memory and synchronize, based on [12]. Thus, the application code does not need to be changed, but the initialization is performed within the library.

Another important restriction is that we cannot dynamically offload simply one piece of code, e.g., one function, for execution on another processor, since every core is running a completely different instance of the operating system and the executed image does not reside in the shared memory. This must also be taken into consideration when defining the tasks that a resource-aware application can offload to newly allocated cores.

V. Conclusions

In this paper, we discussed the benefits of applying resource-aware programming to many-core architectures, including increased resource utilization, performance, power consumption and fault tolerance. We also described the application of resource-aware programming on the SCC, our current work in progress, which will help us fully exploit the potential of this architecture. We presented the experience gathered from our early experimentation with resource management on the SCC, regarding the restrictions and limitations of the currently available programming model, and our first steps towards lifting those limitations and enabling resource-aware programming.

VI. Acknowledgments

The authors would like to thank the Intel Corporation for providing access to the Single-Chip Cloud Computer and the excellent assistance. The conceptual design of the paradigm of resource-aware programming was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, feb. 2010, pp. 108 –109.

[2] P. Palatin, Y. Lhuillier, and O. Temam, "CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Orlando, Florida, USA, 2006, pp. 247–258.

[3] O. Certner, Z. Li, P. Palatin, O. Temam, F. Arzel, and N. Drach, "A Practical Approach for Reconciling High and Predictable Performance in Non-Regular Parallel Programs," in *Proceedings of Design, Automation and Test in Europe (DATE)*, Munich, Germany, 2008, pp. 740–745.

[4] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable Lightweight Processors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Chicago, Illinois, USA, 2007, pp. 381–394.

[5] T. Ebi, M. A. A. Faruque, and J. Henkel, "TAPE: Thermal-Aware Agent-Based Power Economy for Multi/Many-Core Architectures," in *Proceedings of the 27th IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, California, USA, 2009, pp. 302–309.

[6] N. Ioannou and M. Cintra, "Application-Driven Power Management on the Single-Chip Cloud Computer," Nov. 2010, Talk, 1st MARC Symposium, Intel Braunschweig, Germany.

[7] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive Computing: An Overview," in *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds. Springer, 2011, pp. 241–268.

[8] G. Bikshandi, J. Castanos, S. Kodali, V. Nandivada, I. Peshansky, V. Saraswat, S. Sur, P. Varma, and T. Wen, "Efficient, Portable Implementation of Asynchronous Multi-Place Programs," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Raleigh, NC, USA: ACM, 2009, pp. 271–282.

[9] F. Hannig, S. Roloff, G. Snelting, J. Teich, and A. Zwinkau, "Resource-Aware Programming and Simulation of MPSoC Architectures through Extension of X10," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar, Germany: ACM Press, Jun. 2011.

[10] K. Chapman, A. Hussein, and A. Hosking, "X10 on the Single-Chip Cloud Computer," in *Proceedings of the X10 Workshop co-located with 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, USA: ACM Press, Jun. 2011.

[11] R. Van der Wijngaart and T. Mattson and W. Haas, "Light-weight Communications on Intel's Single-Chip Cloud Computer Processor," *Operating Systems Review*, vol. 45, no. 1, pp. 73–83, 2011.

[12] Hemmendinger, "Initializing memory shared by several processors," *IJPP: International Journal of Parallel Programming*, vol. 18, 1989.

ISBN 978-3-86644-717-2

ISSN 1869-9669
ISBN 978-3-86644-717-2