Karlsruher Institut für Technologie

Scalable Task Schedulers for Many-Core Architectures

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Anuj Pathania

aus Neu Delhi, Indien

Tag der mündlichen Prüfung:	16.05.2018
Referentin:	Prof. Dr.-Ing. Jörg Henkel, KIT
Korreferent:	Prof. Dr. Tulika Mitra, NUS

Anuj Pathania
Schumacher-Str.11
76275 Ettlingen

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen, Karten und Abbildungen — die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Anuj Pathania

Acknowledgments

First and foremost, I would like to express my sincere gratitude to Prof. Henkel for providing me an opportunity to pursue my doctorate in his prestigious research group. His impeccable supervision guided me swiftly through this arduous yet immensely rewarding graduate school journey. I am also deeply grateful to Prof. Mitra and Prof. Shafique for helping me along the way. I also want to acknowledge support of all my past and present colleagues at KIT.

I would like to thanks German Research Foundation (DFG) who funded me under *Invasive Computing* (SFB/TR 89) project during my studies. I would also like to thank my defense committee – Prof. Tahoori, Prof. Beigl, Prof. Hartenstein and Prof. Wagner – for reviewing my thesis and attending my defense.

I am also deeply indebted to my family all of whom made immense sacrifices for my success. My wife Kriti who cooperated through those endless hours of work and never-ending deadlines. My father Chander who inspired me to dream more and never be satisfied. My mother Pradnya who always believed in me. My sister Abha and brother-in-law Anand who invariably had my back. Finally, my son Sanchay who filled my life with infinite joy.

Last but not least, I would like to sincerely thank each and every person who contributed in making me the person I am today. I am mindful of all the time and resources that were invested in me, which very few people in this world ever receive. I hope in future I can give back to the society more than I have received.

List of Publications

Following list enumerates conference and journal papers published by the author of this dissertation while pursuing studies as a Ph.D. scholar.

[1] A. Pathania, S. Pagani, M. Shafique and J. Henkel, "Power Management for Mobile Games on Asymmetric Multi-Cores," in *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2015.

[2] A. Pathania, V. Venkatramani, M. Shafique, T. Mitra and J. Henkel, "Distributed Fair Scheduling for Many-Cores," in *Proceedings of Design, Automation & Test in Europe (DATE)*, 2016.

[3] A. Pathania, V. Venkatramani, M. Shafique, T. Mitra and J. Henkel, "Distributed Scheduling for Many-Cores Using Cooperative Game Theory," in *Proceedings of the 53rd Design Automation Conference (DAC)*, 2016. **[HiPEAC Award]**

[4] A. Pathania, V. Venkatramani, M. Shafique, T. Mitra and J. Henkel, "Optimal Greedy Algorithm for Many-Core Scheduling," in *IEEE Transactions on Computer-Aided Design of Integrated Circuit and Systems (TCAD)*, 2016.

[5] S. Pagani, L. Bauer, Q. Chen, E. Glocker, F. Hanning, A. Herkersdorf, H. Khdr, A. Pathania, U. Schlichtmann, D. Schmitt-Landsiedel, M. Sagi, E. Sousa, P. Wagner, V. Wenzel, T. Wild and J. Henkel, "Dark Silicon Management: An Integrated and Coordinated Cross-Layer Approach," in *it-Information Technology (IT)*, 2016. (Invited Paper)

[6] A. Pathania, V. Venkatramani, M. Shafique, T. Mitra and J. Henkel, "Defragmentation of Tasks in Many-Core Architecture," in *IEEE Transactions on Architecture and Code Optimization (TACO)*, 2017.

[7] A. Pathania, H. Khdr, M. Shafique, T. Mitra and J. Henkel, "Scalable Probabilistic Power Budgeting for Many-Cores," in *Proceedings of Design, Automation & Test in Europe (DATE)*, 2017. **[Best Paper Nomination]**

[8] S. Pagani, A. Pathania, M. Shafique, J.-J. Chen and J. Henkel, "Energy Efficiency for Clustered Heterogeneous Multicores," in *IEEE Transaction on Parallel and Distributed Systems (TPDS)*, 2017.

[9] H. Khdr, S. Pagani, E. Sousa, V. Lari, A. Pathania, F. Hanning, M. Shafique, J. Teich and J. Henkel "Power Density-Aware Resource Management for Heterogeneous Tiled Multicores," in *IEEE Transactions on Computers (TC)*, 2017.

[10] A. Pathania and J. Henkel, "Task Scheduling for Many-Cores with S-NUCA Caches," in *Proceedings of Design, Automation & Test in Europe (DATE)*, 2018.

[11] A. Pathania, H. Khdr, M. Shafique, T. Mitra and J. Henkel, "QoS-Aware Stochastic Power Management for Many-Cores," in *Proceedings of the 55th Design Automation Conference (DAC)*, 2018.

[12] M. Rapp, A. Pathania, and J. Henkel, "Pareto-Optimal Power- and Cache-Aware Task Mapping for Many-Cores with Distributed Shared Last-Level Cache," in *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2018.

[13] V. Venkatramani, A. Pathania, M. Shafique, T. Mitra and J. Henkel, "Scalable Dynamic Task Scheduling on Adaptive Many-Core," in *Proceedings of the IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2018. (Invited Paper)

Abstract

Many-core processors contain a large number of cores housed on a single die and can execute multitudes of tasks in parallel. Many-cores are quintessential in several emerging high-performance computing tasks on embedded systems. The accompanying qualitative schedules are the key in achieving the full potential of many-cores for these tasks.

Schedulers are low-level Operating System (OS) sub-routines which develop task execution schedules on processors at run-time. Many-core schedules are several times bigger than existing multi-core schedules. Therefore, we require new schedulers capable of scaling up with the increase in the size of many-cores while preserving the schedule's quality. Many-cores also have several new micro-architectural features and schedulers must exploit them for a more efficient hardware-software co-design of schedules.

A scheduler may have different objectives depending upon the overlying system requirements. This dissertation introduces several different schedulers catering to these objectives. Unfortunately, most of the objectives have NP-hard complexity and hence cannot be achieved in polynomial time for a general case unless P=NP. While most of our peers have tried to attain them heuristically, we attain them using different non-heuristic schedulers. The schedulers provide strong theoretical guarantees on the schedule quality. The schedulers are also designed to be highly scalable and are structurally better suited for further improvements in future.

We present two schedulers for maximizing the performance of many-cores. First is a distributed scheduler, while the second one is a centralized greedy scheduler. Schedulers provide optimal many-core performance by moving allocated cores amongst tasks at run-time.

Maximization of performance is not the objective of all systems and some systems prioritize fairness over performance. We also present a distributed scheduler for maximizing the fairness of many-cores. Scheduler improves overall fairness by moving allocated cores amongst tasks, and results in a fairer schedule than state-of-the-art. Furthermore, the scheduler can maximize fairness optimally particularly in the case of fixed performance.

Performance can be increased even by just spatially rearranging the allocations without any actual change in their sizes. We present a distributed scheduler that defragments the many-core to improve its performance. The scheduler performs optimal defragmentation particularly in the case when all the tasks produce only power-of-two threads outperforming state-of-the-art.

The above-proposed scheduler does not work for a many-core with Static Non-Uniform Cache Access (S-NUCA) caches. We, therefore, introduce a new scheduler for this many-core which exploits topology-introduced heterogeneity in the cores of many-core due to the presence of S-NUCA caches to improve performance. The scheduler uses an exact algorithm on the pruned search-space to quickly develop a schedule and outperforms state-of-the-art.

Many-cores operate within a strict power budget which constraints their performance. We developed a centralized scheduler for probabilistic power budgeting with only linear-time computational complexity. The scheduler performs power budgeting a magnitude faster than state-of-the-art with near-equal performance. On similar lines, we also developed a probabilistic power budgeting scheduler for many-cores when they execute Quality of Service (QoS) tasks.

Finally, we also develop an open-source toolchain called *HotSniper* for real-world representative evaluations of many-core schedulers. *HotSniper* integrates *Hotspot* thermal modeling tool with *Sniper* many-core simulator. *HotSniper* also allows interval thermal simulations of many-cores deployed in open systems, which were not previously possible.

Zussammenfassung

Many-core-Prozessoren beherbergen eine Vielzahl an Prozessorkernen auf einem einzelnen Chip und sind damit in der Lage, eine große Zahl an Anwendungen parallel auszuführen. Daher sind Many-cores unumgänglich für High Performance Computing auf eingebetteten Systemen. Das notwendige Scheduling ist entscheidend, um das Potential von Many-cores effizient zu nutzen.

Der Scheduler eines Betriebssystems erstellt zur Laufzeit Ablaufpläne. Diese Ablaufpläne sind bei Many-cores um ein Vielfaches komplexer als bei Multi-cores. Many-core-Scheduler müssen daher skalierbar bezüglich der Anzahl Prozessorkerne sein. Many-core-Architekturen unterscheiden sich außerdem in mehreren Punkten von klassischen Multi-core-Architekturen. Ein Scheduler muss diese Besonderheiten nutzen, um effiziente Ablaufpläne erstellen zu können.

Die Optimierungsziele eines Scheduler variieren abhängig von den Systemanforderungen. Diese Dissertation stellt eine Reihe an Schedulern für verschiedene Optimierungsziele vor. Die meisten Optimierungsziele sind NP-schwer und können damit im Allgemeinen nicht in polynomieller Laufzeit optimal erreicht werden, außer P=NP. Die meisten bisherigen Arbeiten setzen daher Heuristiken ein, wohingegen wir die Optimierungsziele ohne Heuristiken verfolgen. Für unsere Scheduler geben wir starke theoretische Schranken für die Qualität der Ablaufpläne an. Weiter sind die Scheduler mit dem Ziel entworfen, skalierbar und leicht erweiterbar zu sein.

Wir stellen zwei Scheduler für die Maximierung der Performance von Many-cores vor, einen verteilten Algorithmus und einen zentralisierten Greedy-Algorithmus. Diese Scheduler maximieren die Performance, indem Prozessorkerne zwischen Anwendungen umverteilt werden.

Maximierung der Performance ist nicht bei allen Systemen das Optimierungsziel. In einigen Systemen ist Fairness wichtiger. Wir stellen dafür einen verteilten Scheduler vor, die die Fairness maximiert, indem Prozessorkerne zwischen Anwendungen umverteilt werden. Dieser Scheduler erreicht eine höhere Fairness als der Forschungsstand und erreicht sogar das theoretische Optimum an Fairness, wenn die Performance festgesetzt ist.

Die Performance kann sogar durch einfaches Umordnen von allokierten Prozessorkernen verbessert werden, ohne dabei deren Anzahl zu ändern. Wir stellen einen verteilten Scheduler vor, der mithilfe von Defragmentierung die Performance erhöht. Dieser Scheduler erreicht die optimale Performance, wenn die Anzahl der Threads der Anwendungen eine Zweierpotenz ist.

Dieser Scheduler ist nicht auf Many-cores mit Static Non-Uniform Cache Access (S-NUCA) Caches anwendbar. Für diesen Fall präsentieren wir einen neuen Scheduler, der diese topologiebedingte Heterogenität der Prozessorkerne nutzt, um die Performance zu verbessern. Dieser Scheduler verwendet einen Branch-and-Bound-Algorithmus auf einem Teil des Suchraums um mit kurzer Rechenzeit einen Ablaufplan zu erstellen und übertrifft damit den Forschungsstand.

Die elektrische Leistung eines Many-cores ist strengen Schranken unterworfen, die die Performance einschränken. Wir stellen einen zentralisierten Scheduler für probabilistische Leistungsschranken mit linearer Laufzeitkomplexität vor. Dieser Scheduler ist um Größenordnungen schneller als der Forschungsstand und erreicht dabei fast die selbe Performance. Wir stellen weiter einen Scheduler für probabilistische Leistungsschranken vor, der für Anwendungen mit Anforderungen an Quality of Service (QoS) eingesetzt werden kann.

Schließlich stellen wir unsere quelloffene Toolchain HotSniper vor, die im Laufe dieser Arbeit entwickelt wurde. *HotSniper* kombiniert den Many-core-Simulator *Sniper* mit dem thermischen Modellierprogramm *HotSpot*. Mit *HotSniper* können Scheduler unter echten Anwendungen und einer intervallbasierten thermischen Simulation evaluiert werden, was bisher nicht möglich war.

Contents

1. Introduction

Processors are the fundamental processing entities in any computing paradigm. Processors come in wide-varieties but in this dissertation, we focus mostly on General-Purpose Processors (GPPs) deployed in the embedded system domain. Processors deployed on embedded systems have two main differences vis-a-vis processors deployed on other systems [14]. First, they are powered by batteries which force them to be energy-efficient to maximize the battery life. Energy efficiency is especially important given the fact that advances in battery technology have not kept pace with advances in processor technology. Secondly, they have limited heat dissipation capacities due to the small form-factor of embedded systems which force them to be power-efficient to avoid any thermal damage. Power and energy are related by the following equation; but an energy-efficient processor is not necessarily a power-efficient processor.

$$Energy = Power \cdot Time$$

1.1. Evolution of Processors into Many-Cores

Embedded processors are now ubiquitous. They find use in all kind of devices such as digital cameras, smart phones, smart watches, notebooks, tablets, virtual/augmented reality headsets and even mobile robots. These devices expect high performance from their underlying processors while subjecting them to grueling power and energy constraints. Surprisingly, more efficient embedded processors become with their limited available resources, more performance demanding tasks emerge for them to execute. Therefore, embedded processors are subjected to *Jevons' Paradox* [15] which states that usage of an entity increases instead of decreasing with increase in efficiency of its usage.

Performance of processors has continued to increase mainly because of continued success of *Moore's* Law [16]. Gordon Moore predicted in 1965 that number of transistors in Integrated Circuits (ICs) such as processors would continue to double approximately every two years. Figure 1.1 shows that this law is valid even now. For example, *Qualcomm Snapdragon 835* embedded processor introduced in 2016 has more than 3 billion transistors compared to *Intel 4004* processor introduced in 1971 with only 2300 transistors.

Moore's law was driven by the continued success of semi-conductor industry in reducing the size of the transistors. Figure 1.2 shows how the technology node on which a processor is fabricated has continued to shrink over the years. For example, *Qualcomm Snapdragon 835* processor is fabricated on 10-nm technology node compared to *Intel 4004* processor fabricated on $10\,\mu$m technology node. It is expected that processors fabricated on 7-nm technology node would make an appearance around 2020. Industry is confident that even 5-nm technology node is feasible, though sizing down of technology node is becoming more difficult with introduction of every new technology node. This increasing difficulty is slowing down *Moore's* law. Gordon Moore, himself, had stated that no exponential law can last forever including *Moore's* law.

Increase in the processor's performance was also enabled due to the success of *Dennard* scaling [17]. Robert Dennard in 1974 proposed that power density of an IC such as processor would continue to remain same. Power density of a processor is given by the following relation [18].

$$PowerDensity = Count \cdot Capacitance \cdot SupplyVoltage^2 \cdot Frequency$$

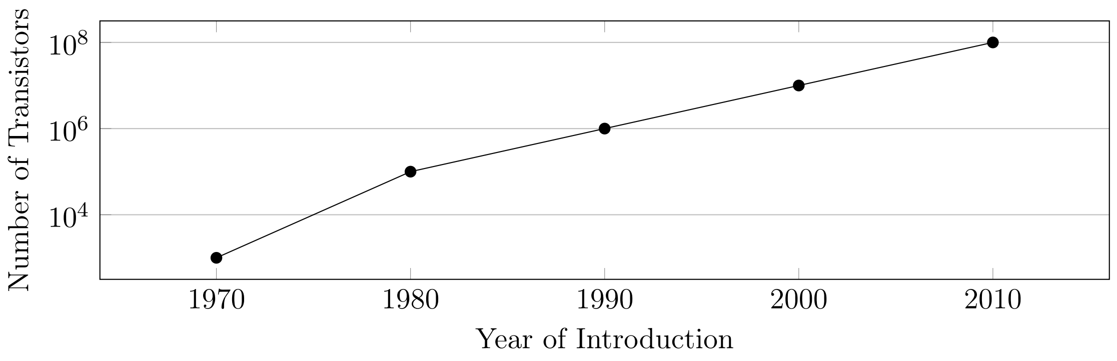


Figure 1.1.: Number of transistors in processors introduced.

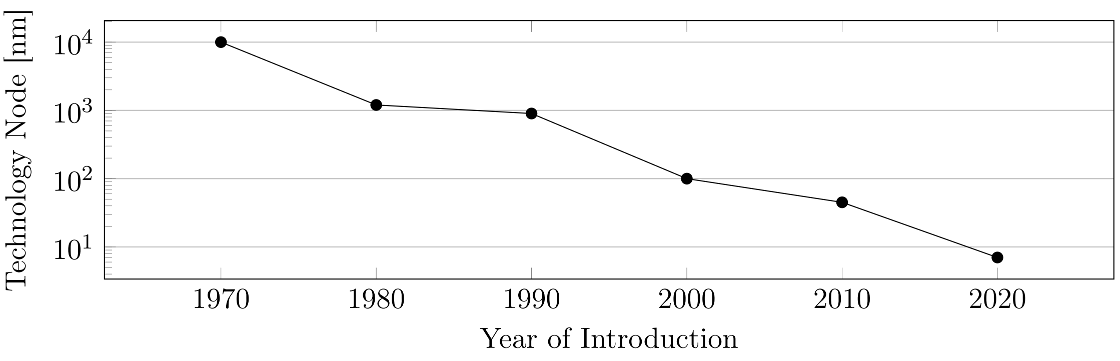


Figure 1.2.: Technology node used to fabricate processors.

Dennard scaling proposed that when a transistor is scaled down by a factor of $1/S$, the number of transistors in a given area (count) increases by a factor of S^2. Capacitance and supply voltage both reduce by a factor of $1/S$. Frequency increases by a factor of S. As a result, all factors cancel out each other keeping the value of power density constant.

Unfortunately, with the reduction in size of a transistor with introduction of every new technology node the leakage power of a processor continued to rise due to quantum effects such as quantum tunneling which *Dennard* scaling did not foresee. Soon the leakage power became as dominant as the dynamic power of a processor and could no longer be ignored. The leakage power of processor is given by the following equation [19].

$$LeakagePower = SupplyVoltage \cdot LeakageCurrent\ (SupplyVoltage,\ Temperature)$$

where leakage current is itself a function of supply voltage and temperature. Processor designers were forced to keep the supply voltage high in order to keep the leakage current manageable. As a result, the supply voltage stopped scaling with the technology node. This failure lead to increase in the power density by a factor of S^2 with every reduction in technology node leading to the eventual breakdown of *Dennard* scaling. Figure 1.3 shows how the power density of a processor has increased over time.

Increase in power density also leads to higher temperature which results in higher leakage current which in turn again leads to higher temperature. This feedback loop between temperature and leakage current (or leakage power) is known as thermal-runaway effect and would eventually damage the processor if not kept under check.

The processor designers mitigated the power density problem to some extent by not increasing the processor frequency. This strategy still led to an increase of power density by a factor

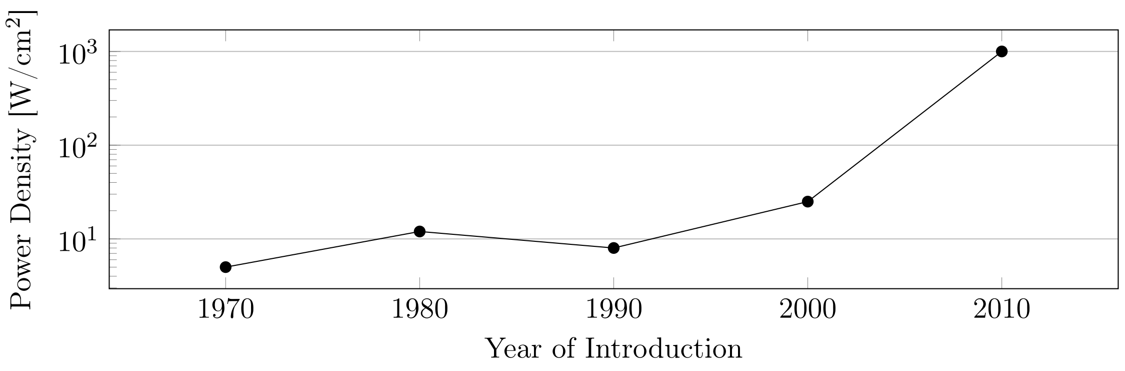


Figure 1.3.: Power density of processors introduced.

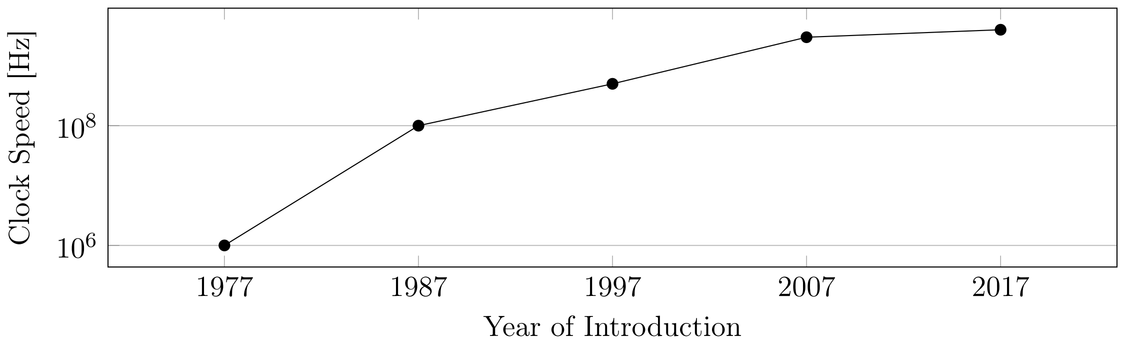


Figure 1.4.: Clock frequency of processors introduced.

S with scaling down of transistor by a factor of S. This eventually led to the problem of Dark Silicon [19] wherein all the transistors within a processor cannot be simultaneously turned on. Figure 1.4 shows how processor frequency have now stagnated around 4 GHz. The frequency at which a core operates and the corresponding minimum supply voltage required by it are given by the following equation [20].

$$Frequency = k \cdot \frac{(SupplyVoltage - ThresholdVoltage)^2}{SupplyVoltage}$$

Providing a core supply voltage higher than the minimum value given by the above equation for its frequency to be stable is both power- and energy-inefficient. Accordingly, a higher frequency needs a higher voltage to sustain it creating a cubic relationship between frequency and power density. Designers exploited this relationship to manage power density by decreasing instead of increasing processor frequency with the shrinking of technology node. For example, *Qualcomm Snapdragon 835* processor runs at a peak frequency of only 2.45 GHz.

Designers also introduced Dynamic Voltage and Frequency Scaling (DVFS) technology into the processors. DVFS allows cores of a processor to run at different discrete frequencies below their peak frequencies to reduce their power consumptions but this comes at the cost of their performance. DVFS, therefore, provides a knob to trade-off performance with power consumption. We have used DVFS widely throughout this dissertation. Nevertheless, the limitations on frequencies enforced limitations on the single-threaded performance of uni-core processors.

Furthermore, processor designers due to the limited capabilities of their design tools were not able to put all the transistors being made available by *Moore's* law to efficient use by building ever more sophisticated uni-cores. Notwithstanding this productivity gap, designers with great efforts used the abundant transistors to create longer and more complicated pipelines

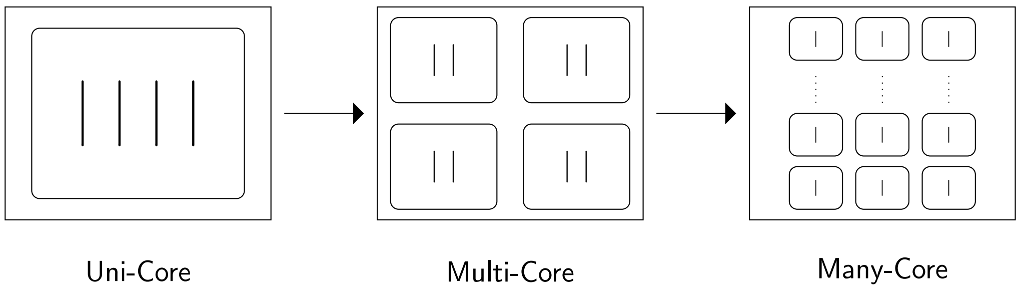


Figure 1.5.: An abstract diagram depicting transition from uni-cores to many-cores via multi-cores.

for uni-cores. Unfortunately, limited Instruction Level Parallelism (ILP) in single-threaded tasks inhibited any substantial performance gain from longer pipelines after a certain point. Uni-core processors were also limited in performance by lower clock frequencies of accompanying memories forcing them to stall in their execution.

All these limitations commonly refereed as the power-, design-, ILP- and memory-wall forced the industry to move from uni-core processors to multi-core processors. Major turning point in processor design was when *Intel* around 2006 abandoned further development in its *Pentium* line of uni-cores and instead introduced *Core* line of multi-cores. Multi-cores were composed of less sophisticated cores with smaller pipelines compared to uni-cores. Cores of multi-cores also ran at lower frequencies compared to equivalent uni-cores. In a nutshell, designers sacrificed single-threaded performance in favor of multi-threaded performance.

This move not only forced task developers to move from a single-threaded to multi-threaded programming model but also forced system developers to develop new multi-threaded schedulers to effectively manage the multi-cores. Schedulers are the OS sub-routines that perform resource management for processors. A scheduler works towards an objective defined by the overlying system where a processor is deployed under the constraints stipulated by the system's design and environment. This dissertation is dedicated to the study of scheduler designs.

The trend of adding more and more cores to a processor has continued over the last decade and now we are entering the realm of many-core processors from multi-core processors [21]. Compared to multi-cores that have a dozen or so cores, many-cores house tens or even hundreds of cores on a single-die. Beside the core count, many-cores also have a fundamentally different micro-architecture in comparison to multi-cores.

For example, it is common for multi-cores to have a physically-unified logically-shared Last-Level Cache (LLC) shared by all cores using a memory bus. Unfortunately, this bus-based architecture does not scale up beyond few cores. Therefore, many-cores come with a physically-distributed LLC which accessible to all cores via a Network-on-Chip (NoC). In many-cores, the number of threads dominate the number of cores and hence a core in multi-core executes multiple threads in parallel using context switching. On the other hand, in many-cores number of cores dominate the number of threads. Therefore, there is no need to waste clock cycles in context switching and many-cores operate with a one-thread-per-core model. Figure 1.5 shows the transition from uni-cores to multi-cores to many-cores using an abstract block diagram.

1.2. Multi-Core Vs. Many-Core Scheduling

Many-cores are inherently imbued with an immense parallel processing potential. Many-cores can be used to run embarrassingly parallel tasks capable of spawning hundreds of threads which can exploit their potential. Though, these tasks, one can also run efficiently using multi-thread [22] or multi-grid processors [23]. True use of a many-core emerges when one wants to

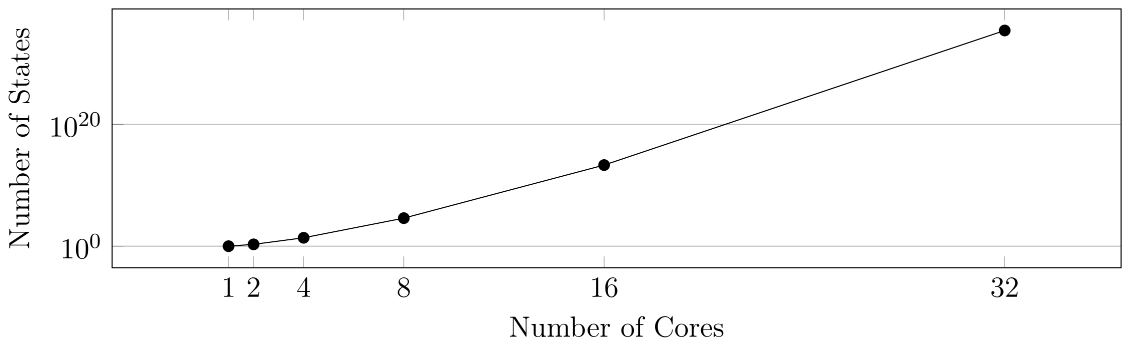


Figure 1.6.: Search-space for varisized many-cores.

run tens of tasks in parallel with each task capable of scaling to a dozen or so cores but not to all cores of the many-core unlike an embarrassingly parallel task. Many-cores are especially useful when these limited scalability tasks have to be executed on embedded platforms [24].

Multi-core schedulers can even operate using a single-threaded brute-force algorithm because the size of scheduling problem is small on multi-cores when compared to the size of scheduling problem on many-cores. Figure 1.6 shows how the search-space for scheduling on many-cores increases factorially with the increase in the size of many-core. The size of the search-space is already unmanageably large with 32 cores. No algorithm can brute-force its way to the optimal solution. Therefore, better alternatives need to be developed.

Many-core schedulers have no option but to find ways to scale up with increase in the number of cores. For example, one way for a many-core scheduler to scale up is to distribute its processing across all cores in the many-core by using a multi-threaded distributed scheduling algorithm to develop a schedule for the many-core. Other possible ways to make schedulers scale up are to prune the search-space they need to explore or even make them probabilistic. This dissertation primarily focuses on the problem of how to make many-core schedulers scale up with increase in the number of cores in many-cores but without losing schedule quality.

Researchers have often resorted to heuristics to meet the challenges of many-core scheduling. Unfortunately, heuristic schedulers make no promise on the schedule quality and suffer from corner cases where schedule quality is especially poor. We on the other hand in this dissertation, create schedulers which make no compromises on the schedule quality for the sake of scalability. Therefore, we provide a schedule quality similar to multi-core schedulers on many-cores.

Furthermore, since cores of a multi-core can execute multiple threads in parallel it makes the search-space for a multi-core scheduler continuous. The search-space for multi-core scheduling thereby can efficiently be traversed using methods like linear programming or convex optimization. On the other hand, many-core operates with one-thread-per-core model making the search-space for many-core schedulers discrete. Because of this, most many-core scheduling optimization problems have NP-hard computationally complexity and are even difficult to solve optimally at run-time with well-established efficient tools like ILP and convex solvers.

Therefore, there is a need for studying many-core scheduling problems even with the most basic goals like performance maximization. Beside performance maximization, we introduce several new schedulers in this dissertation which target several diverse goals with strong schedule quality guarantees. We also develop many-core schedulers that exploit the new micro-architectural features in many-cores such as physically distributed LLC and NoC to achieve their objectives in line with the hardware-software codesign methodology.

1.3. Many-Core Vs. Multi-Thread Computing Paradigm

Many-core computing paradigm is not the only computing paradigm offering massive parallel processing potential to its end users. Multi-thread computing paradigm enabled by processors like Graphic Processing Units (GPUs) is also a well-established computing paradigm design to support massive parallel processing. Multi-thread processors have now also forayed into embedded systems [25]. Nevertheless, multi-thread computing paradigm differ from many-core computing paradigm in some crucial ways making them each occupy a different niche.

Multi-thread processors were initially designed for running games for which they work with a GPP Central Processing Unit (CPU) in tandem [26, 27]. Multi-thread processors were soon also deployed to execute scientific computing workloads which were very similar to graphic workload in games. Multi-thread processors by design can execute lot of threads in parallel similar to many-cores on hundreds of similar cores. But unlike cores of many-cores, cores of multi-thread processors have very small caches. They hide the latency of off-chip memory accesses by the concurrently executing threads by context switching threads waiting for memory with threads ready for execution [28]. This strategy only works when threads do not communicate with each other and are mostly independent.

Therefore, multi-thread processors cannot replace many-cores in a scenario where there are large number of tasks executing in parallel which are independent of each other but threads from a given task are constantly communicating and dependent on each other. Accordingly, schedulers designed for many-cores need to have a completely different design than schedulers design for multi-thread processors [29]. Furthermore, many-thread processors cannot run an OS and hence are dependent upon the accompanying CPU to develop a schedule for them.

1.4. Many-Core Vs. Multi-Grid Computing Paradigm

Multi-grid computing paradigm enabled by supercomputers also allows for massive parallel processing. Multi-grid processors are composed of multiple processors distributed over vast physical distances (sometimes even continents) acting as one processing entity. Trivially, they cannot completely replace many-cores in embedded system domain though they can be used to offload some of the computing from embedded systems using recent advances in cloud/edge computing [30]. Multi-grid processors given their massive hardware and power consumption are also many times more expensive to operate than many-cores.

Another major difference between multi-grid processors and many-cores is the disparity between cost of computation versus cost of communication. In multi-grid processors, the cores are computationally very powerfully but communication between the cores is very slow specially when they are physically separated. On the other hand, in many-cores the cores are comparatively computationally weak compared to cores in multi-grid processors but they can communicate very fast with each other using on-chip NoC and caches. Therefore, grid processors cannot also replace many-cores in a scenario where there are large number of tasks executing in parallel which are independent of each other but threads from a given task are constantly communicating and dependent on each other. Accordingly, schedulers designed for many-cores need to have a different design than schedulers design for multi-grid processors [31].

1.5. Dissertation Contributions

This dissertation made several research contributions on the subject of many-core scheduling. We present several different many-core schedulers with different optimization objectives which make several advancements over the state-of-the-art.

We introduce two schedulers for performance maximization on many-cores. State-of-the-art used Dynamic Programming (DP) to solve the performance maximization problem optimally. We introduce a greedy and distributed scheduler which exploit convex substructures present in the performance maximization problem on many-cores to also solve the problem optimally. Therefore, both proposed schedulers produce results equivalent to the state-of-the-art. The main contribution of proposed schedulers is their ability to optimally maximize the performance of many-cores but with several times less scheduling overheads compared to state-of-the-art.

We introduce a distributed scheduler for fairness maximization on many-cores. The problem of fairness maximization on many-cores is NP-hard but we exploit convex substructures present in fairness maximization problem on many-cores to solve the problem optimally for fixed performance. Our proposed scheduler also outperforms state-of-the-art heuristic schedulers.

We introduce a distributed scheduler for defragmenting task allocations on many-core. Task defragmentation on many-cores is an NP-hard problem but we solve the problem optimally in the special case when all tasks are constrained to produce only power-of-two number of threads. Our scheduler outperforms state-of-the-art heuristic schedulers under the constraint.

We introduce a scheduler for task allocation on many-cores with S-NUCA caches. Presence of S-NUCA caches introduce a design-time heterogeneity in otherwise homogeneous cores of a many-core. Our scheduler uses the knowledge of this design to first prune the search-space and then perform optimal task allocations on the many-core. The scheduler outperforms state-of-the-art heuristic scheduler oblivious to the S-NUCA design.

We introduce a probabilistic scheduler for power budgeting on many-cores. Our scheduler drastically reduced the scheduling overheads in comparison to non-probabilistic schedulers. The scheduler provides strong guarantees on the violation of power budget while providing performance equivalent to state-of-the-art heuristic scheduler. We also extend the proposed scheduler to work with QoS tasks.

Finally, we introduce a toolchain for evaluating many-core schedulers using fast yet precise interval thermal simulations. Toolchain tightly couples together a state-of-the-art many-core interval simulator and thermal modeling tool. The toolchain also adds support for interval thermal simulations of many-cores deployed in open systems which were previously not possible.

1.6. Dissertation Outline

The remainder of this dissertation is outlined as followed.

- Chapter 2 presents the background required for better comprehension of this dissertation.
- Chapter 3 presents the common notations used to describe the proposed schedulers.
- Chapter 4 presents two schedulers for many-core performance maximization.
- Chapter 5 presents a distributed scheduler for many-core fairness maximization.
- Chapter 6 presents a distributed scheduler for many-core task defragmentation.
- Chapter 7 presents a scheduler for many-core with S-NUCA caches.
- Chapter 8 presents a scheduler for many-core power budgeting.
- Chapter 9 presents a scheduler for many-core power budgeting with QoS tasks.
- Chapter 10 concludes this dissertation and present some ideas for possible future works.
- Appendix A presents a toolchain designed to evaluate many-core schedulers using fast and precise interval thermal simulations in real-world representative open systems.

2. Background

This chapter presents the background information required for better comprehension of this dissertation. We first present the type of systems wherein a many-core can be deployed. Thereafter we introduce the different types of tasks which can be executed on many-cores and also how these tasks interact with the many-core scheduler. We then present the timing model used by many-core schedulers to manage task execution. We then introduce how many-cores can be classified into different categories based on their core clusterings and compositions. We then end this chapter with details of some famous many-core platforms and OSs.

2.1. Type of Many-Core Systems

Many-cores can be deployed in three kinds of systems [32] namely fixed, closed or open systems. The systems differ from each other in terms of how a workload is executed on the many-cores. Figure 2.1 shows how the three systems differ from each other using an abstract block diagram.

A fixed many-core starts execution with a predefined set of tasks (workload) and shuts down when entire workload has been executed. Fixed many-cores find use in embedded systems like washing machine and microwaves where workload is more or less same on every execution. Makespan which is the time from start to end of entire workload execution is the preferred metric to measure the performance of fixed many-cores.

A closed many-core starts with a predefined workload but tasks in the workload restart execution as soon as they are finished. Fixed many-cores find use in embedded systems like mobile robots which start analyzing a frame captured using their camera (eyes) with algorithms such as sift feature extraction followed by edge detection as soon as they complete analyzing the previous frame with same algorithms. Throughput which is measured as the number of tasks finished per unit time is the preferred metric to measure the performance of closed many-cores.

An open many-core does not have a previously known workload, and tasks arrive in the open many-core at a non-initial time and leave once they complete execution. Fixed many-cores find use in embedded systems like smart phones where workload is generated on the fly based on user interactions. Average response time which is measured as the average time between tasks arrivals and departures is the preferred metric to measure the performance of open many-cores.

In this dissertation, we evaluate our proposed schedulers on mainly closed and open many-cores. All schedulers should work with bare minimal modifications on all kind of systems wherein many-cores can be deployed even if it is not explicitly specified.

2.2. Type of Many-Core Tasks

Tasks executed on many-cores can be of three different types namely rigid, moldable or malleable. Figure 2.2 depicts with an abstract diagram how different task types interact with a many-core scheduler. All types of tasks start execution with a set of initial cores allocated to them by the scheduler. Initial cores allocated to a rigid task cannot be modified. Cores allocated to a moldable task can be modified using thread migrations, but the total number of cores allocated to it should always remain the same. Cores allocated to a malleable task can be modified even in terms of number of allocated cores at any time during its execution.

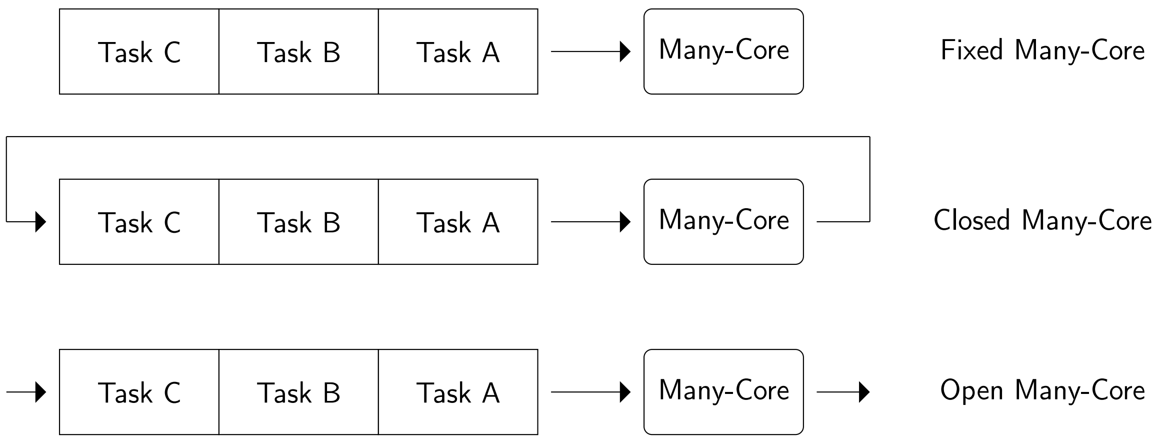


Figure 2.1.: An abstract diagram depicting different kind of many-core systems.

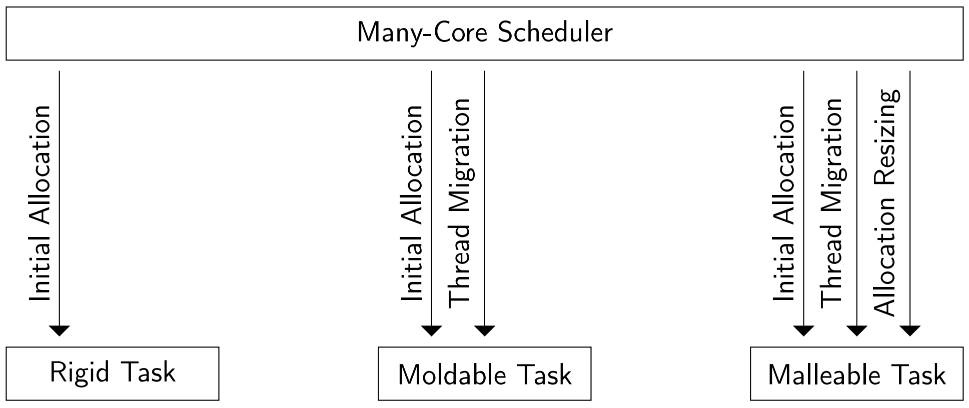


Figure 2.2.: An abstract diagram depicting interaction between many-core scheduler and different types of tasks a many-core can execute.

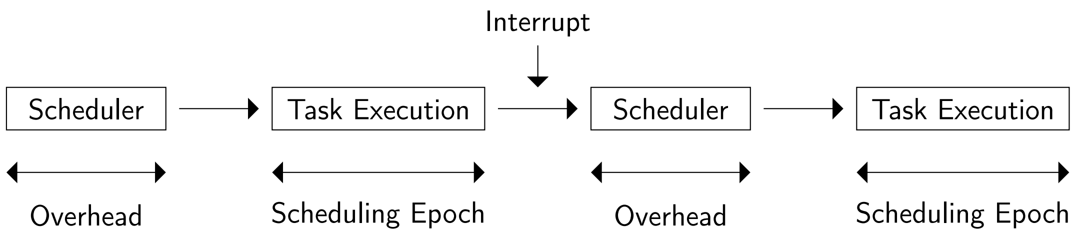


Figure 2.3.: An abstract diagram depicting interrupt-based timing model for a many-core scheduler.

A multi-threaded task in a many-core can be allocated multiple cores, but a core is always allocated exclusively to only one task at any given time. The tasks follow one thread per core execution model. We develop schedulers for all the three types of tasks in this dissertation.

2.3. Many-Core Scheduler Timing Model

Figure 2.3 depicts the interplay facilitated by a many-core OS between a many-core scheduler and task execution it manages. OS invokes the scheduler regularly at a granularity of scheduling epoch using a time-triggered interrupt to perform task scheduling. We set the value of scheduling epoch in this dissertation at 10 ms; same as default *Linux* schedulers [33]. The time taken by the scheduler to come up with an appropriate schedule is the scheduling overhead

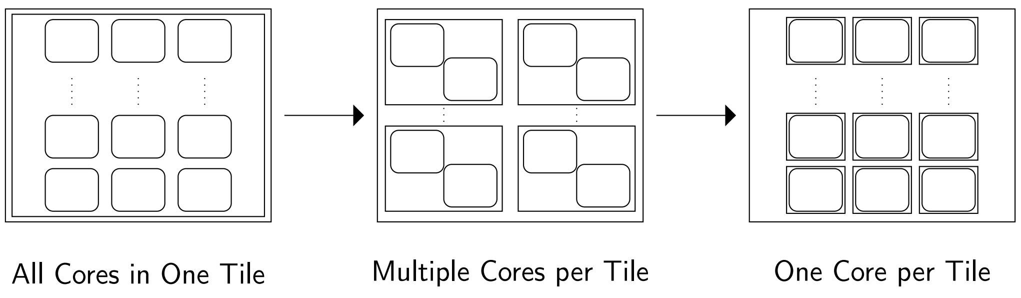


Figure 2.4.: An abstract diagram depicting different ways cores of a many-core can be divide into tiles.

induced by it. The fundamental challenge in many-core scheduling research is to minimize the scheduling overhead while simultaneously maximizing the schedule quality.

Another scheduler timing model is where the scheduler executes in parallel to the executing tasks and manage them without using interrupts. This parallel scheduler timing model is not explored in this dissertation and we focus on only interrupt-based scheduler timing model.

2.4. Many-Core Tiling Models

Cores in a many-core can be clustered into tiles at design-time. Cores within a tile share a cache which is accessible to them using an intra-tile memory bus and hence accessing this shared cache has the same latency for all cores. Cores within a tile also always operate at the same frequency and hence granularity of tiling determines the granularity at which DVFS can be performed within the many-core. All tiles are connected together using a NoC with each tile containing one NoC router or switch.

Figure 2.4 shows with an abstract block diagram three generic ways tiling can be done in a many-core. The first way is to place all cores of the many-core into one tile. The second way is to place cores into varisized tiles with each tile containing different (or same) number of cores. The third way is to place only one core in every tile.

Placing all cores within one monolithic tiles is the least complicated design but this design is also least flexible in terms of optimization potential it offers an accompanying many-core scheduler to exploit. This design also contains no NoC and hence does not scale up very well. Scalability of the design can be improved by having varisized tiles. The design having only one core per tile is most flexible in terms of optimization potential it offers to the scheduler though it also involves substantially more design effort than the other designs. The run-time advantages offered by this design easily justifies its design-time disadvantages. This dissertation therefore focuses mostly on many-cores with one core per tile design.

2.5. Many-Core Composition Models

Many-cores can be divided into three types depending upon the micro-architectural composition of its cores [34] as shown in Figure 2.5. All the cores in a homogeneous many-core are micro-architecturally the same. Still, even cores in a homogeneous many-core may diverge in terms of their performance potential due to many factors such as fabrication-induced process variation or topology-induced non-uniform LLC latency.

A heterogeneous many-core is composed of cores that inherently have a different micro-architecture. Heterogeneity in cores can be function- or performance-based. Functionally heterogeneous cores of a many-core have a completely different Instruction Set Architecture (ISA)

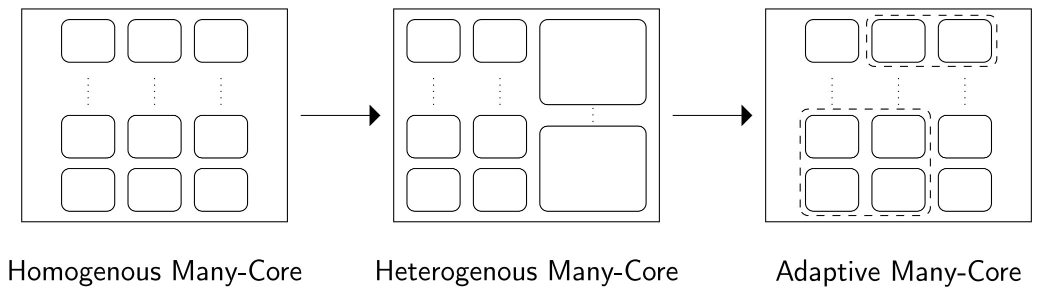


Figure 2.5.: An abstract diagram depicting types of many-cores classified based on core composition.

from each other and hence task compiled for one type of core cannot be executed on another type of core. For example, a heterogeneous many-core can couple together embedded CPU cores with embedded GPU cores on the same die. On the other hand, performance heterogeneous cores of a many-core have the same ISA and hence a task compiled for one type of core can be seamlessly executed on another type of core. For example, a heterogeneous many-core can couple together high-performance high-power out-of-order cores with low-performance low-power in-order cores on the same die [35]. A heterogeneous many-core can be simultaneously functional and performance heterogeneous.

An adaptive many-core is a special type of many-core which contains large number of simple base cores which can coalesce together at run-time to form bigger complex cores [36]. While the design of homogeneous and heterogeneous many-cores are fixed after fabrication, adaptive many-cores are capable of turning into either one even after fabrication. This imbues adaptive many-cores with the ability to adapt to any possible kind of workload they can encounter. We focus mostly on adaptive and homogeneous many-cores in this dissertation.

2.6. Many-Core Platforms

In this section, we introduce some state-of-the-art many-core platforms. We begin by introducing *InvasIC* many-core platform in whose development the author of this dissertation actively participated. We then introduce some of the other available many-core platforms.

2.6.1. InvasIC (Funding Project)

Invasive computing (*InvasIC*) [21] is an on-going collaborative project mainly amongst three German university *Karlsruhe Institute of Technology*, *Technical University of Munich* and *Friedrich-Alexander-Universität Erlangen-Nürnberg*. The goal of the project is to solve the scientific challenges involved in designing many-cores processor and develop the associated technologies. The many-core paradigm brings in such a radical shift in processor design that every step of the the design needs to carefully reworked to produce an efficient many-core. The Ph.D. of author of this dissertation was funded under the Invasive computing project.

Figure 2.6 shows the conceptual block diagram for an *InvasIC* many-core. In an *InvasIC* many-core, the cores are divided into tiles. The GPP core used in *InvasIC* many-core is a SPARC-based *Leon3* core [37] from *Gaisler* with Reduced Instruction Set Computer (RISC) ISA. Some of the *Leon3* cores are enhanced with a tightly-coupled reconfigurable fabric turning them into an *i-Core* [38] that can run tasks faster using hardware-acceleration. Enhanced performance from *i-Core* comes at a cost of increased area and hence it is not recommended to turn all *Leon3* cores into *i-Core*s. Each compute tile – a tile with *Leon* and/or *i-Core* – also contains a local memory accessible only to the cores in the tile using a memory bus. A

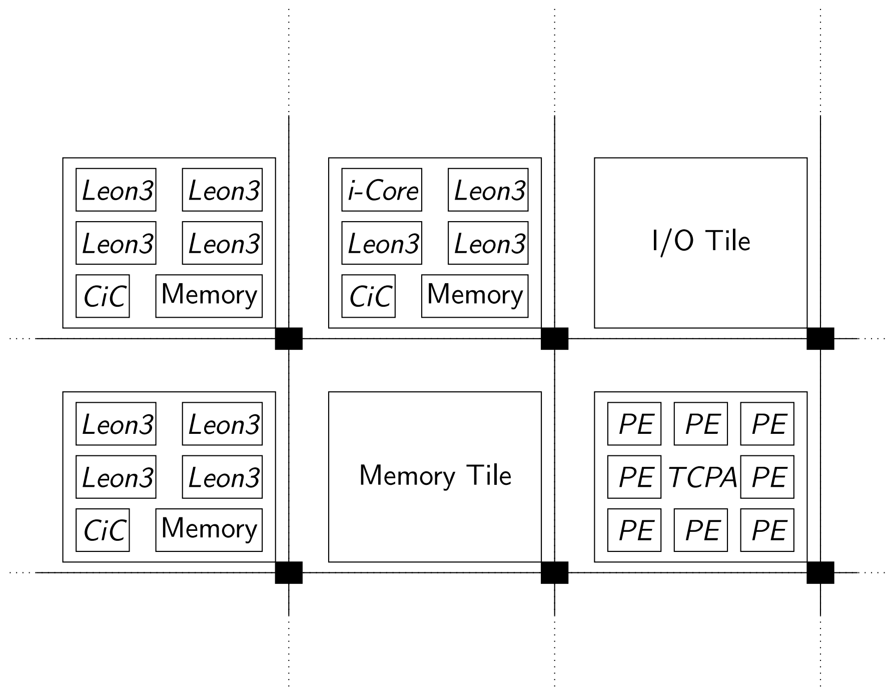


Figure 2.6.: A conceptual block diagram of an *InvasIC* many-core.

compute tile also comes with a hardware-implemented *i-let* controller called *CiC*. In Invasive Computing, *i-lets* are similar to threads but unlike the threads *i-lets* cannot be preempted and always run to completion once they start execution.

InvasIC many-core is designed to execute malleable tasks [39] which spawn hundreds of *i-lets*. *InvasIC* tasks produce *i-lets* that do not communicate with each other and hence can scale up to a large number of core. *InvasIC* tasks also allow their underlying allocations to be modified anytime during their execution. *InvasIC* task acquire an initial set of cores to begin execution via an *invade* construct. *InvasIC* task can release cores at run-time via a *retreat* construct whereas it can acquire more cores at run-time via a *reinvade* construct. Cores held by an *InvasIC* task is called its *claim* and the task can start using its claim via an *infect* construct. *InvasIC* tasks can exchange cores among themselves with the help of an agent system [40]. These *InvasIC* constructs also make *InvasIC* tasks substantially fault-tolerant [41].

InvasIC many-core deploys a distributed library OS called *OctoPos* [42]. Each tile in *InvasIC* many-core runs its own instance of *OctoPos*. The different instances of *OctoPos* running in parallel communicate with each other using Remote Procedure Calls (RPCs). The agent system together with *OctoPos* is called *iRTSS*. *OctoPos* supports execution of tasks written in both *C++* and *X10* [43]. It can run natively on both *SPARC* and *x86* architectures. It also supports a so called "Guest Layer" that allows rapid functional prototyping via emulation of the entire Invasive Computing stack on *Linux*.

Beside the numerous compute tiles with *Leon3* and *i-Core* cores, *InvasIC* many-core contains few other types of specialized tiles. A *TCPA* tile [44] is similar to an integrated GPU and can be used for hardware-acceleration of *InvasIC* tasks using an array of Processing Elements (PEs). A memory tile is a tile containing the global memory which can be accessed from any tile. Finally, an *I/O* tile is a tile containing components like *Ethernet* through which *InvasIC* many-core can communicate with external entities. Tiles are connected using a special reconfigurable NoC called *iNoC* [45]. *InvasIC* tasks can exploit several unique features provided by *iNoC* such as end-to-end connections using service level guarantees and Direct Memory Access (DMA) [46].

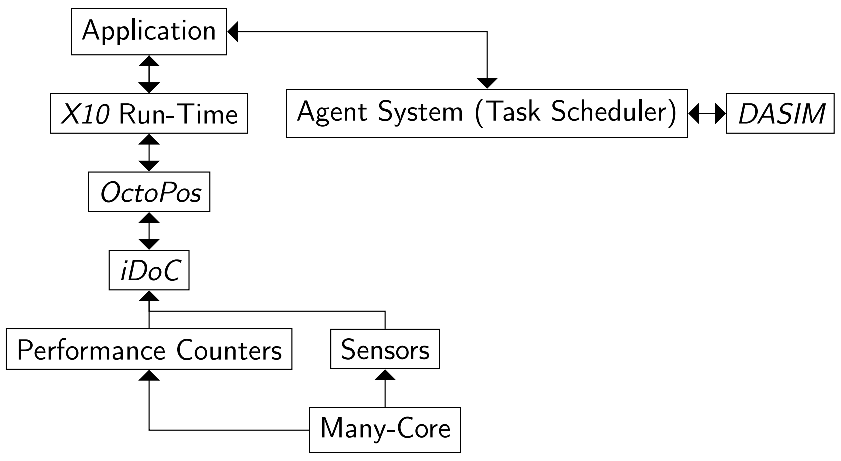


Figure 2.7.: Scheduling control flow on an *InvasIC* many-core.

Standard directory-based cache-coherency protocols do not scale up well on many-cores [47]. Therefore, *InvasIC* many-core supports a novel highly scalable region-based on-demand cache-coherency [48] where parts of the many-core can be made cache-coherent at run-time and the remaining many-core operates using message-passing. *InvasIC* memory model fits very well with the memory model of *X10* programming [49]. Tasks written in *X10* work on the principle of a *place* which is internally cache-coherent wherein *places* themselves communicate using message-passing. A *X10* task written using an *ActorX10* library [50] can run directly on an *InvasIC* many-core or an equivalent *InvasIC* many-core simulator [51].

InvasIC many-core is of great use in High-Performance Computing (HPC) on embedded systems [52]. HPC tasks, like shallow water tsunami simulation written in *ActorX10* which spawn hundreds of *i-lets*, have already been shown to be feasible on *InvasIC* many-core [53]. *InvasIC* many-core is also of great use for executing tens of tasks in parallel each of which can only span to a limited number of cores such as tasks produce by a robot while performing motion planning [54] or tasks executed in a 5G base station [55]. This scenario inspired most of the schedulers presented in this dissertation.

Figure 2.7 shows the control flow of scheduling performed on an *InvasIC* many-core. The distribution of *i-lets* produced by a task amongst cores in its claim is by default performed by *CiC* hardware module which attempts to balance the processing load equally amongst all cores. Though the scheduling logic in the agent system can disable *CiC* and take control of the *i-let* distribution in software for a more desirable distribution than the basic distribution provided by *CiC*. Agent system in an *InvasIC* many-core then act as a distributed task scheduler.

When an application starts it asks the task scheduler (agent system) for some cores to start execution. Task scheduler then assigns some free cores to the task as its initial claim. If no free cores are available, then the task scheduler takes away some cores from already executing tasks based on an overall cost-benefit analysis [56] to create an initial claim. The cost-benefit analysis is based on data collected using performance counter and sensors using a data aggregator implemented in hardware called *iDoC* [5] accessible through *OctoPoS*. Sensors can also be replaced with sensor emulation tools such as *MatEx* [57] while using simulators.

The initial claim assigned to task can expand or shrink in size through cost-benefit negotiations between executing tasks at run-time facilitated by the task scheduler [40]. Dark silicon constraints [58] force cores allocated to a task to run at less than their peak frequency to prevent thermal violations. A thermally safe power budget of all the cores in a tile are determined based on a core-level power budget technique called *TSP* [59] by another power-budgeting scheduler

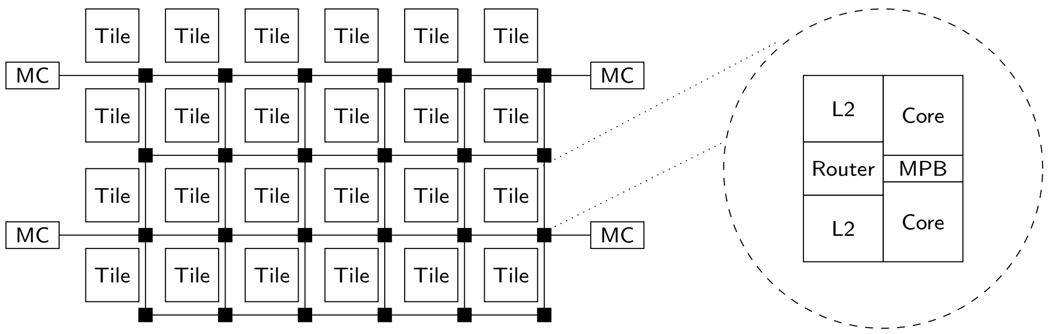


Figure 2.8.: A conceptual block diagram of an *Single-Chip Cloud Computer (*SCC) many-core.

called *DASIM* [5]. The task scheduler then tries to maximize the frequencies at which cores can operate under *DASIM* assigned power budget with the help of *i-let* patterning [60].

2.6.2. SCC

Intel introduced *SCC* [61] processor around 2009 on 45-nm technology node using high K Complementary Metal-Oxide-Semiconductor (CMOS) transistors under its *TeraScale* research program. Total die area for *SCC* is around 567 mm^2 [62]. *SCC* pioneered the idea of a many-cores with an efficient hardware/software co-design. Figure 2.8 shows a conceptual block diagram of the *SCC* many-core. *SCC* integrates 48 32-bit cores with *x86* ISA on a single die. Each core has a private 16 KB L1 instruction and data cache, and a unified 256 KB private L2 cache. Cores in *SCC* can run at multiple discrete frequency levels using DVFS.

Every two cores in *SCC* are grouped together to form a tile with a die area of 18 mm^2. Cores within a tile share a 16 KB Message-Passing Buffer (MPB) and NoC router. Tiles are arranged in a 6x4 grid pattern connected over a 2D-mesh NoC. Tiles communicate with each other using message-passing. Therefore, *SCC* is geared towards high-performance execution of Message-Passing Interface (MPI) tasks. NoC links have a bandwidth of 64 GB/s per link with a total bi-section bandwidth of 2 TB/s. Tiles can access the main memory using four Memory Controllers (MCs) on the periphery. *Intel* made the *SCC* platform available to different research institution including ours to foster the research on subject of many-cores.

SCC by default ships with a distributed *Linux* OS with an instance of *Linux* kernel running on each tile. It also supports a bare-metal run-time environment for minimal overhead execution of MPI tasks. Even *Barrelfish* OS [63] described in Section 2.7.2 has been ported onto to *SCC* [64]. *C*, *C++* and *Fortan* compilers are available for *SCC*. Authors of [65] developed an agent-system based scheduler for task scheduling on *SCC*.

Though in this dissertation we focus on developing schedulers for only cache-coherent many-cores, many of the introduced schedulers can also be easily modified to work with message-passing based many-cores such as *SCC*.

2.6.3. TILE-Gx100

TILE-Gx100 [66] is a *TILE-Gx* series many-core developed by *Tilera* (now *EZchip Semiconductor*) around 2011 on 40-nm technology node. It contains 100 64-bit RISC cores arrange in 10x10 grid on a single-die. Cores are connected together using a mesh NoC. Figure 2.9 shows a conceptual block diagram for *TILE-Gx100* many-core.

The cores in *TILE-Gx100* can run at frequency between 1 to 1.5 GHz using DVFS. Each core contains 32 KB L1 data and instruction cache. Each core also contains a 256 KB L2 cache. Cores are 3-issue Very Long Instruction Word (VLIW) in-order cores with a 5-stage pipeline.

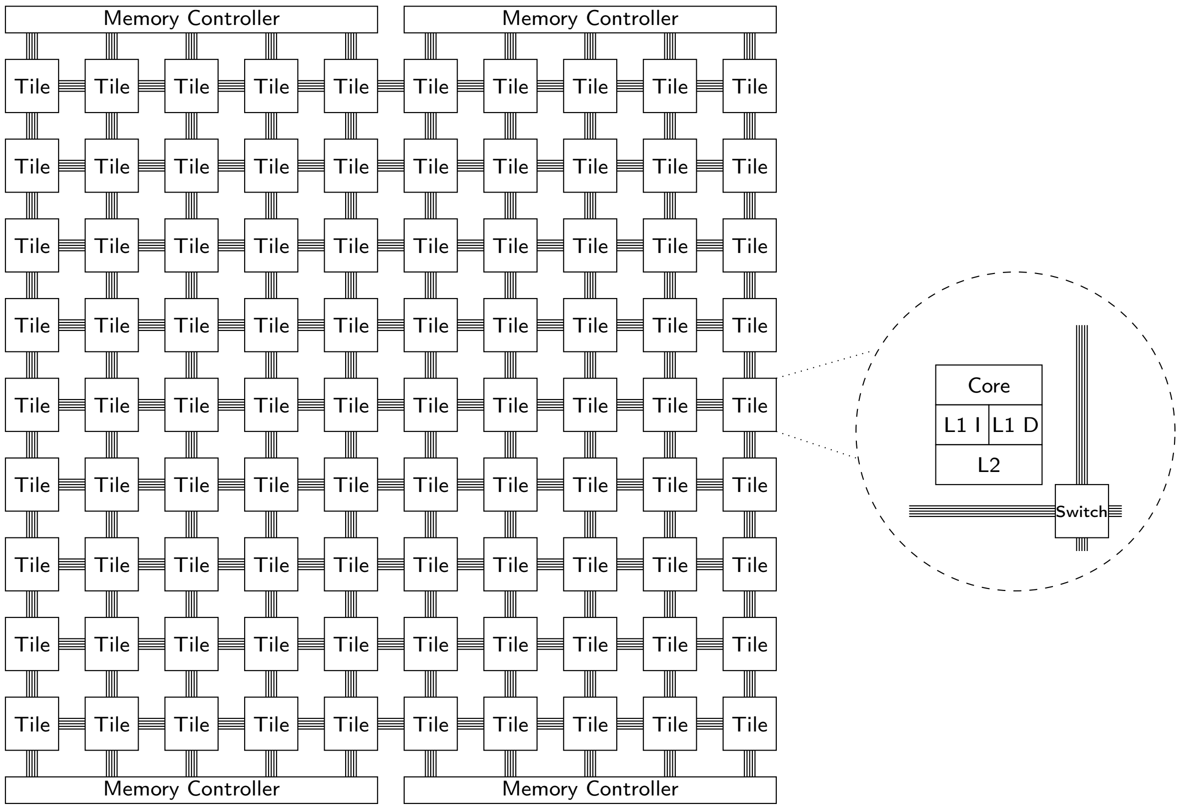


Figure 2.9.: A conceptual block diagram of a *Tilera TILE-Gx100* many-core.

Cores are connected together using 5 different NoCs with each NoC serving different kind of data. All cores have their own NoC router. The cores can be both clock- and power-gated to save power. *TILE-Gx100* also contains a large L3 cache. It also ships with several types of hardware accelerators for tasks such as random number generation and cryptography.

TILE-Gx100 is cache-coherent and hence can run Symmetric Multi-Processing (SMP) applications. It can run one of three types of OS namely SMP *Linux*, zero overhead *Linux* or *BareMetal*. Different parts of *TILE-Gx100* can run different OS in parallel using a *hypervisor* layer. It supports standard *C*, *C++* and *Java* library out-of-the-box.

2.6.4. Epiphany-V

Epiphany-V [67] is a 1024-core many-core developed by *Adapteva* in 2017 on 16-nm technology node. It has a die area of 117.44 mm^2 which packs together 4.56 billion transistors. Figure 2.10 shows a conceptual block diagram for *Epiphany-V*. All cores are 64-bit RISC cores with a low-power design but together they can provide impressive energy-efficient performance of 75 GFLOPS/Watt. In total, 1024 cores can provide a throughput of 4 TFLOPS. Cores are connected together with three mesh NoC. *Epiphany-V* was preceded by 28-nm 64-core *Epiphany-IV* and 65-nm 16-core *Epiphany-III* before that.

Epiphany-V has a cache-less memory design similar to a scratchpad memory design. The memory is physically distributed along the cores but the entire memory is accessible to all cores making it a shared-memory many-core but without cache-coherency. In total, there is 64 MB of on-chip memory. It supports 64-bit memory addressing and 64-bit floating point operations. It also ships with a custom ISA for domains like machine learning and cryptography. Multiple *Epiphany-V* many-cores can be connected together to create a much bigger virtual many-core. Creators of *Epiphany-V* claim that the virtual many-core can scale up to a billion cores.

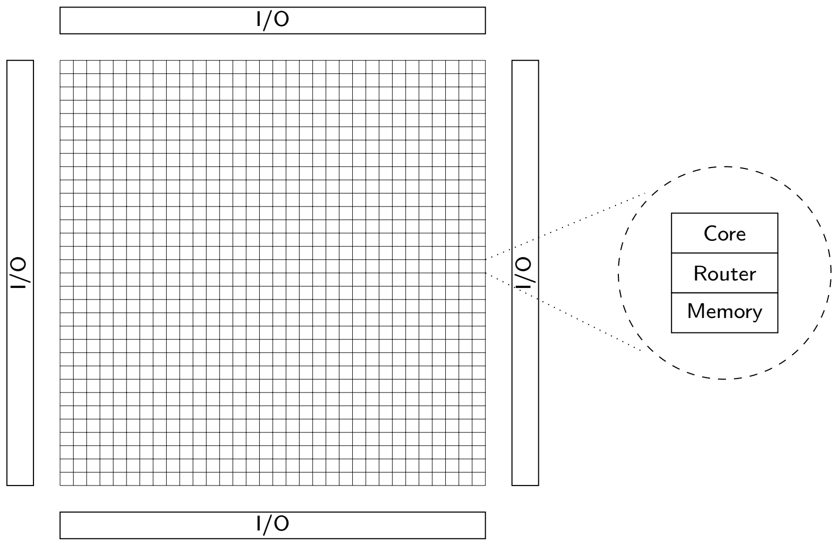


Figure 2.10.: A conceptual block diagram of an *Epiphany-V* many-core.

Epiphany-V supports standard C/C++. There are also several community-supported parallel programming frameworks that can be used to write programs for *Epiphany-V*. *Epiphany-V* also comes with a functional simulator to speedup up the software development.

2.7. Many-Core Operating Systems

In this section, we introduce some state-of-the-art OS designed especially for many-cores. Multi-core OSs like *Linux* can be used on many-cores but only with limited success [68]. Therefore, OS researchers have come up with some alternatives which we discuss below.

2.7.1. Corey

Corey [69] OS is designed for shared-cache based many-cores. The fundamental idea behind *Corey* is that tasks executing on a many-core must control the level of sharing of data, constructs and states with other tasks and the OS itself. This control should allow a multi-threaded task to scale up to much more cores than possible with traditional multi-core OS even when there is substantial inter-thread communication between the threads of the tasks. If the task does not desire any sharing, then it is also possible for *Corey* to completely step aside and induce no overhead on task execution. To transfer the sharing control to tasks, *Corey* arranges its data structures in such a way that by default only one core needs to use it and also provide interfaces with which other cores can access these structures if task explicitly wants to do so.

Corey works on *Intel Xeon* and *AMD Opteron* processors. *Corey* can be deployed on many-cores as both library- and kernel-based OS. *Corey* is based on three core abstractions namely Address Ranges, Kernel Cores and Shares. Address Ranges allows a task to define which part of address space associated with cores assigned to it are private to the cores and which part is shared with other cores and OS. Modification to private address ranges generates no invalidation coherency traffic on the NoC. This is in contrast to traditional multi-core OS where cores share the entire address space. A Kernel Core is a dedicated core that executes a specific kernel code. Non-Kernel codes must trigger the appropriate kernel core when they wish to perform a system call. This is in contrast to traditional multi-core OS where the core that executes the kernel code is the same where the system call is made requiring substantial data sharing. A Share is a data structure such as lookup tables for kernel objects which allow tasks to control the visibility of these objects to other cores at run-time.

Authors of *Corey* show that it results in superior performance for *MapReduce* and *Web-Server* tasks when compared to traditional multi-core OS even on multi-cores. They claim the performance gains would be even higher for these tasks when executing on many-cores.

2.7.2. Barrelfish

Barrelfish [70] OS is based on *multikernel* OS model designed for heterogeneous many-cores. It can operate on shared-memory based many-cores but its real target is message-passing based many-cores such as *SCC* describe in Section 2.6.2. It supports several different ISAs and thereby can work on processors from multiple vendors such as *Intel*, *AMD* and *ARM*.

Multikernel OS model is based on three core design principles. First principle stipulates that all communication in-between cores must be performed using explicit messages in contrast to implicit messages exchanged with cache-coherency. Second principle stipulates that OS structures must be delineated from the underlying hardware making the OS hardware neutral and also better suited for handling all kinds of heterogeneity. Third and final principal stipulates that global OS states must not be shared but replicated across all cores of the many-core.

Barrelfish has one lightweight kernel running on each core of the many-core. The entire many-core is seen as a network of independent cores which do not share any data. *Barrelfish* strives to bring ideas from distributed systems into the design of many-core OS. Even traditional OS subroutines such as schedulers in *Barrelfish* are implemented as distributed system processes that operate as one using message-passing.

Authors of *Barrelfish* show it to be as efficient as well-developed multi-core OS like *Windows* and *Linux* on shared-memory multi-cores. They also claim it to be much superior in performance to a multi-core OS on many-cores especially the ones based on message-passing.

2.7.3. Tesselation

Tesselation [71] is a many-core OS designed to execute real-time tasks on many-cores with QoS guarantees. It also supports execution of best-effort and interactive tasks. *Tesselation* is based on two core ideas namely Space-Time Partitioning and Two-Level Scheduling. *Tesselation* has more than 22000+ lines of code and can run on commercial *Intel x86* platforms as well as custom Field-Programmable Gate Array (FPGA) platforms.

The fundamental component of Space-time Partitioning in *Tesselation* is a Cell. Size of a Cell is defined by a fraction of guaranteed system resources such as cores, memory and bandwidth. Cells hence act as varisized virtual stand-alone processors themselves. A task executing under *Tesselation* is executed on several performance-isolated cells. Cells from a task are gang-scheduled together and communicate with each other using secure communication channels. *Tesselation* provides a task full control of the cells assigned to it. Task cells can also use the secure channels to communicate with OS subroutines such as device drivers which are themselves executing in their own cells.

Two-Level Scheduling in *Tesselation* uses a global scheduler and several cell-level schedulers to perform resource management for its underlying many-core. At the first level, the global scheduler distributes resources between the cells of different tasks based on the objectives of the overlying system while keeping the QoS of tasks under consideration. At the second level, a cell-level scheduler performs the resource management within the cell independent of other cell-level schedulers executing concurrently. A task is expected to provide cell-level schedulers to all the cells assigned to it. Authors of *Tesselation* claim that Two-Level Scheduling combined with Space-time Partitioning can scale up to a very large number of cores.

3. Many-Core Notations

We begin by presenting common notations used in this work to model many-cores.

- T represents set of $|T|$ tasks executing on many-core, indexed by t_i.
- C denotes set of $|C|$ cores in many-core, indexed by c_j.
- F indicates set of $|F|$ frequencies cores can operate using DVFS, indexed by f_k.
- $C_{t_i}^{f_k}$ means set of C_{t_i} cores allocated to task t_i operating at frequency f_k. We assume all the cores can perform independent DVFS. Still, multiple cores allocated to any given task always operate at the same frequency.
- C_{t_i} represents abridged notation in which all cores allocated to task t_i are assumed to be operating at the highest frequency of the many-core. Notation C_{t_i} simplifies the explanation of schedulers that do not employ DVFS.
- $\zeta(C_{t_i}^{f_k})$ represents Instruction per Cycle (IPC) of task t_i.
- $\rho(C_{t_i}^{f_k})$ represents Instruction per Second (IPS) of task t_i.
- $\alpha(C_{t_i}^{f_k})$ denotes DVFS-speedup of task t_i. Ratio of IPS of task t_i operating at frequency f_k to IPS of the task operating at the lowest frequency defines DVFS-speedup $\alpha(C_{t_i}^{f_k})$.
- $\beta(C_{t_i})$ means core-speedup of task t_i. Ratio of IPC of task t_i with $|C_{t_i}|$ cores allocated to IPC of the task with only one core allocated defines core-speedup $\beta(C_{t_i})$.
- $\gamma(C_{t_i})$ means core-slowdown of task t_i. Ratio of IPC of task t_i with the maximum cores allocated to IPC of the task with $|C_{t_i}|$ cores allocated defines core-slowdown $\gamma(C_{t_i})$.
- $S = \cup_{t_i} C_{t_i}^{f_k}$ denotes current state of allocations.
- $\zeta(S)$ represents aggregate IPC of all tasks in state S.
- $\rho(S)$ represents aggregate IPS of all tasks in state S.
- $\alpha(S)$ represents aggregate DVFS-speedup of all tasks in state S.
- $\beta(S)$ represents aggregate core-speedup of all tasks in state S.
- $\gamma(S)$ represents aggregate core-slowdown of all tasks in state S.

4. Many-Core Task Schedulers for Performance Maximization

This chapter introduces two schedulers with the objective of maximizing the performance of many-core.[1] Schedulers maximize performance by employing core-speedup[2] to accelerate tasks. Chapter 8 explores the use of DVFS to accelerate tasks. In this chapter, we use a particular kind of many-core called an adaptive many-core [36, 72]. Adaptive many-core can execute not just multi-threaded but also single-threaded tasks on multiple cores. Schedulers are not limited to adaptive many-cores but are also equally applicable to non-adaptive many-cores.

Adaptive many-core comprises of several simple base cores each with a small issue pipeline. Multiple cores allocated to a single-threaded task form a unified virtual core with a bigger issue pipeline which can extract more ILP. Multiple cores allocated to multi-threaded task extract Thread Level Parallelism (TLP) by co-executing threads of the task in parallel on allocated cores. The number of cores allocated to the single-threaded and multi-threaded task should depend upon their exploitable ILP and TLP potential, respectively. Many-core puts a task without any allocated core to sleep. The above formulation applies to several adaptive multi/many-core architectures with minimal modifications. We avoid adding constraints imposed by any specific architecture to the many-core formulation described above to keep schedulers proposed in this chapter to be generic.

Homogeneous non-adaptive many-cores are a special case of formulation presented above wherein only multi-threaded tasks can be allocated multiple cores but not single-threaded tasks. Figure 4.1 illustrates a generic adaptive many-core where varisized virtual cores are executing single-threaded tasks with different levels of ILP, along with two threads of a multi-threaded task executing in parallel on two different cores. There is also a sleeping task that is awaiting cores to become available to resume its execution.

The number of possible combinations in which scheduler can distribute cores of a generic adaptive many-core amongst tasks is analogous to the integer partition problem in number theory [73] as shown in Figure 4.2. Therefore, search-space for developing a performance maximizing schedule expands combinatorically with the increase in the number of cores.

4.0.1. Dynamic Scheduling Motivation

Figure 4.3a shows how processing requirements of two tasks – *mcf* and *bizp2* – vary over time. Furthermore, ILP/TLP exploitation potential in tasks also change during their execution resulting in time-varying core-speedups on multiple cores as shown in Figure 4.3b. The scheduler must transfer cores from task entering low-performance phase to task entering high-performance phase to keep many-core operating at peak performance. Note that only malleable tasks[3] allow transfer of cores amongst themselves at run-time. Figure 4.4 shows how a scheduler improves average throughput by 13.28% – measured as the sum of IPC of tasks – by performing dynamic reallocations in a 4-core processor running two tasks (*mcf* and *bizp2*) compared to a static scheduler, where scheduler statically allocates two cores to each task.

[1]The work presented in this chapter was originally published in [3] ©2016 *ACM* and [4] ©2016 *IEEE*.
[2]Refer Chapter 3 for the definition of core-speedup.
[3]Refer Chapter 2.2 for the definition of a malleable task.

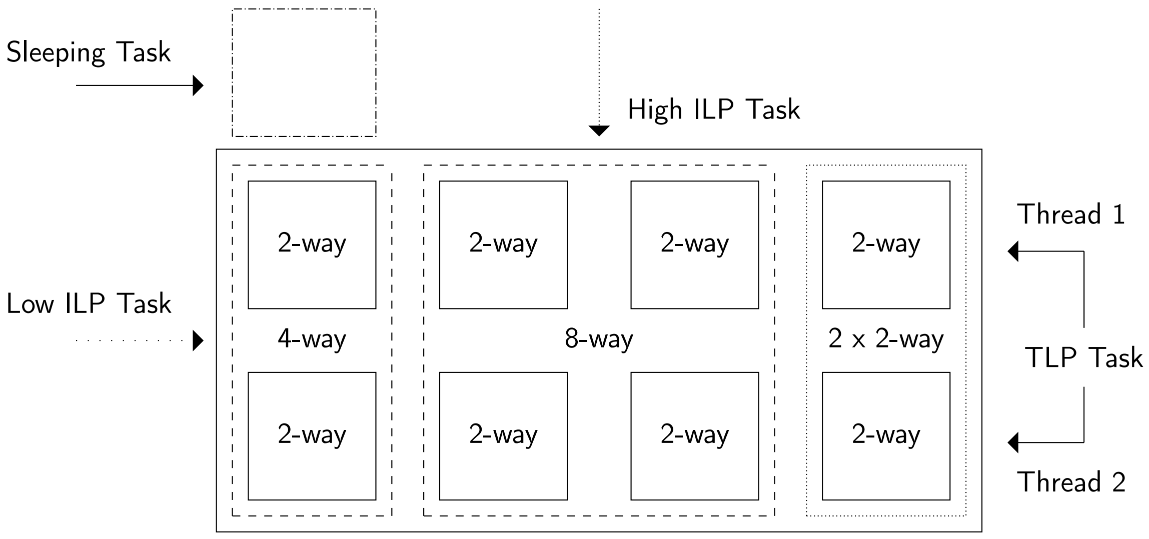


Figure 4.1.: Generic adaptive many-core with eight cores and three tasks.

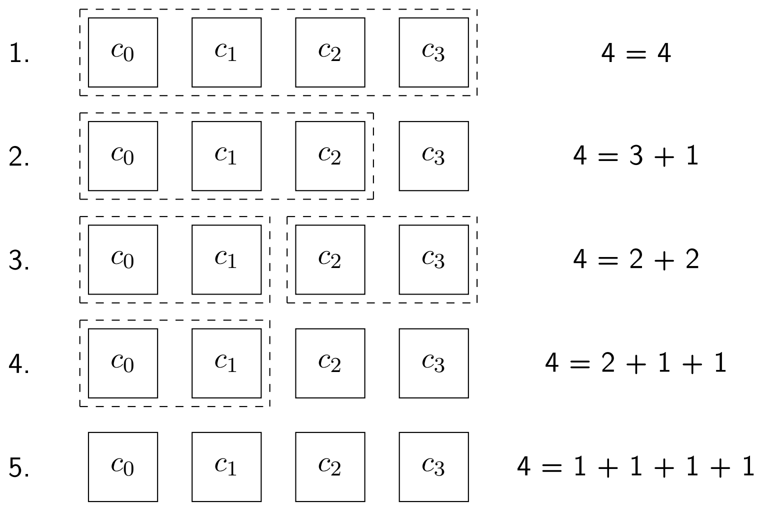
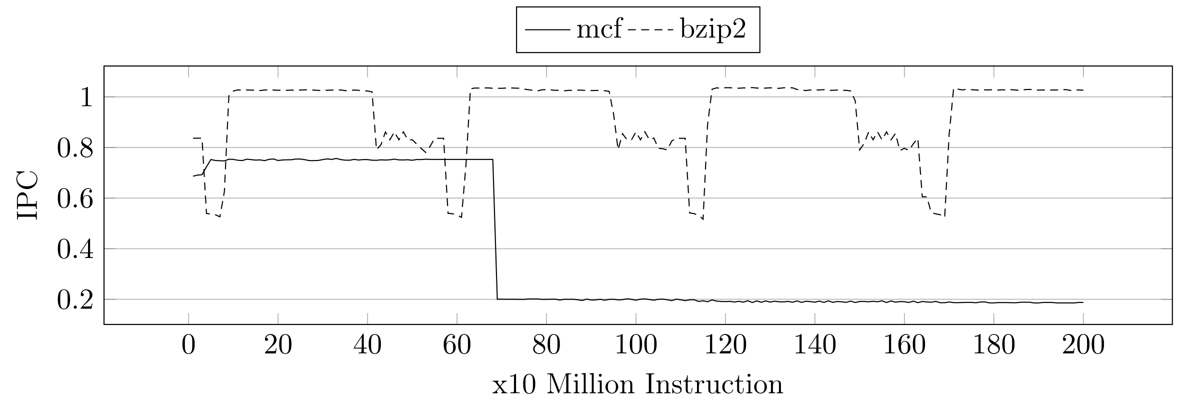


Figure 4.2.: Analogy between integer partitions and possible core allocations on a 4-core processor.

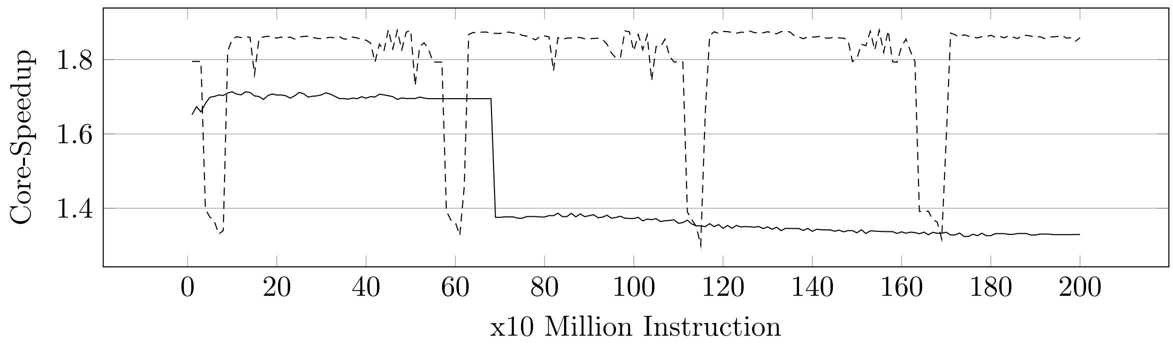
Authors in [74] proposed a scheduler based on DP that can optimally solve the problem of performance maximization as studied in this chapter within polynomial time. DP is an inherently centralized computation-intensive algorithm that uses only one core of many-core for performing scheduling-related computations. Therefore, DP which works well in multi-cores cannot be scaled up in many-cores due to increase in the search-space.

4.0.2. Novel Contributions

As an alternative to DP, we propose two new schedulers in this chapter. First is a distributed scheduler called Distributed Performance Many-Core Scheduler (*DPMS*) and second one is a centralized greedy scheduler called Greedy Performance Many-Core Scheduler (*GPMS*). The schedulers are theoretically proven to be optimal. The schedulers provide performance equivalent to a scheduler based on DP but with several times less scheduling overheads. Therefore, both proposed schedulers are better suited for performing task scheduling on many-cores at run-time than the DP based scheduler.



(a) IPC over execution



(b) Core-speedup with two cores over execution

Figure 4.3.: Execution profiles of *mcf* and *bzip2*.

4.1. Distributed Scheduler

We first present a distributed scheduler called *DPMS* in this section. Chapter 3 describes the common notations used to describe *DPMS*. We assume all tasks to be malleable in this chapter. Let throughput measured as aggregate IPC $\zeta(S)$ be the measure of performance.

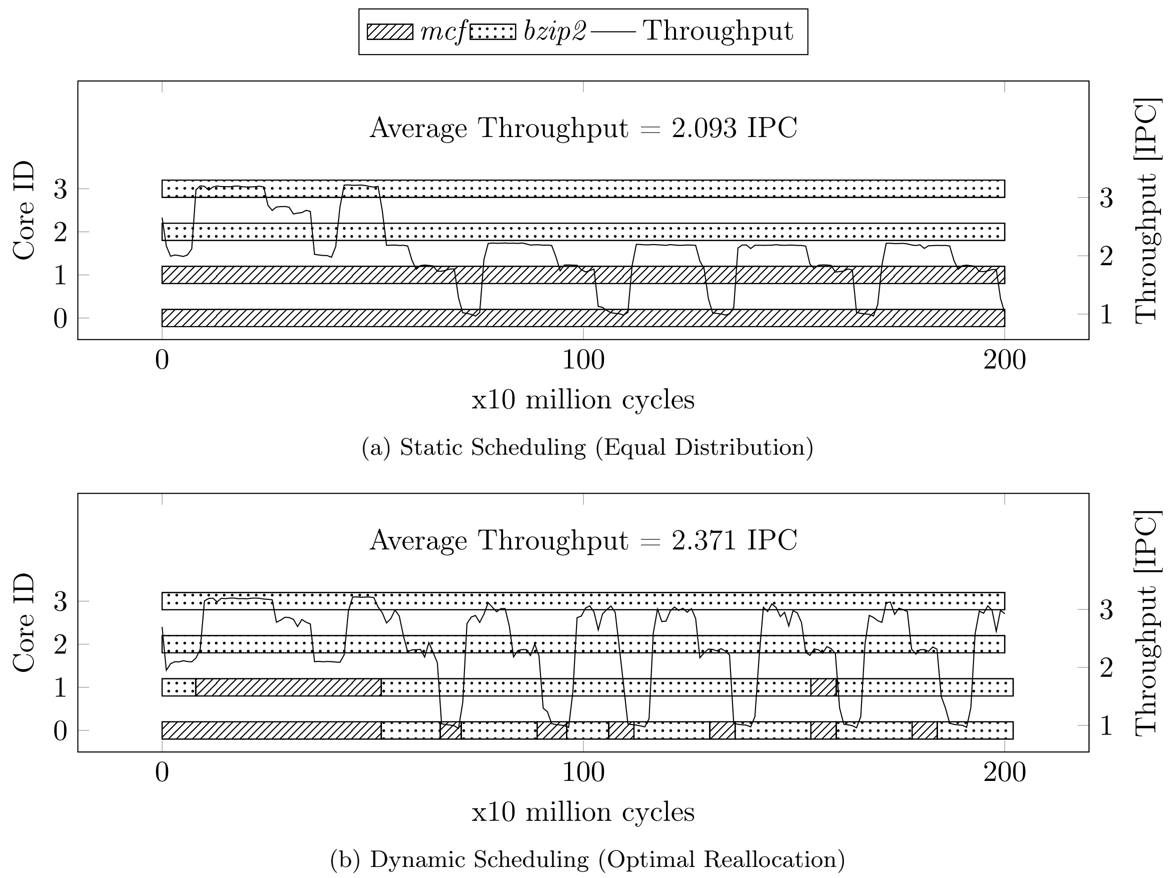
We define the utility of core $c_j \in C_{t_i}$ represented by symbol $u_{c_j}(C_{t_i})$ as the increase in IPC core c_j brings to task t_i. Model presented in [75] inspires the design of utility.

$$u_{c_j}(C_{t_i}) \quad = \quad \zeta(C_{t_i}) - \zeta(\{C_{t_i} - c_j\}) \tag{4.1}$$

4.1.1. Execution Flow

Figure 4.5 shows execution flow for *DPMS*. Initially, all cores are either unallocated or equally distributed amongst tasks. At the beginning of each scheduling epoch[4], series of rounds take place to determine allocations for that epoch. In each round, all cores evaluate benefits of joining every other task against benefits of staying with their current tasks based on their utilities calculated using Equation (4.1). Cores then myopically take decisions to move amongst tasks to increase their utilities. It suffices for one core allocated to given task to perform utility calculations and take decisions on behalf of all other cores allocated to that task to reduce overheads. Core c_j will move from task t_i to task $t_{i'}$ if utility $u_{c_j}(C_{t_{i'}} \cup \{c_j\})$ for moving is higher than its current utility $u_{c_j}(C_{t_i})$ of staying. For estimating utilities, cores use IPC

[4]Refer Chapter 3 for the definition of scheduling epoch.



(a) Static Scheduling (Equal Distribution)

(b) Dynamic Scheduling (Optimal Reallocation)

Figure 4.4.: Dynamic scheduling can provide additional throughput over static scheduling.

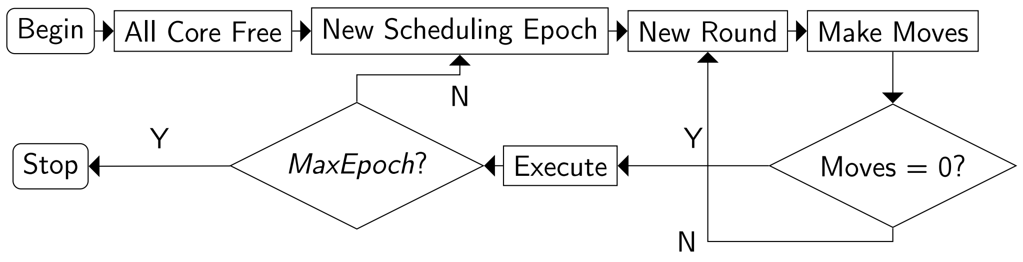


Figure 4.5.: Execution flow for *DPMS*.

prediction techniques for adaptive many-cores developed by authors of [76]. Core reallocation rounds stop when no more moves are possible. *DPMS* then execute tasks with cores allocated to them in the final round. Many-core halts when execution reaches a user-defined *MaxEpoch*.

4.1.2. Execution Characteristics

Figure 4.6 shows average core-speedup for different tasks when allocated a different number of cores. Average core-speedup is both monotonically increasing and concave. Concavity arises because of saturation of exploitable ILP or TLP in tasks with the increase in the number of allocated cores. Still, the addition of every allocated core brings a non-negative increase in IPC of the associated task. We observed similar behavior for all tasks listed in Table 4.1. Therefore, allocations follow the basic economic law of diminishing returns. This behavior by definition makes IPC of task non-decreasing and utility of core allocated to that task sub-modular.

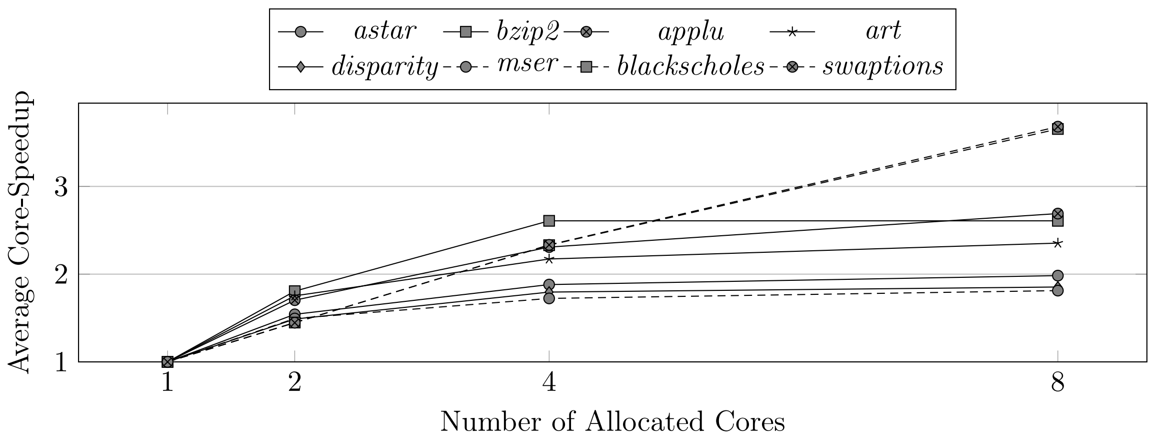


Figure 4.6.: Average core-speedup of a different tasks when allocated different number of cores.

$$\zeta(C_{t_i}) \;\geq\; \zeta(C'_{t_i}) \text{ if } |C_{t_i}| \;\geq\; |C'_{t_i}| \tag{4.2}$$

$$\zeta(C_{t_i}) - \zeta(\{C_{t_i} - c_j\}) \;\leq\; \zeta(C'_{t_i}) - \zeta(\{C'_{t_i} - c_j\}) \text{ if } |C_{t_i}| \;\geq\; |C'_{t_i}| \tag{4.3}$$

Figure 4.7 shows standard boxplot for core-speedup for all of our single-threaded tasks with two cores allocated. Many tasks exhibit high entropy over their execution. This entropy provides substantial opportunities for performance optimization. Tasks also differ among themselves in absolute core-speedup they obtain. Interestingly, some of the tasks like *h264ref* can sometimes obtain more than two times core-speedup when allocated only two cores. This behavior happens because allocating two cores to task can on some occasions opens up a secondary bottleneck such as allocated memory resulting in super-linear core-speedup.

4.1.3. Equilibrium

Equilibrium for *DPMS* means that if many-core's load does not change, then many-core will achieve oscillation-free allocations. In equilibrium, no core has the incentive to move.

$$\forall_{(c_j \in C_{t_i}) \in S} \quad \nexists_{C_{t_{i'}} \in S} \;\; s.t. \;\; u_{c_j}(C_{t_{i'}} \cup \{c_j\}) \;>\; u_{c_j}(C_{t_i}) \tag{4.4}$$

We choose to prove equilibrium using *Sharkovsky* theorem [77], by denying the existence of a two-period cycle in *DPMS*. Let $C_{t_x} \rightleftharpoons C_{t_y}$ denote allocation of tasks t_x and t_y in equilibrium. Let c_x and c_y represent cores that are part of allocations C_{t_x} and C_{t_y}, respectively. Based on Equations (4.1) and (4.4) the following equations hold.

$$\zeta(C_{t_x}) - \zeta(\{C_{t_x} - c_x\}) \;\geq\; \zeta(C_{t_y} \cup \{c_x\}) - \zeta(C_{t_y}) \tag{4.5}$$

$$\zeta(C_{t_y}) - \zeta(\{C_{t_y} - c_y\}) \;\geq\; \zeta(C_{t_x} \cup \{c_y\}) - \zeta(C_{t_x}) \tag{4.6}$$

Lemma 1. *In equilibrium $C_{t_x} \rightleftharpoons C_{t_y}$, if core c_x does not want to move, then core set $\{c_x, c_{x'}\} \subseteq C_{t_x}$ will also not want to move.*

Proof. From Equation (4.3) we know,

$$\begin{aligned} \zeta(C_{t_x} - \{c_x\}) - \zeta(C_{t_x} - \{c_x, c_{x'}\}) \;&\geq\; \zeta(C_{t_y} \cup \{c_x, c_{x'}\}) - \zeta(C_{t_y} \cup \{c_x\}) \\ \zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_x, c_{x'}\}) \;&\geq\; \zeta(C_{t_y} \cup \{c_x, c_{x'}\}) - \zeta(C_{t_y})[\because \text{Eq. (4.5)}] \end{aligned}$$

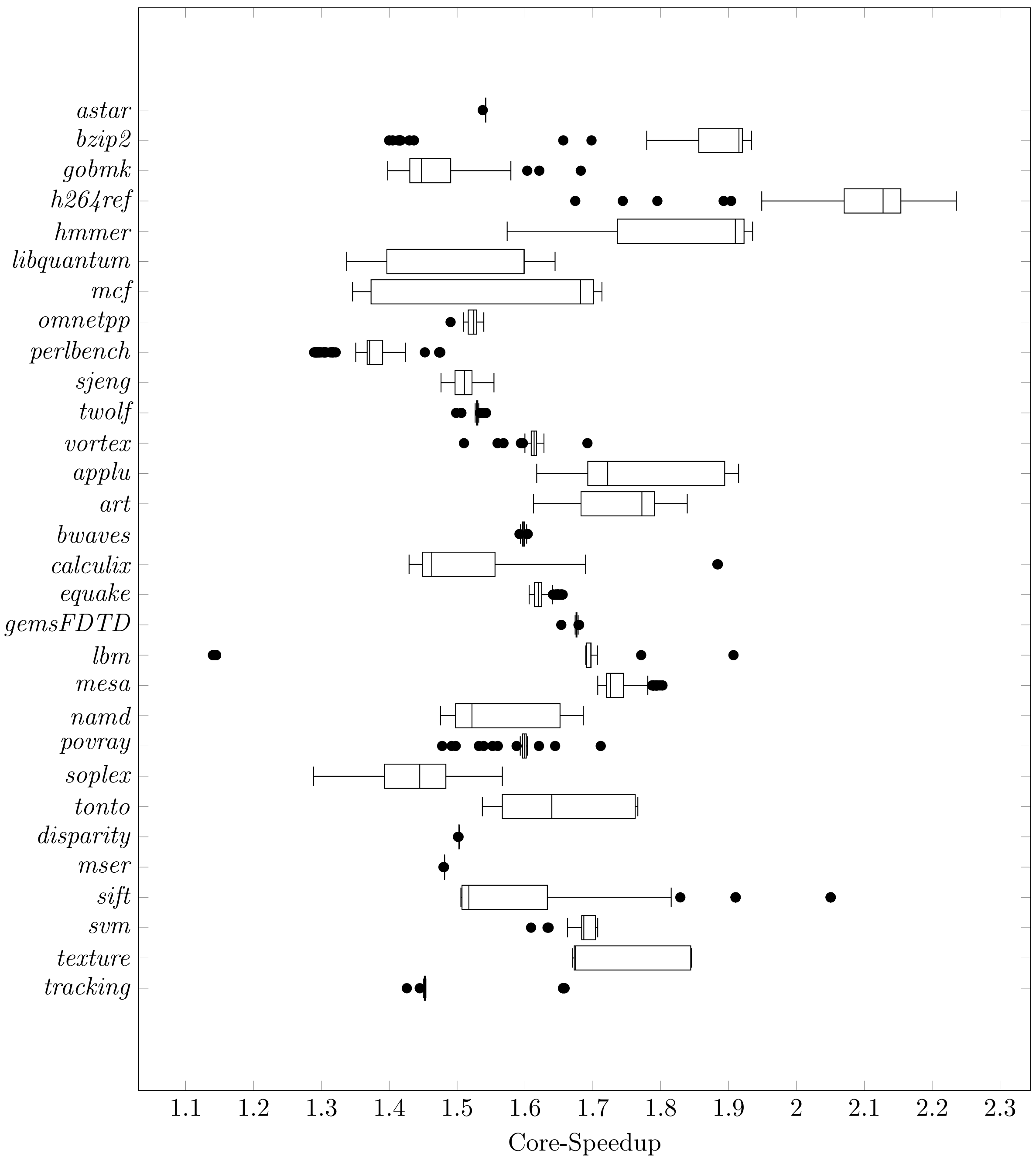


Figure 4.7.: Entropy in core-speedup for different tasks when allocated two cores.

Above result can be extended to any size core set containing core c_x derivable from allocation C_{t_x}. This lemma then implies that results shown for movement of single core allocated to given task will also extend to movement of the subset of cores allocated to that task; hence proved. $\square$

Lemma 2. *Equilibrium $C_{t_x} \rightleftharpoons C_{t_y}$ will continue to hold with the addition of core c_z.*

Proof. Without loss of generality, let us say utility for core c_z is higher on joining allocation C_{t_x} than joining allocation C_{t_y}. Therefore, core c_z joins allocation C_{t_x}. By this assumption following equation is true,

$$\zeta(C_{t_x} \cup \{c_z\}) - \zeta(C_{t_x}) \;\geq\; \zeta(C_{t_y} \cup \{c_z\}) - \zeta(C_{t_y})$$

By adding and subtracting core c_z from allocation C_{t_x} we get,

$$\zeta(C_{t_x} \cup \{c_z\}) - \zeta(C_{t_x} \cup \{c_z\} - \{c_z\}) \;\geq\; \zeta(C_{t_y} \cup \{c_z\}) - \zeta(C_{t_y})$$

After joining allocation C_{t_x}, core c_z now has the same utility as core c_x. Hence, we can say

$$\zeta(C_{t_x} \cup \{c_z\}) - \zeta(C_{t_x} \cup \{c_z\} - \{c_x\}) \;\geq\; \zeta(C_{t_y} \cup \{c_x\}) - \zeta(C_{t_y})$$

Therefore, core c_x would not move from allocation C_{t_x} even after core c_z joins allocation C_{t_x}. From Equation (4.3) we know,

$$\begin{aligned}
\zeta(C_{t_x} \cup \{c_y\}) - \zeta(C_{t_x}) \;&\geq\; \zeta(C_{t_x} \cup \{c_z, c_y\}) - \zeta(C_{t_x} \cup \{c_z\}) \\
\zeta(C_{t_y}) - \zeta(C_{t_y} - \{c_y\}) \;&\geq\; \zeta(C_{t_x} \cup \{c_z, c_y\}) - \zeta(C_{t_x} \cup \{c_z\})[\because \text{Eq. (4.6)}]
\end{aligned}$$

Therefore, core $a_y \in C_{t_y}$ would also not move from its current allocation; hence proved.

□

Lemma 3. *Equilibrium $C_{t_x} \rightleftharpoons C_{t_y}$ is reattained in no more than one move with the removal of core $c_{x'} \in C_{t_x}$.*

Proof. Without loss of generality, this result would hold even if core $c_{y'} \in C_{t_y}$ was removed from equilibrium $C_{t_x} \rightleftharpoons C_{t_y}$ instead. From Equation (4.3) we get,

$$\begin{aligned}
\zeta(C_{t_x} - \{c_{x'}\}) - \zeta(C_{t_x} - \{c_{x'}, c_x\}) \;&\geq\; \zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_x\}) \\
\zeta(C_{t_x} - \{c_{x'}\}) - \zeta(C_{t_x} - \{c_{x'}, c_x\}) \;&\geq\; \zeta(C_{t_y} \cup \{c_x\}) - \zeta(C_{t_y})[\because \text{Eq. (4.5)}]
\end{aligned}$$

Therefore, core c_x would not move from allocation C_{t_x} even after its fellow core $c_{x'}$ leaves.

Now a move is possible from allocation C_{t_y} to allocation C_{t_x} since after departure of core $c_{x'}$ utility of joining allocation C_{t_x} has increased for cores. We assume core $c_{y'} \in C_{t_y}$ makes that move. From Equation (4.6) we know,

$$\begin{aligned}
\zeta(C_{t_y}) - \zeta(C_{t_y} - \{c_y\}) \;&\geq\; \zeta(C_{t_x} \cup \{c_y\}) - \zeta(C_{t_x}) \\
\zeta(C_{t_y} - \{c_{y'}\}) - \zeta(C_{t_y} - \{c_{y'}, c_y\}) \;&\geq\; \zeta(C_{t_y}) - \zeta(C_{t_y} - \{c_y\})[\because \text{Eq. (4.3)}]
\end{aligned}$$

Since, allocation C_{t_x} is equivalent to allocation $(C_{t_x} - \{c_{x'}\} \cup \{c_{y'}\})$ we obtain,

$$\zeta(C_{t_y} - \{c_{y'}\}) - \zeta(C_{t_y} - \{c_{y'}, c_y\}) \;\geq\; \zeta(C_{t_x} - \{c_{x'}\} \cup \{c_{y'}, c_y\}) - \zeta(C_{t_x} - \{c_{x'}\} \cup \{c_{y'}\})$$

Therefore, core c_y would also not move from its current allocation C_{t_y}; hence proved.

□

Lemma 4. *Addition of allocation C_{t_z} to equilibrium $C_{t_x} \rightleftharpoons C_{t_y}$ results in new equilibrium $C_{t_x} \rightleftharpoons C_{t_y} \rightleftharpoons C_{t_z}$.*

Proof. Without loss of generality, let us assume only two allocations interact at a time, beginning with allocations C_{t_x} and C_{t_z}. By Lemmas 2 and 3 when allocations C_{t_x} and C_{t_z} unidirectionally exchange cores until reaching equilibrium $C_{t_x} \rightleftharpoons C_{t_z}$, equilibrium $C_{t_x} \rightleftharpoons C_{t_y}$ continues to hold. Allocations C_{t_y} and C_{t_z} then unidirectionally exchange cores until reaching equilibrium $C_{t_y} \rightleftharpoons C_{t_z}$, while equilibrium $C_{t_x} \rightleftharpoons C_{t_y}$ continues to hold. Thereby, allocations reach new equilibrium $C_{t_x} \rightleftharpoons C_{t_y} \rightleftharpoons C_{t_z}$; hence proved.

□

Theorem 1. DPMS *will achieve oscillation-free equilibrium from any given initial state S in $O(|T|)$ number of rounds.*

Proof. From any initial state, any allocation is in equilibrium with itself when considered in isolation. Equilibrium can be iteratively extended using Lemma 4 to any number of allocations. Furthermore, Lemmas 2 and 3 show that cycles of period two cannot exist in *DPMS* because the exchange of cores between allocations is always unidirectional. Since period two cycle is most straightforward to create, a corollary of *Sharkovsky* theorem [77] says that in a dynamic system if period two cycle does not exist then any higher period cycle also does not exist.

Additionally, since no cycle exists a core cannot return to allocation, it has previously left. Thus, any core can make a maximum of $O(|T|)$ jumps (including incorrect myopic movements) before many-core reaches equilibrium; hence proved.

□

Theorem 2. *In equilibrium, allocations are optimal.*

Proof. In equilibrium $C_{t_x} \rightleftharpoons C_{t_y}$, let us assume throughput $\zeta(S)$ is not optimal and there exists another permutation of allocations C'_{t_x} and C'_{t_y} that are instead optimal. By definition of throughput in Chapter 3 we know,

$$\zeta(C'_{t_x}) + \zeta(C'_{t_y}) \; > \; \zeta(C_{t_x}) + \zeta(C_{t_y}) \tag{4.7}$$

We assume all cores are part of either allocation and without loss of generality due to Lemma 1 we can say,

$$C'_{t_x} \; = \; C_{t_x} - \{c_x\} \text{ and } C'_{t_y} \; = \; C_{t_y} \cup \{c_x\}$$

Now since we have equilibrium $C_{t_x} \rightleftharpoons C_{t_y}$, from Equation 4.5 we get

$$\begin{aligned} \zeta(C_{t_x}) - \zeta(\{C_{t_x} - c_x\}) \; &\geq \; \zeta(C_{t_y} \cup \{c_x\}) - \zeta(C_{t_y}) && (4.8) \\ \implies \zeta(C_{t_x}) - \zeta(C'_{t_x}) \; &\geq \; \zeta(C'_{t_y}) - \zeta(C_{t_y}) && (4.9) \\ \implies \zeta(C_{t_x}) + \zeta(C_{t_y}) \; &\geq \; \zeta(C'_{t_x}) + \zeta(C'_{t_y}) \end{aligned}$$

Above equation is contradicting Equation (4.7); hence proved.

□

4.1.4. Dynamics Illustration

We now illustrate autonomous dynamics that occur under *DPMS*. Dynamics is series of best response myopic moves made by cores converging towards equilibrium. Figure 4.8 shows simple dynamics that can occur on a processor with four cores c_1, c_2, c_3, and c_4. Many-core starts execution with two tasks t_1 and t_2 that are allocated empty allocations C_{t_1} and C_{t_2}, respectively.

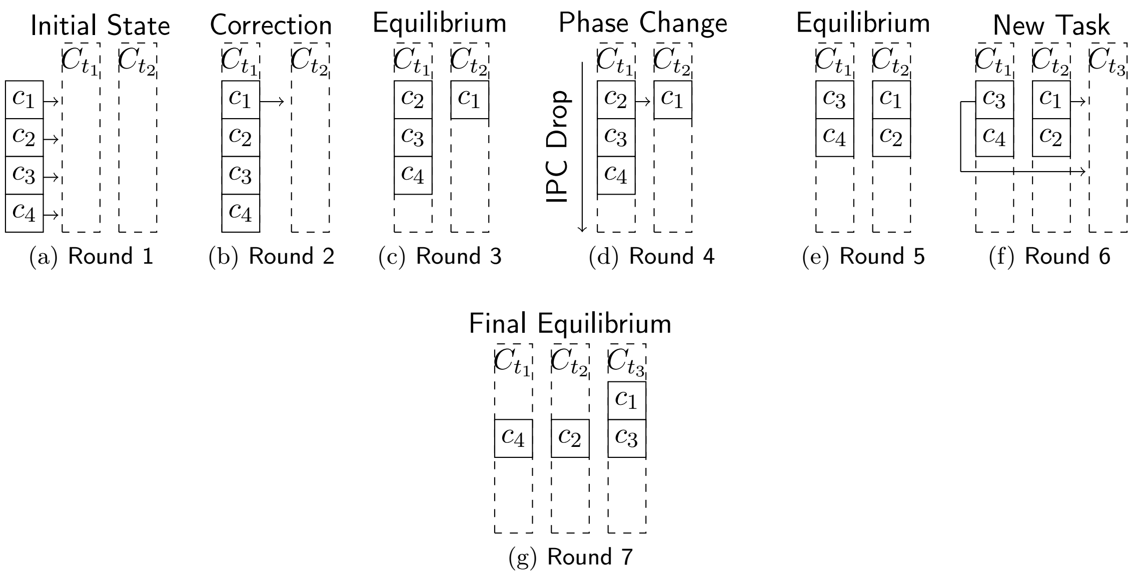


Figure 4.8.: Illustration showing the dynamics under *DPMS* on 4-core processor with three tasks.

Let task t_1 have higher throughput than task t_2. In equilibrium $C_{t_1} \rightleftharpoons C_{t_2}$, allocations C_{t_1} and C_{t_2} should be allocated three cores and one core, respectively.

Round 1 (Figure 4.8a): initially both allocations C_{t_1} and C_{t_2} are empty, and all cores are unbounded. All cores evaluate benefits of joining either one of two allocations. All cores then come to the same decision of joining allocation C_{t_1} associated with higher throughput task t_1.

Round 2 (Figure 4.8b): core c_1 evaluates the possibility of joining allocation C_{t_2} on behalf of all cores in allocation C_{t_1}. Allocation C_{t_1} already has three other cores allocated to it now, which core c_1 did not know will also join allocation C_{t_1} in Round 1. Core c_1 thereby concludes that it is better off joining allocation C_{t_2}.

Round 3 (Figure 4.8c): many-core reaches equilibrium as none of the cores want to move from their current allocations.

Round 4 (Figure 4.8d): task associated with allocation C_{t_1} enters a new phase, and its IPC decreases. New equilibrium should have two cores allocated to both allocations C_{t_1} and C_{t_2}. Core c_1 still does not want to move from allocation C_{t_2}, but core c_2 now decides to move to allocation C_{t_2} from its current allocation C_{t_1}.

Round 5 (Figure 4.8e): dynamics stop as many-core reattains equilibrium.

Round 6 (Figure 4.8f): another task t_3 with empty allocation C_{t_3} enters many-core and has the highest throughput. New trilateral equilibrium $C_{t_1} \rightleftharpoons C_{t_2} \rightleftharpoons C_{t_3}$ has one core allocated to allocations C_{t_1} and C_{t_2} each, with two cores allocated to allocation C_{t_3}. Note that both cores c_1 and c_3 move to allocation C_{t_3} in parallel from allocations C_{t_2} and C_{t_1}, respectively.

Round 7 (Figure 4.8g): dynamics come to a halt again in new equilibrium.

4.1.5. Complexity

Each round of *DPMS* requires all the cores in C to perform $|T|$ utility calculations as described in Equation (4.1). In the worst-case, equilibrium can take up to $O(|T|)$ rounds as per Theorem 1. Therefore, in total a maximum of $O(|C||T|^2)$ calculations are required to ensure the stability in worst-case. However, the processing overhead is distributed across all the cores resulting in $O(|T|^2)$ worst-case processing overhead per-core.

The studied problem can also be solved optimally using DP. DP has a centralized overhead of $O(|C||T|^2)$, which is difficult to parallelize [78]. Every time core-speedup or IPC of any task changes, the optimal schedule needs to be re-evaluated. In many-cores, where the number of tasks $|T| >> 1$, changes are nearly continuous, making online scheduler employing DP impractical. However, we can still utilize DP in this chapter to create theoretically optimal oracular scheduler for comparison.

DP has space overhead of $O(|C||T|)$, while in *DPMS* space overhead is $O(1)$. Under *DPMS*, O($|T|$) messages need to be broadcasted every round in worst-case. Thus, in worst-case $O(|T|^2)$ messages need to be transmitted every epoch. These messages can be transmitted with low overhead using NoC proposed for many-core architectures. *DPMS* can operate with partially correct (predicted) or incomplete information (NoC delays) to produce a near-optimal schedule, while DP requires accurate and complete knowledge at all times to operate optimally.

4.2. Greedy Scheduler

We now present a centralized greedy scheduler called *GPMS* for performance maximization in many-cores by using core-speedup to accelerate tasks. Chapter 3 describes the common notations used to describe *GPMS*. *GPMS* has even lower overheads than *DPMS* scheduler proposed in Section 4.1 while still providing equivalent performance.

The proposed greedy algorithm is composed of following sequential steps performed by *GPMS* before every scheduling epoch.

1. Assume allocation $C_{t_i} = \emptyset \; \forall t_i \in T$.
2. Sort all tasks in T in ascending order by using comparator $[\zeta(C_{t_i} \cup \{c_j\}) - \zeta(C_{t_i})]$, where core c_j is unallocated. Store sorted tasks in a queue.
3. Allocate core c_j to task t_x in front of the queue and update corresponding IPC $\zeta(C_{t_x})$ using IPC prediction models from [76].
4. Reposition task t_x according to updated IPC $\zeta(C_{t_x})$ in the sorted queue using binary search insertion.
5. Repeat Steps 3 and 4 to allocate all cores in C.
6. Execute tasks with greedy allocations.

4.2.1. Optimality

GPMS like all greedy algorithms is minimalistic in its approach and is quite easy to implement. Nevertheless, its real strength comes from its ability to provide optimal results. We now proceed to prove theoretical optimality of *GPMS*.

Theorem 3. *Greedy allocations under* GPMS *are optimal.*

Proof. We prove the theorem using proof by induction. Let us assume *GPMS* preforms allocations $\langle C_{t_1}, C_{t_2}, ..., C_{t_{|T|}} \rangle$.

Base Case: Assume allocations $\langle C_{t_1}, C_{t_2}, C_{t_x} - \{c_j\}, ..., C_{t_y} \cup \{c_j\}, ..., C_{t_{|T|}} \rangle$ instead to be optimal in which *GPMS* should have allocated core c_j to task t_y instead of task t_x. Now for optimal allocations to be better than greedy allocations following equation must hold.

$$\zeta(C_{t_y} \cup \{c_j\}) - \zeta(C_{t_y}) > \zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_j\}) \tag{4.10}$$

The above equation says that benefit (regarding increased IPC) of allocating additional core c_j to task t_y must outweigh loss (regarding decreased IPC) of taking away that core c_j from task t_x under optimal allocations.

DPMS does not reconsider its allocations. So, the suboptimal decision of allocating core c_j to task t_x with allocation $C_{t_x} - \{c_j\}$ happens when task t_y had either allocation C_{t_y} or allocation C'_{t_y} such that $|C'_{t_y}| < |C_{t_y}|$.

If suboptimal allocation happened when task t_y had allocation C_{t_y}, then by greedy design it implies the following relation.

$$\zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_j\}) \geq \zeta(C_{t_y} \cup \{c_j\}) - \zeta(C_y)$$

Above equation is in contradiction to Equation (4.10).

On the other hand, if suboptimal allocation happened when task t_y had allocation C'_{t_y} such that $|C'_{t_y}| < |C_{t_y}|$ then it implies the following relation.

$$\zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_j\}) \geq \zeta(C'_{t_y} \cup \{c_j\}) - \zeta(C'_{t_y})$$

But we know from concavity Equation (4.3),

$$\begin{aligned} &\zeta(C'_{t_y} \cup \{c_j\}) - \zeta(C'_{t_y}) \geq \zeta(C_{t_y} \cup \{c_j\}) - \zeta(C_{t_y}) \\ \implies &\zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_j\}) \geq \zeta(C_{t_y} \cup \{c_j\}) - \zeta(C_{t_y}) \end{aligned}$$

Above is a contradiction to Equation (4.10). Hence, we prove that base case is optimal.

Step Case: Suppose allocations $\langle C_{t_1}, C_{t_2}, C_{t_x} - \{c_j, c_{j'}\}, ..., C_{t_y} \cup \{c_j\}, C_{t_z} \cup \{c_{j'}\}, ..., C_{t_{|T|}}\rangle$ instead to be optimal in which *GPMS* should have allocated cores c_j and $c_{j'}$ to tasks t_y and t_z instead of task t_x, respectively. Without loss of generality, the proof will also hold if both cores from task t_x were allocated to only tasks t_y or t_z instead. Now for above optimal allocations to be better than greedy allocations the following equation must hold.

$$\zeta(C_{t_y} \cup \{c_j\}) - \zeta(C_{t_y}) + \zeta(C_{t_z} \cup \{c_{j'}\}) - \zeta(C_{t_z}) > \zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_j, c_{j'}\}) \tag{4.11}$$

As argued in the base case, the suboptimal decision of allocating core c_j to task t_x with allocation $C_{t_x} - \{c_j, c_{j'}\}$ happened when task t_y had allocation $C'_{t_y} \subseteq C_{t_y}$. Therefore, by greedy design the following relation must hold.

$$\zeta(C_{t_x} - \{c_{j'}\}) - \zeta(C_{t_x} - \{c_j, c_{j'}\}) \quad \geq \quad \zeta(C'_{t_y} \cup \{c_j\}) - \zeta(C'_{t_y}) \tag{4.12}$$

Similarly, the suboptimal decision of allocating second core $c_{j'}$ to task t_x with allocation $C_{t_x} - \{c_{j'}\}$ happened when task t_z had allocation $C'_{t_z} \subseteq C_{t_z}$. Therefore, by greedy design the following relation must hold.

$$\zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_{j'}\}) \quad \geq \quad \zeta(C'_{t_z} \cup \{c_{j'}\}) - \zeta(C'_{t_z}) \tag{4.13}$$

By adding Equations (4.12) and (4.13) we get,

$$\zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_j, c_{j'}\}) \geq \zeta(C'_{t_y} \cup \{c_j\}) - \zeta(C'_{t_y}) + \zeta(C'_{t_z} \cup \{c_{j'}\}) - \zeta(C'_{t_z})$$

But we know from concavity Equation (4.3),

$$\begin{aligned}\zeta(C'_{t_y} \cup \{c_j\}) - \zeta(C'_{t_y}) &\geq \zeta(C_{t_y} \cup \{c_j\}) - \zeta(C_{t_y}) \\ \zeta(C'_{t_z} \cup \{c_{j'}\}) - \zeta(C'_{t_z}) &\geq \zeta(C_{t_z} \cup \{c_{j'}\})) - \zeta(C_{t_z})\end{aligned}$$

Therefore, we can say

$$\zeta(C_{t_x}) - \zeta(C_{t_x} - \{c_j, c_{j'}\}) \geq \zeta(C_{t_y} \cup \{c_j\}) - \zeta(C_{t_y}) + \zeta(C_{t_z} \cup \{c_{j'}\}) - \zeta(C_{t_z})$$

Above is in contradiction to Equation (4.11). Hence, we prove our step case to be optimal.

Assumption Case: We assume greedy allocations are optimal till cores in core set $C_n \subseteq C_{t_x}$ are removed from task t_x and distributed among remaining tasks in any combination. Following relationship is assumed to be true.

$$\zeta(C_{t_x}) - \zeta(C_{t_x} - C_n) \geq \zeta(C_{t_y} \cup C_n) - \zeta(C_{t_y}) + ... + \zeta(C_{t_{|T|}} \cup C_{n_{|T|}}) - \zeta(C_{t_{|T|}}) \quad (4.14)$$

where $C_n \cup ... \cup C_{n_{|T|}} = C_n$.

Induction Case: We assume in optimal allocations $(n+1)^{th}$ core c_j is removed from task t_x and given to task t_y. Assumption case holds for previously removed core set C_n. Optimal allocations are then better than greedy allocations if the following equation holds.

$$\zeta(C_{t_y} \cup C_n \cup \{c_j\}) - \zeta(C_{t_y}) + ... + \zeta(C_{t_{|T|}} \cup C_{n_{|T|}}) - \zeta(C_{t_{|T|}}) \geq \zeta(C_{t_x}) - \zeta(C_{t_x} - C_n - \{c_j\}) \quad (4.15)$$

Since *GPMS* chooses to allocate core c_j to task t_x with allocation $C_{t_x} - C_n - \{c_j\}$ instead of task t_y with allocation $C_{t_y} \cup C_n$, by greedy design the following relationship must hold.

$$\zeta(C_{t_x} - C_n) - \zeta(C_{t_x} - C_n - \{c_j\}) \geq \zeta(C_{t_y} \cup C_n \cup \{c_j\}) - \zeta(C_{t_y} \cup C_n)$$

Adding Equation (4.14) to above equation we get,

$$\zeta(C_{t_x}) - \zeta(C_{t_x} - C_n - \{c_j\}) \geq \zeta(C_{t_y} \cup C_n \cup \{c_j\}) - \zeta(C_{t_y}) + ... + \zeta(C_{t_{|T|}} \cup C_{n_{|T|}}) - \zeta(C_{t_{|T|}})$$

The above equation is in contradiction to Equation (4.15), proving that induction step is also optimal. Hence, greedy allocations under *GPMS* are proven optimal by induction.

□

4.2.2. Complexity

Under *GPMS*, sorting in Step 2 has processing overhead of $O(|T| \lg |T|)$. Additionally, binary search in Step 4 has processing overhead of $O(\lg |T|)$. Since Step 4 is repeated $|C|$ times, total processing overhead of *GPMS* is $O(|T| \lg |T| + |C| \lg |T|)$ or $O(\max\{|C|, |T|\} \lg |T|)$. This overhead is significantly less than $O(|C||T|^2)$ processing overhead of DP and even per-core processing overhead of *DPMS* of $O(|T|^2)$.

Furthermore, *GPMS* requires maintenance of only one queue data structure with space overhead of $O(T)$. On another hand, DP has significantly higher space overhead of $O(CT)$. Space overhead of *GPMS* is though bigger than $O(1)$ space overhead of *DPMS*. Communication overhead of *GPMS* is $O(1)$ similar to DP due to its centralized nature.

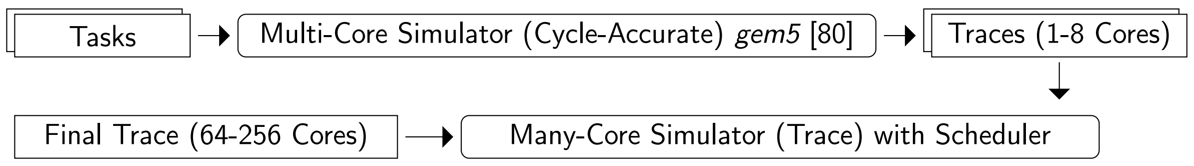

Tasks
Multi-Core Simulator (Cycle-Accurate) gem5 [80]
Traces (1-8 Cores)
Final Trace (64-256 Cores)
Many-Core Simulator (Trace) with Scheduler

Figure 4.9.: Experimental setup used in evaluation of *DPMS* and *GPMS*.

Table 4.1.: List of all tasks used in evaluation of *DPMS* and *GPMS*.

Type	*Task Name*
Integer	*astar, bzip2, gobmk, h264ref, hmmer, mcf, omnetpp, perlbench, sjeng, twolf, vortex*
Float	*art, bwaves, calculix, equake, gemsfdtd, lbm, namd, povray, tonto*
Vision	*disparity, mser, sift, svm, texture, tracking*
Parallel	*blackscholes, cholesky, fmm, fluidanimate, lu, radix, radiosity, swaptions, streamcluster, water-sp*

4.2.3. Fairness

Optimal performance does not translate into optimal fairness. In fact, performance and fairness are often contradictory goals [79]. In a work complementary to work presented in this chapter, we present a distributed scheduler for fair scheduling in many-cores in Chapter 5.

4.3. Experimental Evaluations

4.3.1. Experimental Setup

We use a two-stage adaptive many-core simulator for evaluations as shown in Figure 4.9. We use simulators because no real-world adaptive many-core platform is available at present. In the first stage, we use cycle-accurate *gem5* simulator [80] with up to eight cores with *ARMv7* ISA. Each core is two-way out-of-order core with separate 64 KB L1 instruction and data cache. All cores share 2 MB unified L2 cache. L1 caches are 4-way associative, while the L2 cache is 8-way associative with all caches having line size of 64 bytes.

Multi-threaded tasks can run directly on the simulator. For single-threaded tasks, we modify *gem5* simulator to model *Bahurupi* adaptive multi-core architecture [14] that can execute a single-threaded task on a virtual core of at most eight cores. With the increase in every core in simulated multi-core, cycle-accurate simulation time starts to increase exponentially, which makes cycle-accurate many-core simulations (with hundreds of cores) timewise infeasible. To bypass this limitation, we first collect isolated execution traces of tasks from the cycle-accurate simulator, albeit restricted to virtual core size of at most eight. We use a second trace-driven simulator that operates on these execution traces to model adaptive many-core with up to 256 cores. Akin to other trace-driven simulators, our simulations also cannot capture complex behavior arising due to shared resource contentions in multi-program execution [81].

We create workloads out of 26 single-threaded, and ten multi-threaded tasks as listed in Table 4.1. Single-threaded tasks comprise of integer, floating point and vision tasks from *SPEC* [82, 83] and *SD-VBS* suites [84]. Multi-threaded tasks come from *PARSEC* [85] and *SPLASH-2* [86] suites. In total, we work with 36 tasks listed in Table 4.1. Tasks are compiled using *ARM* cross-compiler provided by *gcc* with “O2” optimization flag enabled. Syscall Emulation (SE) mode is used to execute tasks. *SPEC* and *SD-VBS* tasks execute with “ref” and “full-hd” inputs, respectively. *PARSEC* and *SPLASH-2* tasks execute with “sim-small” input.

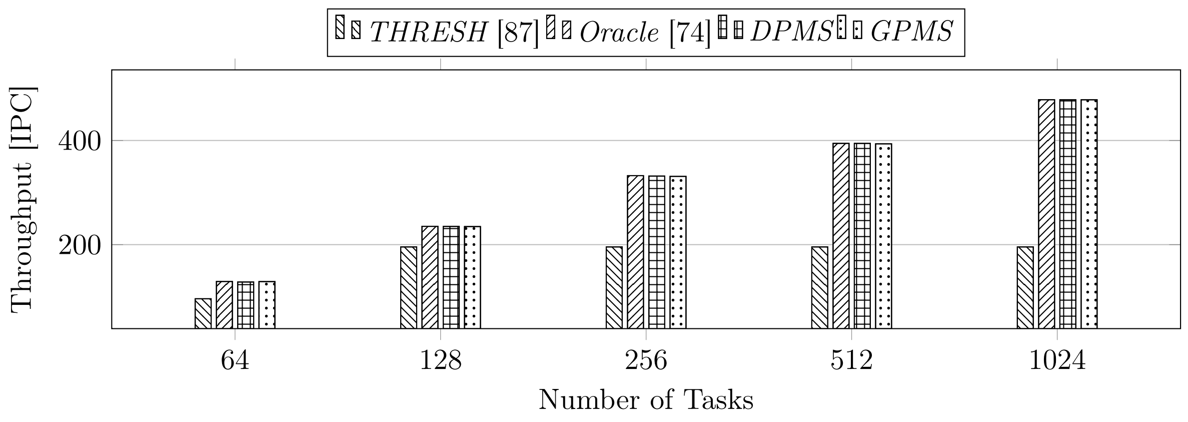


Figure 4.10.: Performance of schedulers on closed 256-core many-core under varisized workloads.

4.3.2. Comparative Baselines

Authors in [74] presented DP based centralized optimal scheduler for multi-cores called *Profile*, which takes in execution profiles of all tasks as input and operates on average core-speedups. We extend this scheduler to *Oracle*, which maximizes instantaneous IPC in every scheduling epoch to get our oracular comparison. Unlike *Profile* though, for a fair comparison, *Oracle* also employs IPC prediction from [76] instead of profiling; same as *DPMS* and *GPMS*.

$$\text{Maximize } \sum_{t_j} \zeta(C_{t_j}), \text{ given constraint } \sum_{t_j} |C_{t_j}| \;\le\; |C|$$

For perspective, we also compare against threshold-based heuristic called *Thresh* presented in [87]. *Thresh* allocates cores to task as long as task's gain in core-speedup from allocating an additional core is more than 40%.

4.3.3. Closed Many-Core

We begin with closed[5] 256-core many-core. Throughput is standard performance metric [32] for closed many-cores. A scheduler must result in highest throughput. Figure 4.10 shows throughput with different schedulers under varisized workloads. Experiment evaluates ten random workloads generated with uniform distribution among all available tasks and reports the average results. Figure 4.10 shows the performance of both *DPMS* and *GPMS* is equivalent to *Oracle*. *Thresh* being heuristic cannot adapt to workloads and hence lags behind.

4.3.4. Open Many-Core

The schedulers are not limited to closed many-cores but can be applied to open many-cores[6] as well. Response time – measured as the time difference between task's arrival and departure – is standard performance metric [32] for open many-cores. Experiment evaluates ten random workloads of 1024 tasks with uniform distribution amongst all available tasks and reports the average results. The arrival time of tasks follows a *Poisson* distribution. Figure 4.11 shows response time under different schedulers with varisized workloads. Results show *Oracle*, *DPMS*, and *GPMS* provide equivalent performance for all workloads.

[5]Refer Chapter 2.1 for definition of closed many-core.
[6]Refer Chapter 2.1 for definition of open many-core.

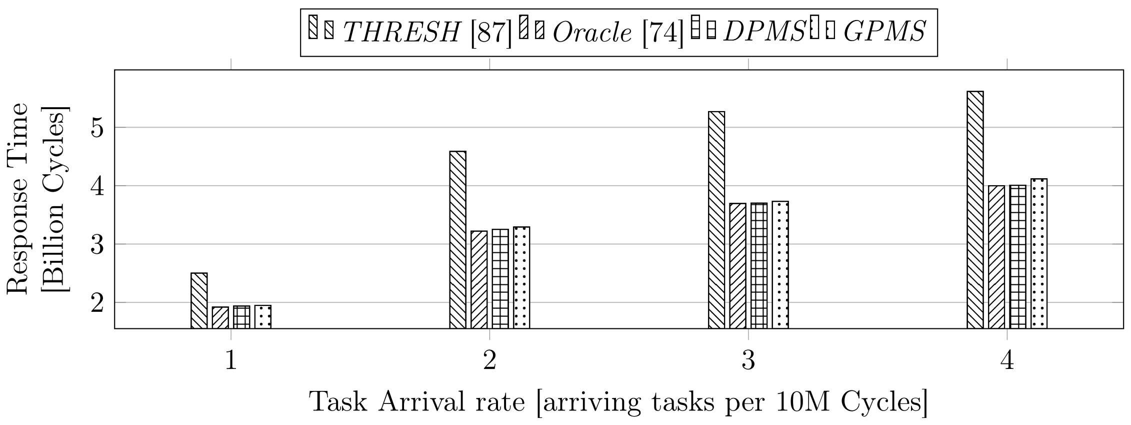


Figure 4.11.: Performance of schedulers on open 256-core many-core under varisized workloads.

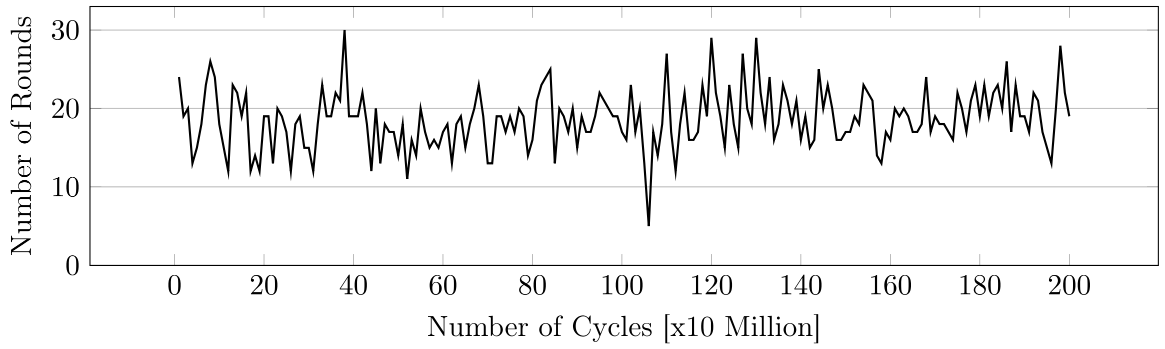


Figure 4.12.: Rounds until convergence under *DPMS* on half-loaded closed 256-core many-core.

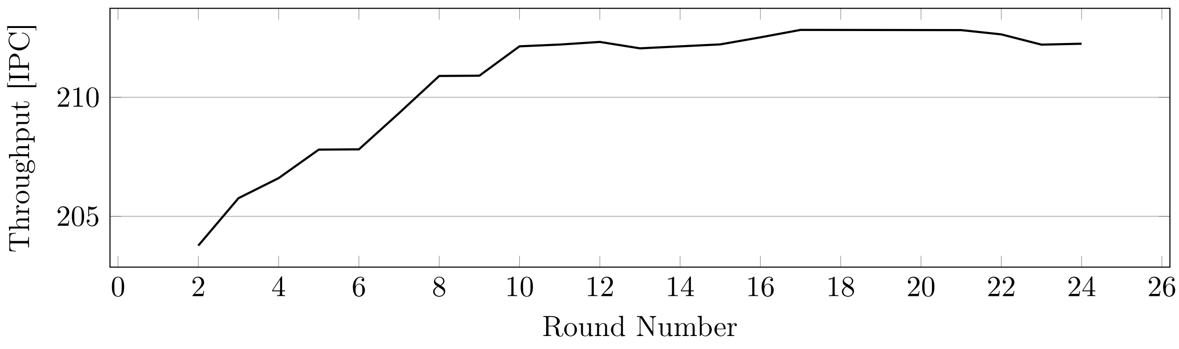


Figure 4.13.: Change in throughput for first scheduling epoch until convergence under *DPMS* on half-loaded closed 256-core many-core.

4.3.5. Convergence

DPMS organizes itself autonomously to produce better results with every successive round of core moves until it converges to equilibrium in $O(|T|)$ rounds (Theorem 1) from any given state. Figure 4.12 shows the number of rounds it takes for *DPMS* to attain equilibrium in every scheduling epoch for closed 256-core many-core with a 128-task workload. Convergence can take a large number of rounds in an initial state if all cores are initially unbounded as many-core would then require substantial reorganization to reach stability. We can avoid this initial penalty by starting many-core with a predefined distribution such as allocating equal cores to all tasks in closed many-core. Distribution would substantially hasten initial convergence

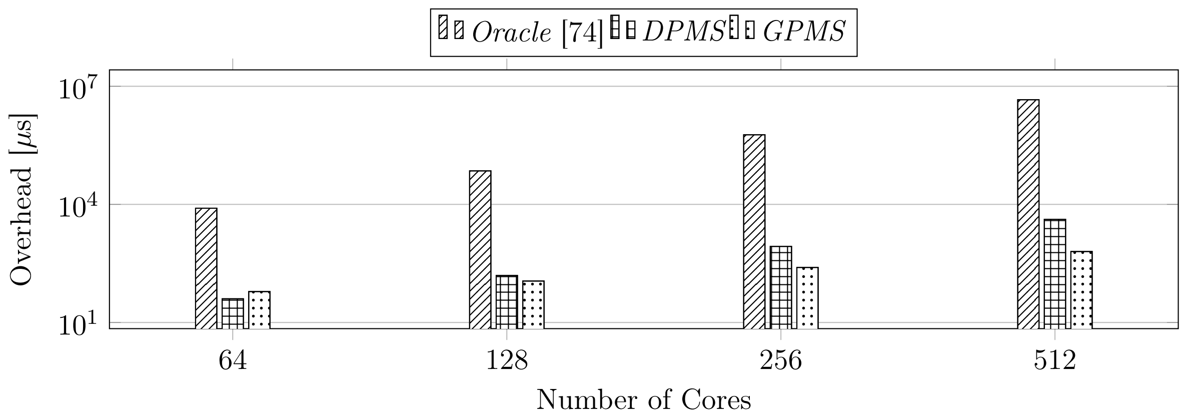


Figure 4.14.: Scheduling overhead for different schedulers on varisized many-cores under half load.

without affecting underlying performance. We observed in practice that the number of rounds needed for convergence is far less than predicted worst-case.

Figure 4.13 shows for the first scheduling epoch how throughput increases in general with every consecutive round till it reaches the maximum value at convergence. The throughput curve is not always smooth and increasing due to prediction errors and super-concavity smoothing.

4.3.6. Scalability

The main advantage of *DPMS* and *GPMS* over *Oracle* is their ability to perform scalable scheduling. In a nutshell, *DPMS* reduces per-core processing overhead by disbursing processing across all cores in many-core, but this also increases communication overhead compare to *Oracle*. *GPMS* reduces per-core processing overhead by employing a less computationally complex algorithm when compare to *Oracle*. Therefore, it is essential to get a measure of real-world benefits that *DPMS* and *GPMS* provide over *Oracle*.

Since running schedulers with real workloads cycle-accurately on *gem5* are time-wise infeasible for large size many-cores, we instead execute the logic of schedulers cycle-accurately with representative implementations and report observed problem-solving time. We implement *Oracle* and GPMS in C as single-threaded tasks. We implement DPMS as a multi-threaded C++ task. *DPMS* implementation uses the *pthread* library to implement one core as one thread. We subtracted thread spawning overhead from observations because it manifests from the construct of experiment and will not originate in real-world implementation.

Figure 4.14 shows worst-case overhead observed on varisized many-cores under *DPMS*, *GPMS*, and *Oracle* on a logarithmic scale. *DPMS* produces schedule much faster in practice than *Oracle*. *Oracle* requires 7.985 ms to produce schedule for 64-core many-core, whereas *DPMS* only requires 0.040 ms. Therefore, *DPMS* leads to 200x reduction in total overhead in comparison to *Oracle* on 64-core many-core. *DPMS* produces schedule on 64-core many-core even faster than *GPMS* which takes 0.061 ms to produce schedule. It is important to note that *DPMS* is not faster than *GPMS* on many-cores with more than 128 cores.

For larger size many-cores, even use of distributed algorithms may have unsustainable overhead, and perhaps the length of scheduling epoch itself would have to be lengthened beyond current standard 10 ms. It is also not possible for us to combine overhead results shown in Figure 4.14 with performance result shown in Figure 4.10 for more representative results because results are obtained using different types of schedulers.

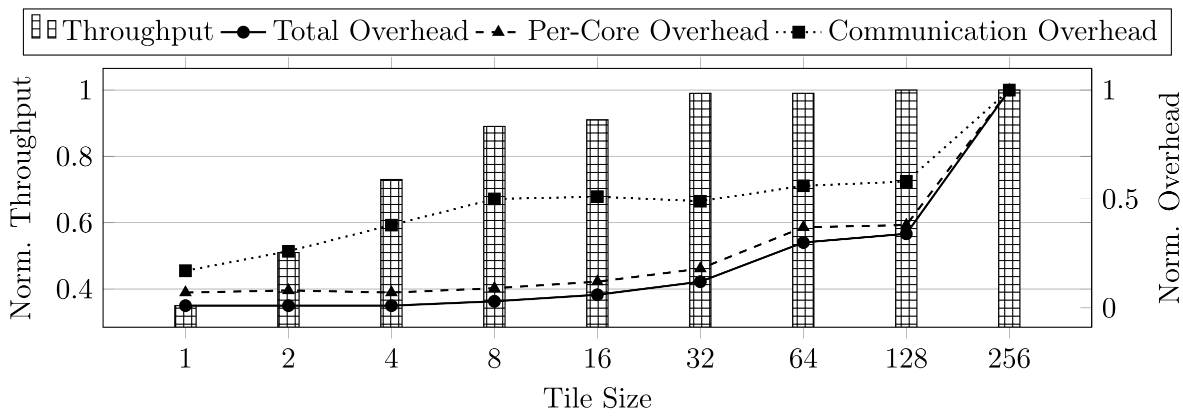


Figure 4.15.: Normalized throughput and overhead under *DPMS* on half-loaded closed 256-core many-core with varisized tiles.

4.3.7. Tiling

Due to performance penalty for maintaining cache-coherency across all spatially separated cores, a many-core design can cluster cores into shared-cache tiles. [21]. In such tiled many-cores, only cores within tile will be able to form virtual cores amongst each other. Figure 4.15 shows normalized throughput and overhead (against monolithic 256-core tile) under varisized tiles on closed 256-core many-core with 128-task workload. Figure 4.15 shows that keeping all cores in one tile (256-core tile) has no substantial performance advantage over many-core in which cores are divided equally into two tiles (128-core tile). However, tiling can substantially reduce scheduling overheads for *DPMS*. As long as tile size remains sufficiently larger than the average size of virtual cores formed without tiling restriction, effects on throughput are insignificant. Further reduction in tile size granularity can lead to significant drop in throughput.

4.4. Summary

In this chapter, we proposed a distributed scheduler called *DPMS* and a centralized greedy scheduler called *GPMS* for many-core task scheduling with the goal of maximizing performance. *DPMS* theoretically guarantees convergence to the optimal solution in given number of steps from any state. Since *DPMS* disburses its processing overhead across all cores in many-core, it can scale up with increase in the number of cores in the many-core. Similarly, *GPMS* is also shown to be very scalable and proven to be optimal.

The scheduling problem studied in this chapter can also be solved optimally using DP based centralized scheduler called *Oracle*. Our evaluations show that all schedulers reach equivalent solutions, but *DPMS* does so with several times reduction in per-core processing overhead compared to *Oracle*. Reduction in the per-core processing overhead under *DPMS* though comes at a nominal increase in communication overhead in comparison to *Oracle*. *GPMS* can provide an even more significant reduction in per-core processing overhead than *DPMS* and that also without any increase in communication overhead. Therefore, *DPMS* and *GPMS* are more suited for performing task scheduling in many-cores than *Oracle* as they provide superior scalability without making any compromise on the quality of schedule. Many-cores bring a new paradigm shift in processor design which requires a redesign of schedulers with even most basic goals such as performance maximization. *DPMS* and *GPMS* represent a step in that direction.

Performance comes at the cost of fairness in many-cores, which is not suitable for all overlying systems. We will study the problem of fairness maximization in many-cores in next chapter.

5. Many-Core Task Scheduler for Fairness Maximization

This chapter introduces a many-core task scheduler with the goal of fairness maximization.[1] Achieving high performance is often the goal of many-core schedulers as discussed in Chapter 4. However, in some systems wherein many-core is deployed fairness in core allocation among tasks is more emphasized than performance. For example, in embedded platforms wherein specific critical tasks should not experience substantial performance degradation to prevent system failure or in servers wherein tasks from different users agnostically running together should not experience any discrimination. Many-cores therefore sometimes need to ensure that all tasks receive their fair share of cores based on their requirements, and performance gain in one task does not happen at the expense of performance drop in another.

Linux *Completely Fair Scheduler* [88] is currently most widely used default fair scheduler in multi-cores. It allocates near-equal execution time slices to tasks so that each task gets a fair share of multi-core. Authors in [89] extend *Completely Fair Scheduler* to asymmetric multi-cores. In multi-cores, the number of tasks dominates the number of cores permitting applicability of concepts like time slicing. In many-cores on the other hand, the number of cores outnumbers the number of tasks thereby making the notion of round-robin execution redundant. Additionally, state-of-the-art fair schedulers for multi-cores proposed in research [90] are innately centralized and will not scale up as we transition from multi-cores to many-cores.

Scalable distributed schedulers for many-cores [91] are mostly performance-oriented and disregard fairness. Authors in [92] proposed a lightweight runtime centralized fair scheduling heuristic based on the notion of efficiency that is scalable but results in suboptimal fair schedules. We, on the other hand, propose scheduler in this chapter, which is not only scalable but is also proven to be optimally fair under certain conditions.

Fair scheduling problem becomes further challenging as processing requirements of tasks keep changing as they go through different phases of their execution [93]. This behavior results in task experiencing variable core-slowdown[2] during its execution. Core-slowdown is conceptually inverse to core-speedup used in Chapter 4 but is not its mathematical inverse. Also, as described in Chapter 4 we again rely upon adaptive many-cores in this work.

Figure 5.1 shows core-slowdown of two tasks – *bzip2* and *lbm* – with one core allocated at the granularity of every ten million instructions executed. To ensure optimal fairness at all times, fair scheduler needs to redistribute cores from tasks entering low-requirement phases to tasks entering high-requirement phases. If cores are scarce, then fair scheduler needs to ensure that all tasks experience similar resource crunch in the form of near-equal core-slowdown.

We choose variance in core-slowdowns of all tasks as our fairness metric [94]. Variance here quantifies dispersion in core-slowdowns being experienced by tasks. Variance is zero when allocations are entirely fair where all tasks are experiencing precisely same core-slowdown. Variance metric can inherently also detect task starvations. Starvation occurs when a scheduler denies task opportunity to execute by allocating it zero core. Core-slowdown of a task with no core allocated is infinite, making the variance in core-slowdowns infinite even if one task starves. Any scheduler with variance minimization goal must avoid task starvation. There are

[1]The work presented in this chapter was originally published in [2] ©2016 *EDAA*.

[2]Refer Chapter 3 for the definition of core-slowdown.

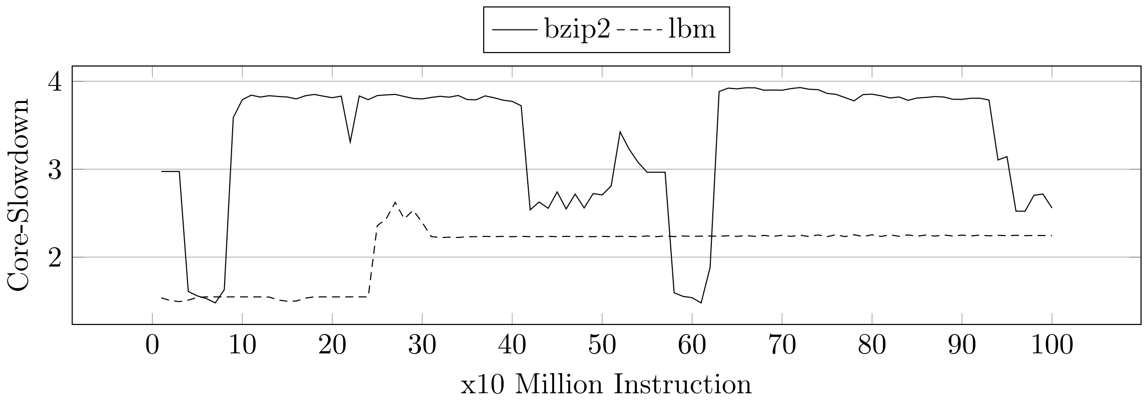


Figure 5.1.: Execution profiles of *bzip2* and *lbm* tasks showing changes in their core-slowdowns with 1-core allocation compared to 8-core allocation.

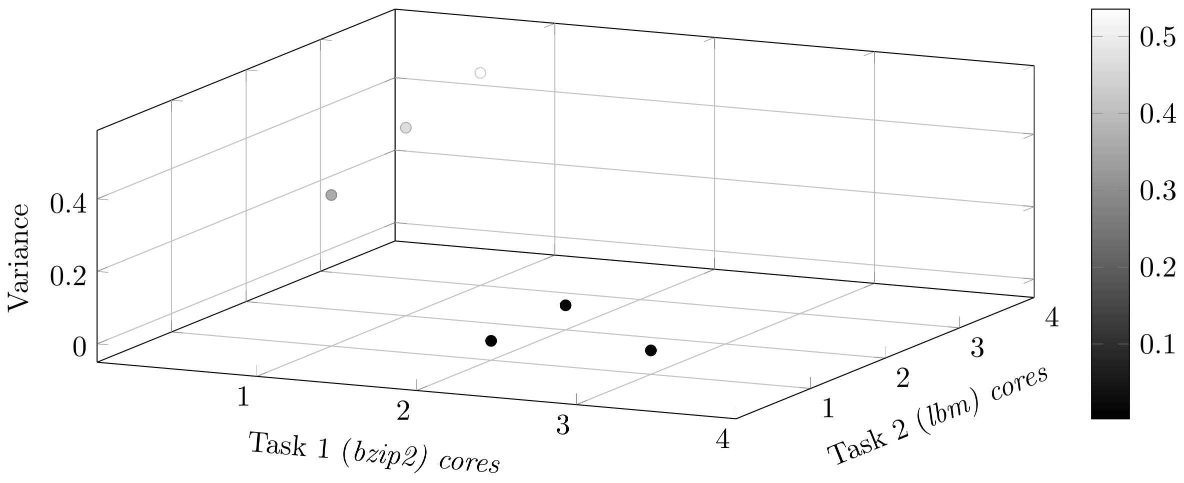
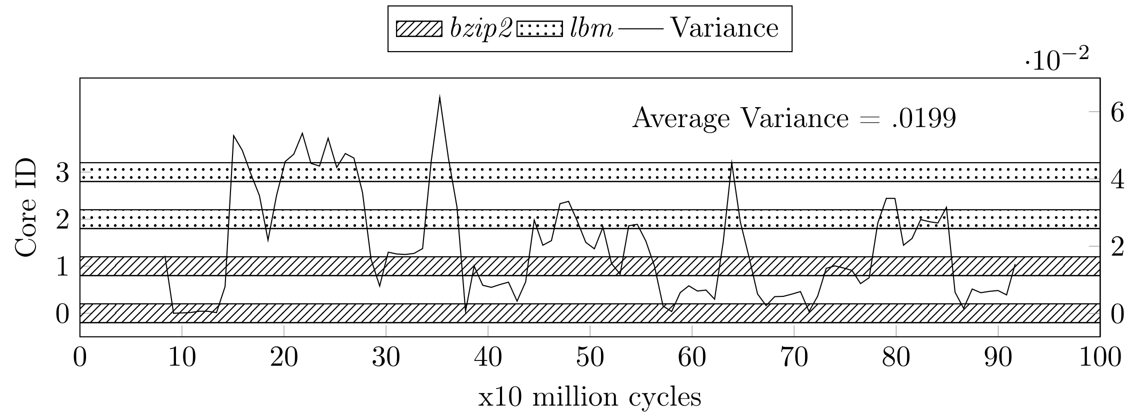


Figure 5.2.: Initial variance in six non-zero allocations of four cores between two tasks.

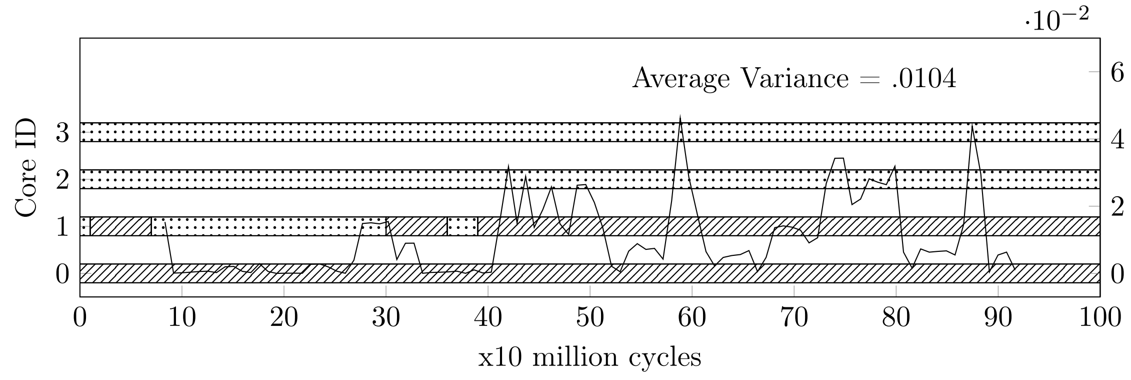
other fairness metrics such as ratio of maximum and minimum core-slowdown [95] or advanced concepts of game theoretic fairness [79]. Scheduler presented here is not limited to variance as fairness metric and can be used to target other fairness metrics if those metrics also exhibit specific properties required for convergence and optimality.

5.0.1. Pareto-Optimal Motivation

Fairness should not cause under-utilization in many-core. Figure 5.2 illustrates initial variance under six different starvation-free allocations of four cores between two tasks (*bizp2* and *lbm*). Amongst all allocations, the variance is minimum for allocation $\langle 2, 1\rangle$ at 0.002, but it does not allocate one of the available cores. This allocation is not Pareto-optimal, which in our context means that an allocation is possible in which core-slowdown of one task can be decreased without increase in core-slowdown of another task. Distributed fair scheduler needs to ensure that it does not converge to such points for sake of fairness. In contrast, allocations $\langle 1, 3\rangle$, $\langle 2, 2\rangle$, and $\langle 3, 1\rangle$ in which scheduler allocates all four cores are not equally fair because of imbalance in core-slowdowns of two tasks in those allocations. In our example amongst all Pareto-optimal allocations, allocation $\langle 3, 1\rangle$ has the lowest variance of 0.015. Note that it is not always possible to achieve complete fairness (i.e. zero variance) because task can be allocated



(a) Equal Static Scheduling



(b) Optimal Dynamic Fair Scheduling (Brute Force)

Figure 5.3.: Dynamic fair scheduling can improve fairness over a static equal distribution of cores.

to only discrete number of cores. We define Pareto-optimal allocations with minimum variance in core-slowdowns of tasks as optimally fair allocations in many-cores.

5.0.2. Dynamic Scheduling Motivation

Figure 5.3 shows the decrease in variance on a 4-core processor that can be provided by optimal (brute force) dynamic fair scheduling. Dynamic fair scheduling redistributes cores every scheduling epoch in comparison to static scheduling that gives equal cores to all tasks. Brute force solution is computationally too expensive to run in many-cores. We run two tasks – *bzip2* and *lbm* – on the 4-core processor. Figures 5.3a and 5.3b show the allocations over time with corresponding instantaneous variance under static and dynamic scheduling, respectively. Figure 5.3 shows that dynamic scheduling reduces average variance in core-slowdowns by 47.49% when compared to static scheduling. Reduction in variance is obtained due to back and forth transfer of core "1" amongst two tasks based on their relative instantaneous demands.

5.0.3. Novel Contributions

We present scheduler called *DFMS*. *DFMS* performs dynamic scheduling to ensure that tasks in many-core experience near-equal core-slowdowns throughout their execution. *DFMS* distributes fair scheduling processing overhead across all cores and thereby can scale up with the increase in the number of cores. The problem of variance minimization is well-studied [96] and is known to be NP-hard [97] in scheduling for a general case. Still, we show that problem's

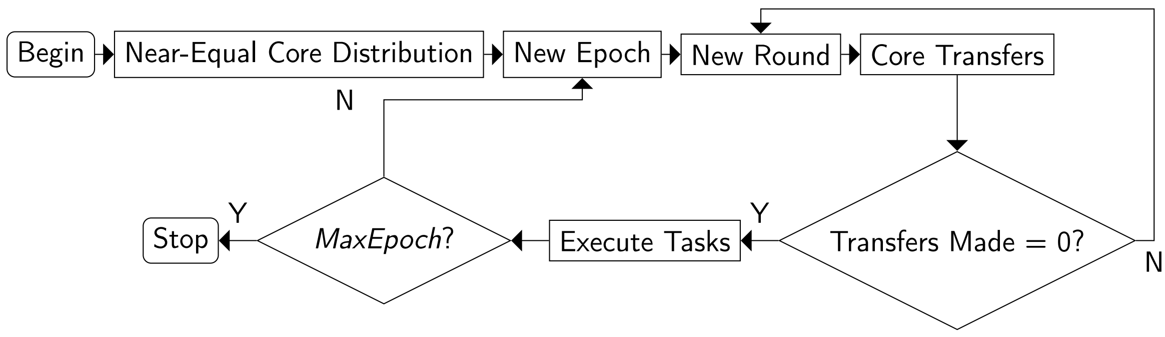


Figure 5.4.: Execution Flow for *DFMS*.

convex structures can be exploited to obtain optimal fair schedule in polynomial-time under some but not all conditions. The problem still remains for the general case.

5.1. Scheduler

Chapter 3 describes common notations used to describe *DFMS*. We assume all tasks are malleable [39] in this chapter similar to Chapter 4. $\bar{\gamma}(S)$ denotes average instantaneous core-slowdown of all tasks in many-core state S. $SoS(S)$ and $\sigma^2(S)$ represent the corresponding sum of squares of core-slowdowns and variance in core-slowdown in state S, respectively.

$$\bar{\gamma}(S) = \frac{1}{|T|}\sum_{t_i}\gamma(C_{t_i}) \tag{5.1}$$

$$SoS(S) = \sum_{t_i}\gamma(C_{t_i})^2 \tag{5.2}$$

$$\sigma^2(S) = \frac{1}{|T|}\sum_{t_i}(\gamma(C_{t_i}) - \bar{\gamma}(S))^2 \tag{5.3}$$

Tasks under *DFMS* transfer cores amongst each other based on a utility function. $u_{t_x\to t_y}(C_n)$ represents utility of transferring core set $C_n \subseteq C_{t_x}$ from task t_x to task t_y. Variance $\sigma^2(S)$ by definition is mean of the square of the distance between core-slowdowns and mean core-slowdown. When task t_x transfers core set C_n to task t_y, the value of core-slowdowns $\gamma(C_{t_x})$ and $\gamma(C_{t_y})$ change to $\gamma(C_{t_x}-C_n)$ and $\gamma(C_{t_y}\cup C_n)$, respectively. Their distances from mean core-slowdown $\bar{\gamma}(S)$ change. However, transfer of core set C_n also changes the value of mean core-slowdown itself from $\bar{\gamma}(S)$ to $\bar{\gamma}(S')$, where S' is modified state after transfer. This change results in the change in distance from mean core-slowdown for all tasks. The scheduler temporarily ignores the effect of transfer on other tasks by assuming $\bar{\gamma}(S) \approx \bar{\gamma}(S')$ to make the problem more tractable. Therefore, variance from transfer will decrease if the combined change in the square of the distance of new core-slowdowns from mean core-slowdown is less than before. This decrease inspires our definition of utility transfer $u_{t_x\to t_y}(C_n)$.

$$u_{t_x\to t_y}(C_n) = \gamma(C_{t_x} - C_n)^2 + \gamma(C_{t_y} \cup C_n)^2 - \gamma(C_{t_x})^2 - \gamma(C_{t_y})^2 \tag{5.4}$$

5.1.1. Execution Flow

Figure 5.4 depicts execution flow of *DFMS* graphically. Initially, all cores are distributed near-equally amongst all tasks. Every scheduling epoch, series of core transfer rounds take place

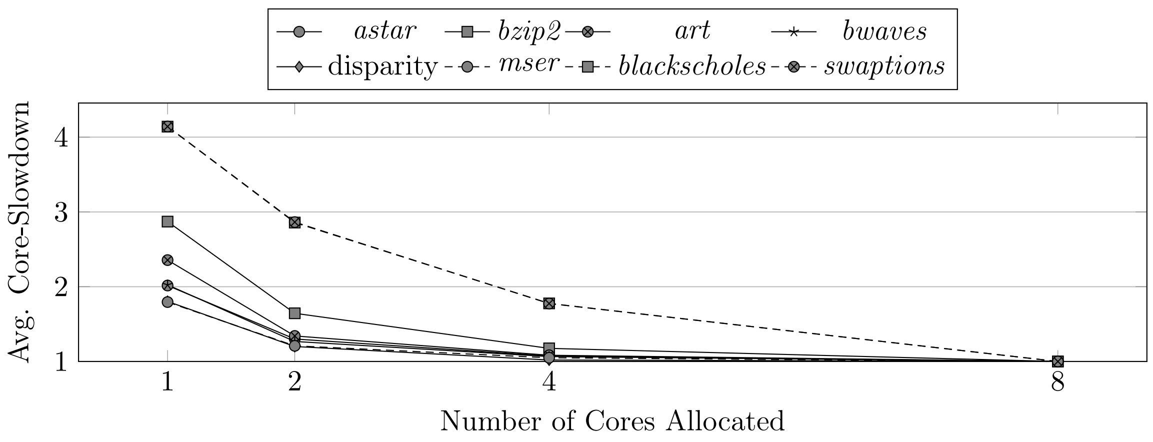


Figure 5.5.: Average core-slowdown in different tasks when allocated different numbers of cores.

to make allocations fairer. Rounds stop when allocations cannot be made any fairer. In every round, all tasks communicate and exchange their instantaneous core-slowdowns. Tasks then calculate the utility of transferring cores they hold to every other task. Transfer under *DFMS* entails a sacrifice in performance by a task for the welfare of other tasks. During a round, every task randomly picks another task with which negative utility transfers are possible and exchanges cores with it. In the round, we restrict the task to interact with only one other task to prevent oscillating transfers. The task is free to interact with any task including ones it had previously interacted with in future rounds within same scheduling epoch. Rounds stop for scheduling epoch when no more negative utility transfers are possible. Tasks then execute with converged allocations. Rounds commence again in next scheduling epoch till scheduling ends at user-defined scheduling epoch called "*MaxEpoch*". We also need to ensure that there should be increase in core-speedup of task to which cores are being transferred to ensure Pareto-optimality. Instantaneous core-slowdowns of tasks can be predicted at run-time using IPC prediction models for adaptive many-cores presented in [76] to avoid any profiling.

5.1.2. Execution Characteristics

Figure 5.5 shows average core-slowdowns in different tasks when allocated a different non-zero number of cores. Core-slowdown in the task decreases monotonically with the increase in number of allocated cores because each allocated core brings with it non-negative increase in task's IPC moving it closer to its maximum achievable IPC. Therefore, there is no harm from keeping cores always allocated to some task.

Core-slowdown in task is also convex with the increase in the number of allocated cores because the increase in IPC brought by each subsequently allocated core is less than previous one until it saturates. This behavior is because of saturation of exploitable ILP and TLP in tasks required to accelerate them. To best of our knowledge, we are the first one to exploit these convex properties in the context of the fair scheduling problem in many-cores.

5.1.3. Optimal Fairness

We now present proofs for guarantees of optimality and convergence provided by *DFMS*.

Lemma 5. DFMS *converges to Pareto-optimal allocation that minimizes sum of squares of core-slowdowns SoS in $O(|T|)$ rounds.*

Proof. After transfer $u_{t_x \to t_y}(C_n)$, sum of squares of core-slowdowns $SoS(S)$ changes to $SoS(S')$.

$$\begin{aligned} SoS(S') &= \sum_{t_i} \gamma(C_{t_i})^2 + \gamma(C_{t_x} - C_n)^2 + \gamma(C_{t_y} \cup C_n)^2 - \gamma(C_{t_x})^2 - \gamma(C_{t_y})^2 \\ &= SoS(S) + u_{t_x \to t_y}(C_n) \ \ [\because \text{Eq. (5.2) and Eq. (5.4)}] \end{aligned}$$

So, the sum of squares of core-slowdowns $SoS(S')$ is greater than $SoS(S)$ when the transfer $u_{t_x \to t_y}$ utility is greater than zero. The sum of squares of core-slowdowns $SoS(S')$ is less than $SoS(S)$ when the transfer $u_{t_x \to t_y}$ utility is less than zero. Hence, only negative utility transfers can reduce the sum of squares of core-slowdowns SoS from any given state. Sum of squares of core-slowdowns SoS cannot be reduced further when no negative utility transfer exists, and scheduler reaches a minimum. We now prove that this minimum reached theoretically minimizes the sum of squares of core-slowdowns SoS.

Instantaneous core-slowdowns of tasks in practice are piecewise linear functions that are computationally expensive to optimize [98]. Based on observations made in Figure 5.5, we assume that core-slowdown $\gamma(C_{t_i})$ of task t_i is convex-extensible to a non-negative discrete convex function of allocation C_{t_i}. Error introduced by this convex relaxation [99] is minimal. Since a square of non-negative convex function remains convex, square of core-slowdown $\gamma(C_{t_i})^2$ is a convex function of allocation C_{t_i}. $SoS(S)$ by definition is then a positive sum of convex functions, therefore is also a discrete convex function of allocations in state S. All minima of discrete convex function are its global minima [100].

Since allocations are discrete, there exist an only finite number of negative utility transfers. Two tasks t_x and t_y that exchange core set C_n once will not exchange cores again if the transfer of core set C_n minimizes associated utility unless disturbed by a third task. Transfer $u_{t_x \to t_y}(C_n)$ utility is also a discrete convex function of the size of core set C_n that can be minimized efficiently using simple gradient descent algorithm. We force tasks under *DFMS* to always make transfers that minimize associated utility. Thus, transfers with negative utility will exhaust in at worst $O(|T|)$ rounds, since a task can interact with at most $|T|$ other tasks (excluding repeated interactions). *DFMS* allows transfers only between tasks, and therefore only Pareto-optimal allocations are explored; hence proved. □

Theorem 4. DFMS *converges to optimally fair allocation under given performance constraint.*

Proof. Beginning with Equation (5.3),

$$\begin{aligned} \sigma^2(C) &= \frac{1}{|T|} \sum_{t_i} (\gamma(C_{t_i}) - \bar{\gamma}(S))^2 \\ &= \frac{1}{|T|} \sum_{t_i} \left(\gamma(C_{t_i})^2 + \bar{\gamma}(S)^2 - 2\gamma(C_{t_i})\bar{\gamma}(S)\right) \\ &= \frac{1}{|T|} \left(\sum_{t_i} \gamma(C_{t_i})^2 + \sum_{t_i} \bar{\gamma}(S)^2 - 2 \sum_{t_i} \gamma(C_{t_i})\bar{\gamma}(S) \right) \\ &= \frac{1}{|T|} \left(\sum_{t_i} \gamma(C_{t_i})^2 + |T|\bar{\gamma}(S)^2 - 2|T|\bar{\gamma}(S)^2 \right) \ \ [\because \text{Eq. (5.1)}] \\ &= \frac{1}{|T|} \left(\sum_{t_i} \gamma(C_{t_i})^2 - |T|\bar{\gamma}(S)^2 \right) \\ &= \frac{1}{|T|} \left(\sum_{t_i} \gamma(C_{t_i})^2 \right) - \bar{\gamma}(S)^2 \\ &= \frac{1}{|T|} SoS(S) - \bar{\gamma}(S)^2 \ \ [\because \text{Eq. (5.2)}] \end{aligned} \tag{5.5}$$

We measure many-core performance as the sum of core-slowdowns. This performance metric is different from throughput used as the performance metric in Chapter 4, but both metrics in practice are positively correlated. We operate *DFMS* with the extra condition that performance $\sum_{t_i} \gamma(C_{t_i})$ should not change while performing transfers. Based on Equation (5.1), this makes mean core-slowdown $\bar{\gamma}(S)$ constant independent of state S since the number of tasks $|T|$ is also constant at any given time. Therefore, based on Equation (5.5) variance $\sigma^2(S)$ is minimized when the sum of squares of core-slowdowns $SoS(S)$ is minimal; hence proved using Lemma 5. □

Sum of core-slowdowns can be maximized optimally using *DPMS* or *GPMS* introduced in Chapter 4. Since core-slowdowns saturate after a certain number of allocated cores, they are convex but not strictly convex. Therefore, there exist multiple allocations with the maximum performance but not all of them are equally fair regarding variance in core-slowdowns. Theorem 4 as special case allows the search of allocations with optimal fairness under optimal performance in polynomial time. To best of our knowledge, we are first to present this result.

5.1.4. Heuristic Fairness

Mean core-slowdown $\bar{\gamma}(S)$ remains a function of state S without fixed performance constraint. Negative of the square of mean core-slowdown $-\bar{\gamma}^2(S)$ is then a concave function of state S. Based on Equation (5.5), variance $\sigma^2(S)$ is a sum of concave function and convex function, which is neither concave nor convex. This makes variance hard to minimize optimally even when all tasks have perfectly convex core-slowdowns. For the general case, we change *DFMS* into a heuristic distributed local search.

Transfer of core set C_n from task t_x to task t_y changes value of variance from $\sigma^2(S)$ to $\sigma^2(S')$. Let Δ_γ and Δ_{SoS} represent the difference between new core-slowdowns and old core-slowdowns, and the difference between the square of new core-slowdowns and square of old core-slowdowns after transfer of core set C_n, respectively.

$$\begin{aligned}
\Delta_\gamma &= \gamma(C_{t_i} - C_n) + \gamma(C_{t_j} \cup C_n) - \gamma(C_{t_i}) - \gamma(C_{t_j}) \\
\Delta_{SoS} &= \gamma(C_{t_i} - C_n)^2 + \gamma(C_{t_j} + C_n)^2 - \gamma(C_{t_i})^2 - \gamma(C_{t_j})^2 \\
\sigma^2(C') &= \frac{1}{|T|}(SoS(S) + \Delta_{SoS}) - (\bar{\gamma}(S) + \frac{\Delta_\gamma}{|T|})^2 \quad \text{[Similar to Eq. (5.5)]} \\
&= \frac{SoS(S)}{|T|} + \frac{\Delta_{SoS}}{|T|} - \bar{\gamma}(S)^2 - \frac{\Delta_\gamma^2}{|T|^2} - \frac{2\bar{\gamma}(S)\Delta_\gamma}{|T|} \\
&= \sigma^2(C) + \frac{\Delta_{SoS}}{|T|} - \frac{\Delta_\gamma^2}{|T|^2} - \frac{2\bar{\gamma}(S)\Delta_\gamma}{|T|} \quad [\because \text{Eq. (5.5)}]
\end{aligned}$$

Now variance $\sigma^2(S') < \sigma^2(S)$ if,

$$\begin{aligned}
0 &> \frac{\Delta_{SoS}}{|T|} - \frac{\Delta_\gamma^2}{|T|^2} - \frac{2\bar{\gamma}(S)\Delta_\gamma}{|T|} \\
&> \Delta_{SoS} - \frac{\Delta_\gamma^2}{|T|} - 2\bar{\gamma}(S)\Delta_\gamma \\
&> \Delta_{SoS} - 2\bar{\gamma}(S)\Delta_\gamma \quad [\because |T| >> \Delta_\gamma^2 \text{ in many-cores}] \qquad (5.6)
\end{aligned}$$

We redefine utility as $u_{t_x \to t_y}(C_n) = \Delta_{SoS} - 2\bar{\gamma}(S)\Delta_\gamma$ for minimizing variance under no performance constraint. Note that this version of *DFMS* is not optimal because it does not guarantee that local minima reached are also global minima. Question of finding a polynomial-time fairness maximizing scheduler in many-cores independent of performance remains open.

5.1.5. Complexity

Under *DFMS*, every task does $O(|T|)$ utility calculations in every round. Each utility calculation $u_{t_x \to t_y}(C_n)$ is required to find the size of core set C_n to be transferred, requiring $O(|C|)$ calculations. Furthermore, there can be at worst $O(|T|)$ rounds. Hence, worst-case complexity of total calculations is $O(|T|^2|C|)$ in a scheduling epoch.

DFMS randomly distributes calculations over $|C|$ cores, and therefore per-core worst-case calculations are $O(|C|^2)$. *DFMS* has a total communication complexity of $O(|T|^2)$ per scheduling epoch as it needs to broadcast $O(|T|)$ messages every round. *DFMS* has $O(1)$ space complexity since it requires no data structures.

5.2. Experimental Evaluations

5.2.1. Experimental Setup

Experimental setup used in this chapter to evaluate *DFMS* is same as setup used in Chapter 4.

5.2.2. Comparative Baselines

We compare *DFMS* against two other schedulers to prove its efficacy. $EQUAL$ is a static fair scheduler that distributes cores near-equally amongst all tasks. We choose to compare against this simple approach to show that equal distribution of cores amongst tasks does not result in fair scheduling even though it is intuitive and scalable.

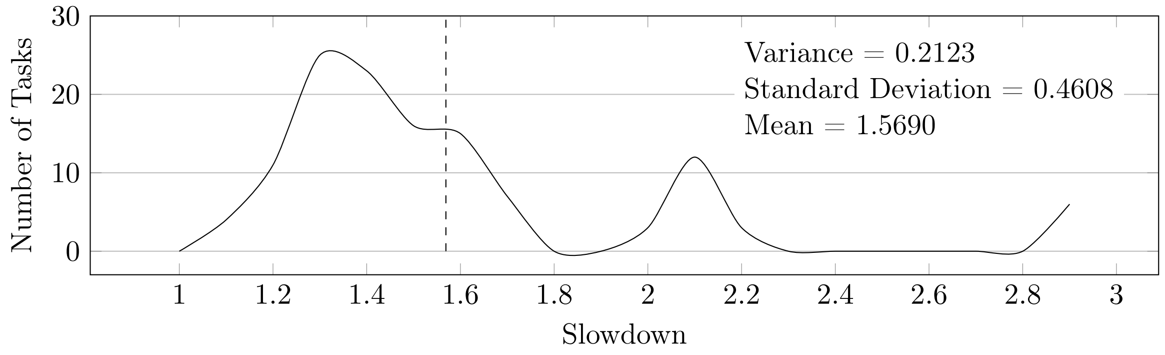
We also choose to compare against a heuristic-based dynamic fair scheduler for many-cores called *PDPA* [92], which works on the notion of "*Execution Efficiency*" defined as core-speedup per unit core. Note that *Execution Efficiency* is neither convex nor concave metric on many-cores even though core-speedup can be assumed to be concave function as shown in Figure 4.6. *PDPA* has two empirically determined thresholds *target_ eff* and *high_ eff*. Tasks are allocated cores in every scheduling epoch so that their execution efficiencies are between *target_ eff* and *high_ eff*. We choose to compare against this approach (with default threshold values) to show that threshold-based heuristics though scalable, can be outperformed. *PDPA* is modified to enforce Pareto-optimality for a fair comparison with *DFMS*.

5.2.3. Optimal Fairness for Fixed Performance

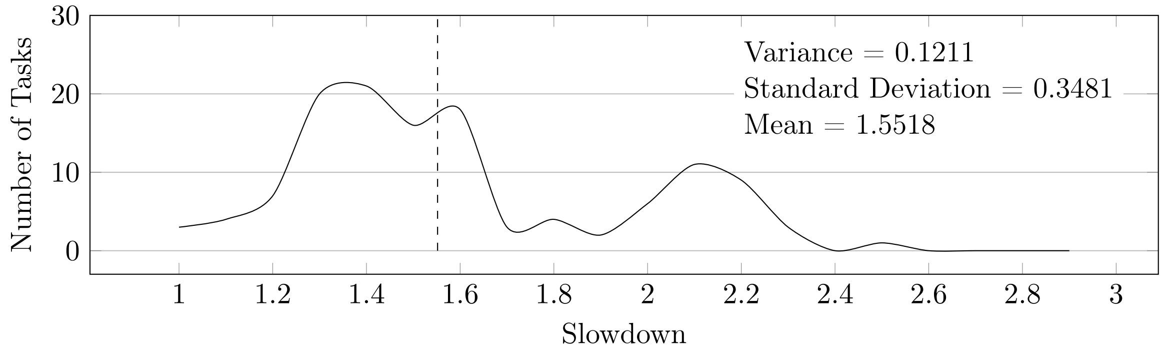
Theoretically, there exist an exponential number of allocations of equal performance with different fairness. In practice, such allocations only exist when there exists a core transfer which increase in core-slowdown of one task is precisely equal to decrease in core-slowdown in another task. This case would be at best rare in the real-world with full precision.

We define symbol $\Delta_{\bar{\gamma}} = \Delta_{\gamma}/\bar{\gamma}(S)$ as threshold representing a change in the sum of core-slowdowns over mean core-slowdown. Equation (5.6) can be rewritten as $0 > \Delta_{SoS} - 2\bar{\gamma}^2(C)\Delta_{\bar{\gamma}}$ to include threshold $\Delta_{\bar{\gamma}}$. As long as threshold $\Delta_{\bar{\gamma}}$ is kept as close to zero as possible, the variance can be minimized near-optimally using *DFMS*.

We run *DFMS* under threshold $|\Delta_{\bar{\gamma}}| \leq .01$, closest to zero we can get in our experiments for any reconfiguration to happen. Bounds on threshold $\Delta_{\bar{\gamma}}$ can be relaxed further but only with the loss of optimality. On closed 256-core many-core with 128-task workload, this results in 42.95% reduction in variance with only 1.09% change in mean slowdown from initial equal core distribution. Figure 5.6 shows how the physical distribution of core-slowdowns of tasks



(a) Before *DFMS*



(b) After *DFMS*

Figure 5.6.: Distribution of core-slowdowns of tasks around mean core-slowdown before and after *DFMS* is applied on closed 256-core many-core with 128 tasks workload with initial equal core distribution under a performance constraint.

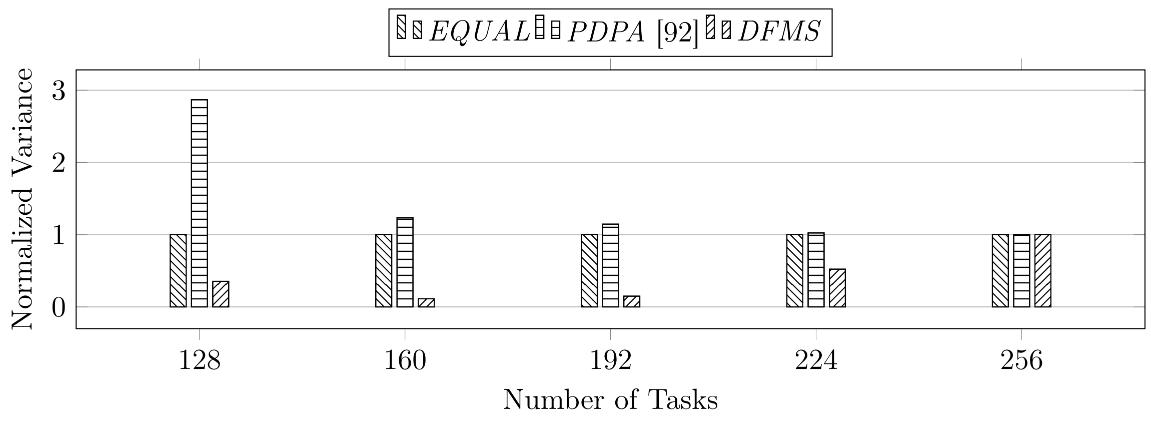
shifts around mean core-slowdown in many-core before and after *DFMS* is applied. *DFMS* substantially reduces dispersion in core-slowdowns resulting in more fairness.

5.2.4. Improved Fairness for Variable Performance

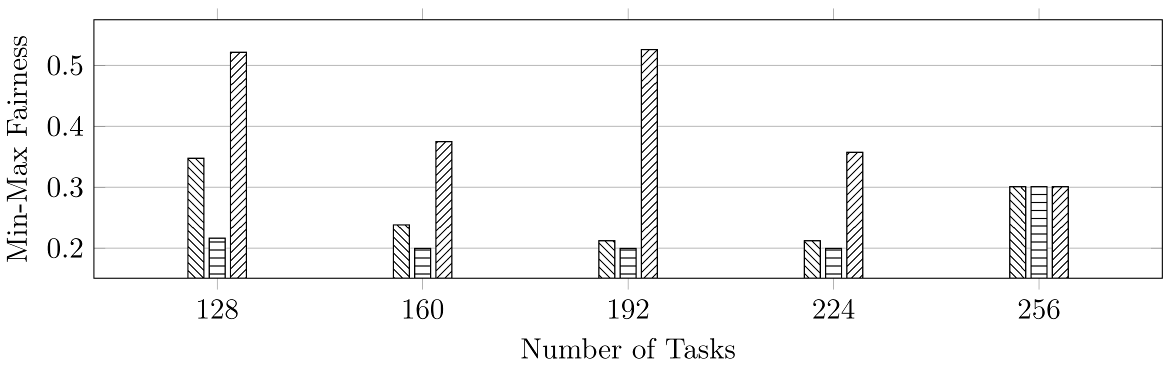
DFMS operates heuristically using Equation (5.6) as the utility when performance is unconstrained. Figure 5.7a shows average variance on closed 256-core many-core with varisized workloads under different schedulers (normalized to *EQUAL* scheduler). With a full load (256 tasks), an entirely fair allocation is to allocate one core to each task as under any other allocation some task will inevitably starve pushing many-core variance to infinite. All schedulers can discover the same optimal solution.

The number of surplus cores increases with the decrease in the number of tasks. This surplus results in expansion of optimization search-space, making it more challenging to maintain fair allocations. Figure 5.7a shows that *DFMS* results in better fair schedules in comparison to *EQUAL* and *PDPA* under all loads. *EQUAL* performs worse since tasks have inherently different execution patterns resulting in different core-slowdowns for the same number of allocated cores as shown in Figure 5.5. *PDPA* does not work because there is no unique set of thresholds that can result in the optimal fair schedule for all possible workloads.

Another common metric used to measure fairness is the ratio of minimum and maximum core-slowdown amongst all tasks. Value of “1” for the ratio of minimum and maximum core-slowdown indicates maximum fairness whereas the value of “0” indicates minimum fairness. We observed that this metric has a high correlation to variance metric that *DFMS* optimizes.



(a) Variance in Core-Slowdown based Fairness (Lower Value is Better)



(b) Min-Max Core-Slowdown Ratio based Fairness (Higher Value is Better)

Figure 5.7.: Fairness under different schedulers on closed 256-core many-core with varisized workloads.

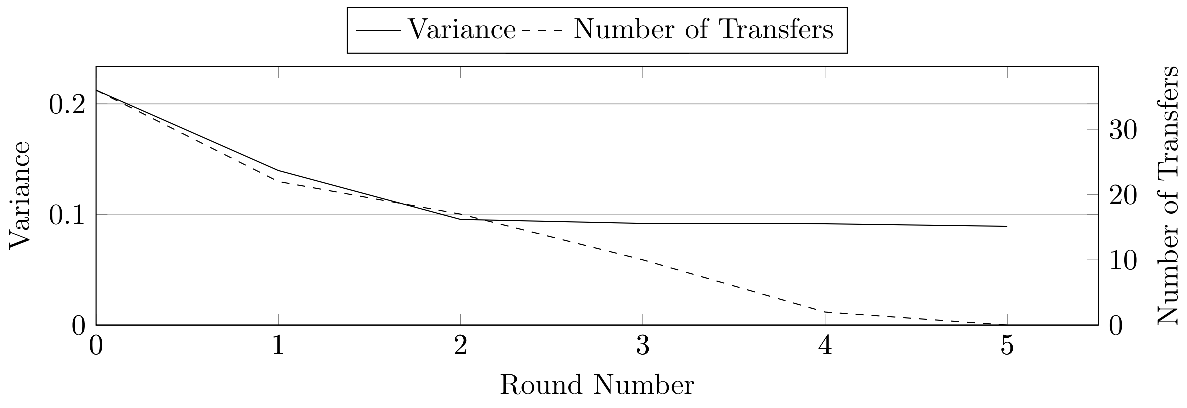


Figure 5.8.: Convergence in series of rounds to optimal fair allocation from equal core distribution under *DFMS* on closed 256-core many-core with 128 tasks workload.

Figure 5.7b shows average min-max fairness for different schedulers. Evaluations show that DFMS also performs better than baselines when using min-max fairness metric. We believe results would hold for several other fairness metrics not tested here.

5.2.5. Convergence

Figure 5.8 shows how from a state of equal distribution of cores to tasks on closed 256-core many-core with 128 tasks workload, transfers with negative utility in series of rounds make

Table 5.1.: Different scheduling overheads under *DFMS* on varisized closed many-cores while operating at half load. Overheads of the smallest many-core act as the normalizer.

Cores	Tasks	Total Processing Overhead	Per-Core Processing Overhead	Communication Overhead
128	64	1x	1x	1x
256	128	5.64x	2.74x	1.25x
512	1024	25.90x	6.16x	3.01x
1024	2048	103.82x	12.34x	9.06x
2048	1024	415.69x	24.41x	16.60x

many-core unilaterally converge to fairer allocation in a scheduling epoch. Initially, the scope of variance minimization is significant, and hence a large number of transfers are made. In subsequent rounds, the number of transfers decreases as *DFMS* approaches optimization minima.

5.2.6. Scalability

Accurate overhead numbers can only be shown in real hardware and also depends upon the quality of implementation. Still, we present some proof-of-concept results. In our simulations, we assume the average number of message broadcasted per scheduling epoch as a rough measure of communication overhead. We use the average number of floating point operations per scheduling epoch for determining schedules as a rough measure of processing overhead. Floating point operations performed across all cores is used to measure total processing overhead. The maximum number of floating point operations performed by a core amongst all cores is used to measure per-core processing overhead.

Table 5.1 shows how different overheads increase as we go from 128-core many-core to 2048-core many-core, all with half loads. Observations made are in sync with the theoretical complexity of *DFMS* stated in Section 5.1.5.

5.3. Summary

In this chapter, we proposed fair scheduler for many-cores called *DFMS*. *DFMS* is a lightweight distributed dynamic scheduler that can be applied at runtime for constant partial reallocation of cores to maintain fair allocation at all times even under continuously changing processing requirements of tasks. *DFMS* is proven theoretically to converge to the optimal fair allocation for given performance. When performance is not given, *DFMS* is forced to operate as a sub-optimal local-search heuristic. *DFMS* distributes its processing overhead across all cores. As a result, it can scale up as the number of cores in the many-cores increase.

The scheduler presented in this chapter, as well as schedulers presented in Chapter 4, focus on determining the number of cores allocated to tasks to maximize fairness and performance of many-core, respectively. Schedulers proposed till now assume all cores to be of equal importance to tasks and ignore actual spatial location of these cores on many-core.

In practice, all cores are not equivalent to a task, and some cores such as cores close to set of cores already allocated to the task are more valuable to the task than cores located far away. In next chapter, we discuss how without even modifying the number of cores allocated to tasks their execution can be accelerated just by rearranging allocations using thread migrations via the process of defragmentation. We also move to a more real-world representative experimental setup in the next chapter. The setup used till now is unable to model sensitivity of a task towards spatiality of its allocated cores and hence is inadequate.

6. Many-Core Task Defragmenter

This chapter introduces a scheduler with the goal of task defragmentation (defragmenter) in many-cores.[1] Diverging from Chapter 4 and 5, we now use tile-based non-adaptive homogeneous many-core in this chapter. Cores in this many-core are spatially placed in two-dimensional lattice connected by mesh NoC as shown in Figure 6.1. The cores have private L1 instruction and data caches, and private L2 cache. Core along with its caches form a tile. Each tile connects through a router to four adjacent tiles - one in each direction. Tiles are kept coherent using cache-directories distributed alongside LLC. Multiple MCs attached to perimeter cores provide access to off-chip Dynamic Random-Access Memory (DRAM). Many-core has Dynamic Non-Uniform Cache Access (D-NUCA) caches.

Intel's *SCC* [101] uses similar many-core architecture but with message-passing instead of cache-coherency and has two cores per-tile. Though this chapter does not explore message-passing based many-core, spatially sensitive multi-threaded execution model at OS level conceptually remains same for it as cache-coherent many-core with D-NUCA caches. Therefore, this chapter applies to both types of many-core architectures.

A multi-threaded task on many-core, in general, executes faster when the set of cores allocated to it are contiguous (spatially connected by isolated NoC link) and compact (shape formed by allocation has minimum perimeter). Under such allocation, communication overhead between task's threads spread over the allocation is minimal. Figure 6.2 shows performance loss experienced by different multi-thread tasks on 64-core many-core when we pin their four threads to four corner cores of many-core against when we pin the threads together at the center of many-core. Performance loss observed in tasks strongly correlate with their characterized inter-thread communications [85].

The relative benefits from thread co-location would, in general, increase with the increase in the number of spawned threads for a task because of increase in inter-thread synchronizations. Furthermore, under contiguous allocation inter-thread NoC-traffic generated by one task remains isolated and does not interfere with another task's NoC-traffic. This isolation reduces NoC-congestion, enhancing many-core's multi-program performance.

Task defragmentation is vital in open many-cores. Depending upon instruction lengths (input sizes) and compositions, the number of cores allocated (threads spawned), and inter-thread communication execution time of different tasks can differ widely. Therefore, we neither know arrival nor departure time of a task in open many-core. Over a span of time, this results in unallocated cores getting scattered all over many-core generating gaps (fragments) in allocations. Formation of these gaps leads to the problem of fragmentation in many-cores.

Fragmentation makes it difficult to perform efficient contiguous compact allocation of new incoming tasks. Defragmenter can reduce fragmentation by consolidating smaller gaps into larger gaps. Defragmentation would lead to a more responsive open many-core. A centralized defragmenter is sufficient for multi-cores. However, for many-cores given the massive optimization search-space it would not scale up. Therefore, a distributed defragmenter which distributes its overheads across all cores and allows multiple gaps to merge in parallel is required.

[1]The work presented in this chapter was originally published in [6] ©2017 *ACM*.

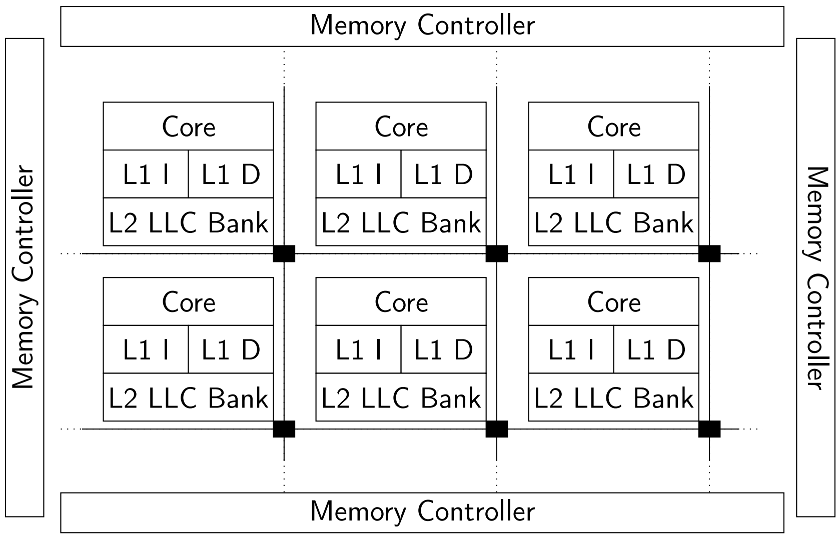


Figure 6.1.: An abstract block diagram of a tiled many-core.

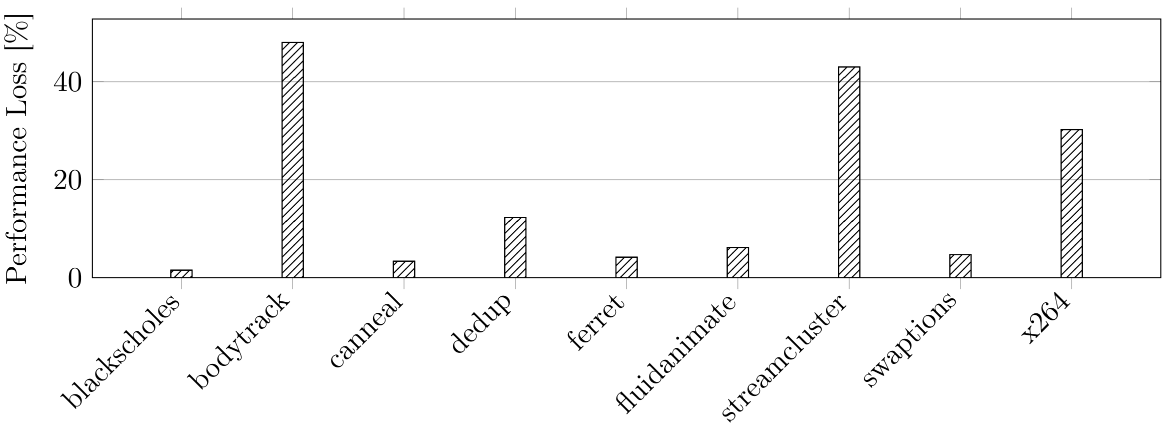


Figure 6.2.: Performance loss in different multi-threaded tasks due to non-compact allocation.

6.0.1. Defragmentation Motivation

Figure 6.3 shows with simple illustration how fragmentation leads to inefficiency on 64-core many-core. Initially, many-core is executing a total of six tasks as shown in Figure 6.3a. Tasks t_1 (*blackscholes*) and t_6 (*x264*) finish, and leave the many-core changing state to Figure 6.3b. Task t_7 (*bodytrack*) then arrives with a requirement of sixteen cores. Figure 6.3c shows state and corresponding execution time of task t_7 if we allocate it without defragmentation. Figure 6.3d shows the state in an alternate timeline if defragmentation is performed first by migrating task t_5 (*streamcluster*) in the middle of its execution before allocation of task t_7 . Figure 6.3e shows state and corresponding execution time of task t_7 with allocation after defragmentation. Experiments show that execution time of task t_7 is reduced by 30 ms (10.16%) in the state depicted by Figure 6.3e in comparison to the state depicted in Figure 6.3c because of optimized inter-thread NoC communication.

In contrast, the performance penalty of migration on task t_5 for defragmentation is comparatively less at 14 ms (3.92%). The task t_5 experiences elongated execution because defragmentation forces its threads to experience cache-misses on the newly allocated core. Nevertheless, we observe that net gain in overall performance is positive.

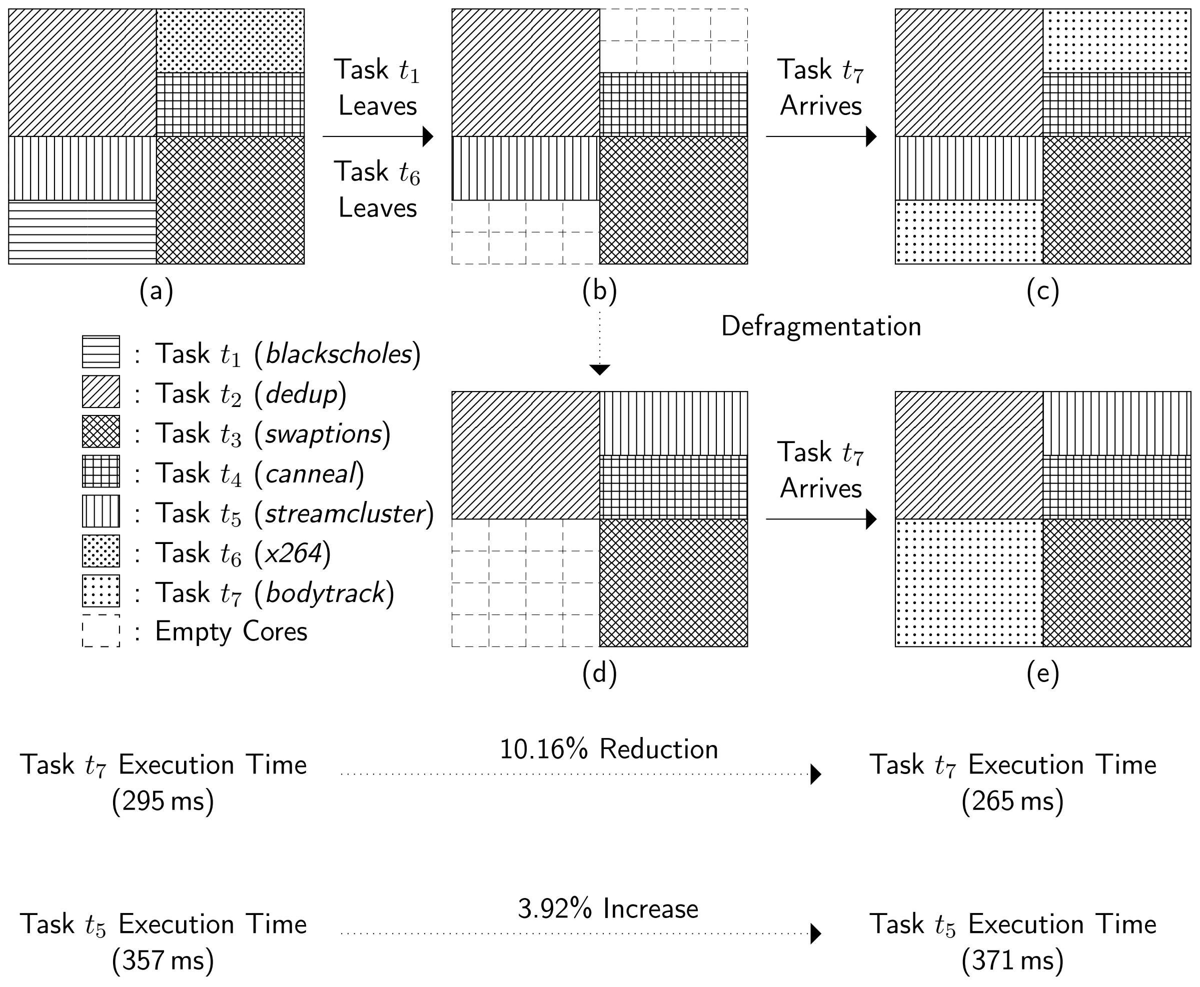

Task t_1 Leaves
Task t_6 Leaves
Task t_7 Arrives
(a)
(b)
(c)
Defragmentation
: Task t_1 (*blackscholes*)
: Task t_2 (*dedup*)
: Task t_3 (*swaptions*)
: Task t_4 (*canneal*)
: Task t_5 (*streamcluster*)
: Task t_6 (*x264*)
: Task t_7 (*bodytrack*)
: Empty Cores
Task t_7 Arrives
(d)
(e)
Task t_7 Execution Time (295 ms)
10.16% Reduction
Task t_7 Execution Time (265 ms)
Task t_5 Execution Time (357 ms)
3.92% Increase
Task t_5 Execution Time (371 ms)

Figure 6.3.: The benefit of task defragmentation in open 64-core many-core.

6.0.2. Problem Complexity

The problem of many-core defragmentation is identical to the problem of placing polyominoes. Polyominoes are geometric shapes formed by combinations of simple unit square shapes. Figure 6.4 shows five distinct shapes that can be formed by tetromino (polyomino of size four) alongside their Average Manhattan Distances (AMDs).

AMD of a polyomino shape is average of all rectilinear distances between unit squares forming the shape in two-dimensional space. Rectilinear distance is the shortest distance between two points on XY grid. A unit square is analogous to the allocation of one core on many-core. A task's performance with contiguous allocation negatively correlates to AMD of polyomino shape formed by its allocation. Figure 6.4 shows the execution time of *streamcluster* with allocations in different tetromino shapes.

The number of shapes that polyomino of given size can form grows super-exponentially due to its combinatorial nature and is given by Online Encyclopedia of Integer Sequences (OEIS) sequence *A000105* [102]. Furthermore, placing a set of polyominoes on to larger underlying polyomino without overlapping is an NP-hard problem [103]. The problem manifests while defragmenting many-cores making the problem of many-core defragmentation also hard.

This work focuses on open many-cores wherein time for both arrival and departure of a task is unknown. Incoming tasks can be allocated on many-core either contiguously or non-

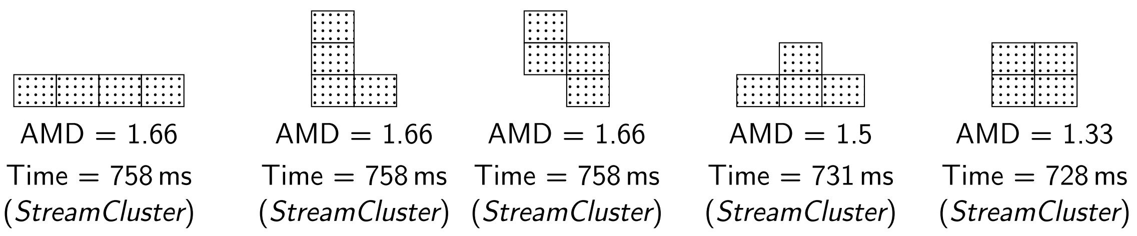


Figure 6.4.: Different distinct shapes of tetromino; polyomino of size four.

contiguously. Contiguous allocation stipulates that cores allocated to the task must always be spatially connected, whereas non-contiguous allocations enforce no such restrictions.

Contiguous allocation ensures that NoC latency experienced by task threads when communicating is minimal as they are spatially collocated. Furthermore, it also ensures isolation of inter-thread NoC communication traffic between tasks executing in parallel reducing NoC congestion. Even under contiguous allocation there will be some external NoC interference due to OS, cache-directories, and MCs on the periphery. Thus, contiguous allocation improves system performance by optimizing NoC communication. *SHiC* contiguous allocation heuristic introduced in [104] uses smart stochastic hill-climbing in an attempt to allocate incoming tasks contiguously with minimal fragmentation.

Non-contiguous allocation, on the other hand, optimizes many-core utilization at the expense of communication. It often occurs in open many-core that there are enough unallocated cores available to satisfy the requirement of an incoming task, but they are not spatially connected. Non-contiguous allocation does not wait for the required number of spatially connected cores to become available but instead allocate them immediately to whichever cores are available irrespective of their locations. This strategy reduces the waiting time for task but affects its performance throughout execution due to increased communication overhead. In multi-program workload execution, non-contiguous allocation also degrades the performance of not just the new incoming task but also previously allocated tasks due to NoC congestion. *CASqA* non-contiguous allocation heuristic introduced in [105] initially attempts to allocate incoming task contiguously, but if there are not enough contiguous unallocated cores available, then it uses nearby cores to allocate remaining task non-contiguously.

Both *SHiC* and *CASqA* attempt to reduce fragmentation. However, their efficacy is limited because they to do not perform any thread migrations. Defragmenter combines multiple sets of non-contiguous unallocated cores into a single contiguous set of unallocated cores by performing thread migrations. The incoming task is then allocated efficiently to this newly created contiguous unallocated core set. However, thread migrations involved in defragmentation introduce performance penalties on migrated threads due to cold cache misses on newly allocated cores. Therefore, defragmenter needs to be careful not to perform too many thread migrations or risk being detrimental instead of being beneficial to overall many-core performance.

Furthermore, many-core defragmenter should also be scalable so that it continues to operate efficiently as the number of cores in many-core increases. Distributed defragmenter provides the required scalability [91]. Authors in [106] presented *ADAM* heuristic defragmenter for a tiled many-core. Neighboring tiles under *ADAM* rearrange unallocated cores to make space for an incoming task via thread migrations. *ADAM* operates only on locally available information and is bound to get stuck in local minima, which may not be optimal.

Past research tackles many-core fragmentation problem by proposing suboptimal defragmentation heuristics. We, on the other hand, solve a constrained version of many-core fragmentation problem optimally in this work. Our evaluations show that our constraint-optimal

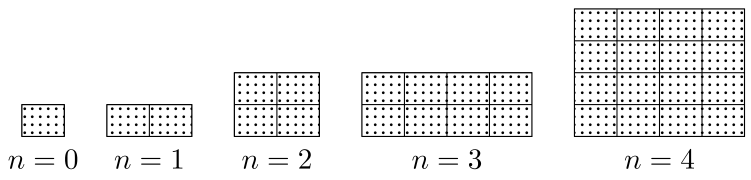


Figure 6.5.: Different size ESA polyominoes.

approach can result in substantial performance gains on many-core enforcing constraints compared to state-of-the-art heuristics.

6.0.3. Novel Contributions

In this work, we present our idea of Exponentially Separable Allocation (ESA), which defines task allocation constraints on many-core. We show that this allocation exhibits properties that allow optimal distributed many-core defragmentation. We also introduce a defragmenter called Many-Core Defragmenter (*McD*). *McD* exploits ESA properties for optimal defragmentation of many-cores. *McD* disburses its processing overhead across all unallocated cores in many-core allowing it to scale up as the number of cores in many-cores continue to increase in future.

6.1. Exponentially Separable Allocations

We begin by introducing our novel idea of ESA that specifies set of allocation constraints for many-cores. ESA puts constraints on the number of cores that can be allocated to a task, shape of polyominoes these cores can form and physical location of those polyominoes. ESA is akin to the projection of binary buddy system [107] in two-dimensional space [108], but with inherent support for distributed optimization.

6.1.1. Number Constraint

ESA requires that size of allocation must be in exponentiation series with base two (or power of two) i.e. 1, 2, 4, 8, ... 2^n cores. If a task comes with a core requirement that is not a power of two, its requirement is buffered up to next highest (ceiling) power of two. Task executes faster with more number of allocated cores as discussed in Chapter 4. Thus, by spawning more threads task would experience an equal or lower response time than response time it would have experienced if buffered cores were left idle; preventing system underutilization. On the other hand, this constraint also limits defragmentation benefits for non-scalable tasks, which are not allowed to or are incapable of spawning additional threads on buffered cores.

6.1.2. Shape Constraint

ESA requires that allocation forms rectangular polyomino with the minimum perimeter. We define polyominoes that follow shape constraint along with number constraint as ESA polyominoes. ESA polyomino of size 2^n can be obtained by symmetrically reflecting size 2^{n-1} ESA polyomino along one of its edges. If n is odd, reflection happens along the x-dimensional edge. If n is even, reflection happens along the y-dimensional edge. Figure 6.5 visualizes ESA polyominoes of different sizes. ESA also requires many-core to be an ESA polyomino.

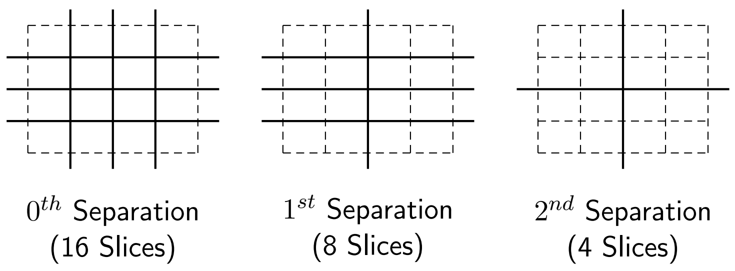


Figure 6.6.: Different size separations of a 16-core processor.

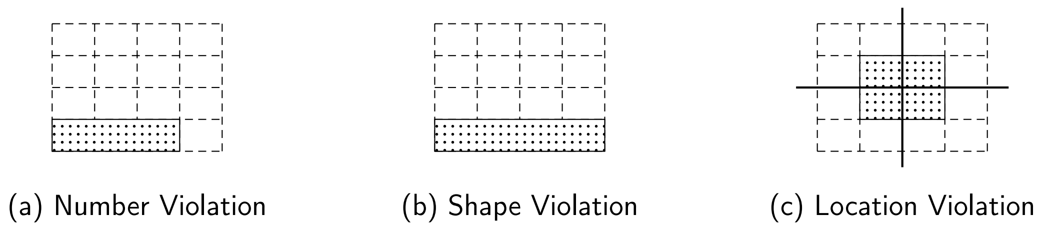


Figure 6.7.: Example of allocations that are not ESA.

6.1.3. Location Constraint

ESA requires that ESA polyomino of size 2^n must not get separated in n^{th} separation of many-core. n^{th} separation of many-core divides many-core into non-overlapping ESA polyominoes of size 2^n, individually referred to as n^{th} slice. Figure 6.6 shows some of the different size separations of a 16-core processor. Note that ESA polyomino that does not get separated in n^{th} separation will also not get separated in $(n + 1)^{th}$ separation as latter produce slices that are larger and subsume slices produced by former.

Figure 6.7 shows simple examples of allocations that are not ESA. Figure 6.7a has a task with three cores allocated to it, which violates number constraint. Figure 6.7b has a task with four cores (2^2 cores) allocated to it but polyomino these cores form violates shape constraint. Figure 6.7c has a task allocated to four cores and polyomino these cores form has the right shape. However, this polyomino gets split in 2^{nd} separation violating location constraint.

6.1.4. Optimal Defragmentation

We now present proofs for properties ESA possesses. *McD* can exploit these properties to perform optimal and distributed defragmentation of many-core that enforces ESA.

Lemma 6. *Set of ESA polyominoes with total polyomino size $\leq 2^n$ can be split into two sets of ESA polyominoes with total polyomino size $\leq 2^{n-1}$.*

Proof. Singleton set of ESA polyomino contains only one element which we cannot split further. For a set of ESA polyominoes with cardinality higher than one, the proof is equivalent to proving that we can split power of two number 2^n into two sets of the smaller power of two numbers each having a sum equal to 2^{n-1}. Figure 6.8 shows binary tree representing all possible combinations in which smaller power of two numbers can be combined to form larger power of two number 2^n. We can reach root 2^n only when constituent numbers form two separate set, each with a total sum equal to 2^{n-1}. We can extend proof to total sum $\leq 2^n$ by removing same numbers from the original set, and two derived separated sets; hence proved.

□

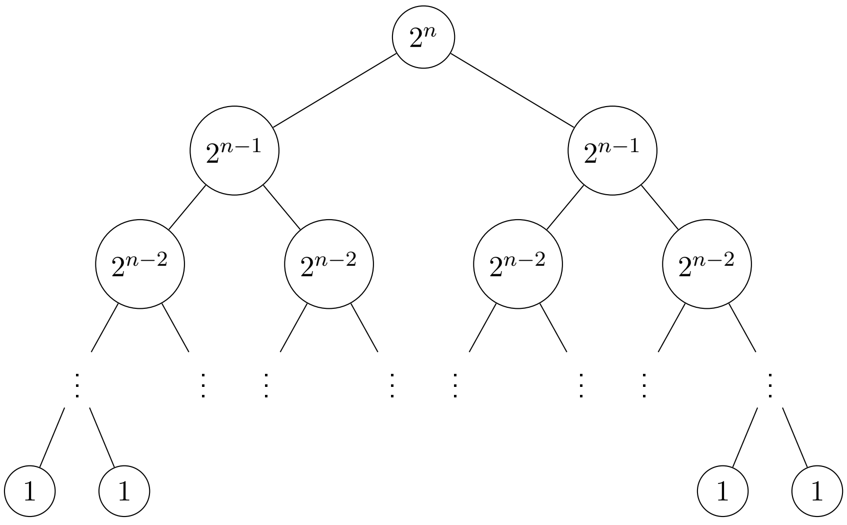


Figure 6.8.: The split of a power of two number 2^n into the smaller power of two numbers.

Lemma 7. *Set of ESA polyominoes of total polyomino size $\leq 2^n$ can be allocated without overlapping onto many-core in shape of ESA polyomino of size 2^n.*

Proof. If the set of ESA polyominoes to be allocated is a singleton, then it can be allocated on many-core, latter being ESA polyomino of greater or equal size in comparison to former. If the set has cardinality higher than one, then based on Lemma 6 it can be separated into two sets of polyominoes of total polyomino size $\leq 2^{n-1}$ each. In parallel, $(n-1)^{th}$ separation of many-core will divide it into two equivalent many-cores (two $(n-1)^{th}$ slices) each in shape of ESA polyominoes of size 2^{n-1}. Each part of the separated many-core can be allocated to one of the separated sets of ESA polyominoes. We can then allocate each part of separated many-core to one of the separated sets of ESA polyominoes. We can repeat argument recursively till we allocate all polyominoes without overlapping on many-core. The allocations obtained satisfy ESA as they violate none of the constraints; hence proved.

□

Lemma 8. *If any of n^{th} slices obtained from n^{th} separation of ESA enforcing many-core contains more than one ESA polyomino allocated to it, then all allocated ESA polyominoes in that slice are of size $\leq 2^{n-1}$.*

Proof. Under location constraint imposed by ESA, ESA polyomino of size 2^n cannot get separated in n^{th} separation of ESA enforcing many-core. In other words, it would not share n^{th} slice it is allocated on with any other ESA polyomino. Sharing can happen only in $(n+1)^{th}$ or larger slice; hence proved.

□

6.2. Defragmenter

Chapter 3 describes the common notations used to describe *McD*. We assume all the tasks to be moldable[2] in this chapter. Let G represent set of $|G|$ gaps in many-core, indexed by g_l. Let $C_{g_l} \subseteq C$ denote the set of $|C_{g_l}|$ unallocated cores forming gap g_l in shape of an ESA

[2]Refer Chapter 2.2 for the definition of a moldable task.

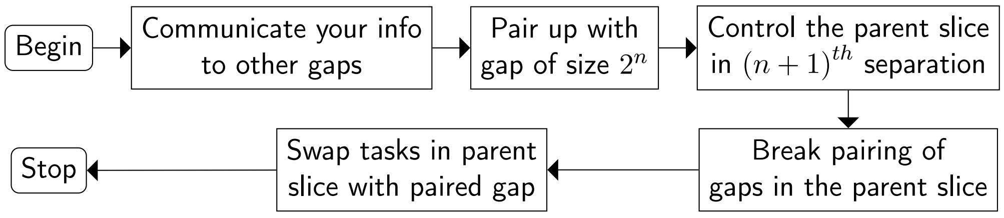

Begin
Communicate your info to other gaps
Pair up with gap of size 2^n
Control the parent slice in $(n+1)^{th}$ separation
Break pairing of gaps in the parent slice
Swap tasks in parent slice with paired gap
Stop

Figure 6.9.: Actions performed by a gap of size 2^n to merge with another gap of the same size.

polyomino. We also assume that the many-core will be itself in shape of an ESA polyomino. Let $||G||$ represent the total number of unallocated cores in the many-core.

$$||G|| = \sum_{g_l} |C_{g_l}|$$

Trivially $||G|| = |C|$ when many-core is empty. We assume First-In, First-Out (FIFO) waiting queue in our open many-core. $|C_{t_x}|$ means core requirement of task t_x waiting in front of the queue. Under ESA, $|C_{t_x}|$ must be number that is a power of two. Based on Lemma 7, we know if core requirement of task in front of the queue $|C_{t_x}| \leq ||G||$ then the task can always be allocated to ESA enforcing many-core. In non-preemptive many-core like one assumed in this chapter, if core requirement of task in front of the queue $|C_{t_x}| > ||G||$ then incoming tasks can never be allocated. The incoming tasks then need to wait in the queue for one or more of previously allocated tasks to leave many-core. The objective of *McD* is to keep the number of gaps $|G| = 1$ and that also always in an shape of ESA polyomino.

6.2.1. Merging Gaps

Merging of gaps happens in series of rounds. Figure 6.9 shows actions performed by a gap (using any one of its unallocated constituent cores) in a round. At the start of every round, gap communicates with other gaps and pairs up with one having the same size. For example, let gaps g_a and g_b pair up with each other, such that $|C_{g_a}| = |C_{g_b}|$. Now they want to merge to form a gap of size $|C_{g_a}| + |C_{g_b}| = 2|C_{g_a}| = 2|C_{g_b}|$. Let τ_{g_a} and τ_{g_b} be two slices in $\ln(|C_{g_a}|) + 1$ separation containing gaps g_a and g_b, respectively. Gap g_a (or g_b) holds half of slice τ_{g_a} (or τ_{g_b}) as unallocated cores. Based on Lemma 8, all other tasks allocated in slice τ_{g_a} (or τ_{g_b}) are self-contained in remaining filled half of the slice. Thus, gap g_a (or g_b) can swap filled half of slice τ_{g_a} (or τ_{g_b}) with unallocated cores in slice τ_{g_b} (or τ_{g_a}) to merge with gap g_b (or g_a) without violating ESA. Another pair of gaps g_c and g_d can swap in parallel as long as neither gap g_c nor g_d is inside slice τ_{g_a} (or τ_{g_b}). Otherwise, merging of gaps g_c and g_d is skipped in this round. Therefore, a merging of smaller gaps is delayed in favor of merging of larger gaps when overlapping. After all parallel swaps finish, next round begins. Rounds end when no more pairings between gaps can occur.

Theorem 5. McD *performs optimal defragmentation of many-core that enforces ESA.*

Proof. Let us assume *McD* is suboptimal. After *McD* has finished defragmentation, there does not exist gap g_y such that $|C_{g_y}| \geq |C_{t_x}|$ when the total number of unallocated cores $||G|| \geq |C_{t_x}|$, where $|C_{t_x}|$ is the core requirement of task t_x in front of FIFO queue. Based on Lemma 8, there can exist at most one gap of size $|C_{t_x}|/2$; otherwise, *McD* would have merged two of them to create a gap of size $|C_{t_x}|$. Recursively there can exist only at most one gap of size $|C_{t_x}|/4$, $|C_{t_x}|/8$ and so on. Therefore, the total number of unallocated cores $||G|| \leq |C_{t_x}|/2 + |C_{t_x}|/4 + ... + 1 \leq |C_{t_x}| - 1$. Based on this argument total number of

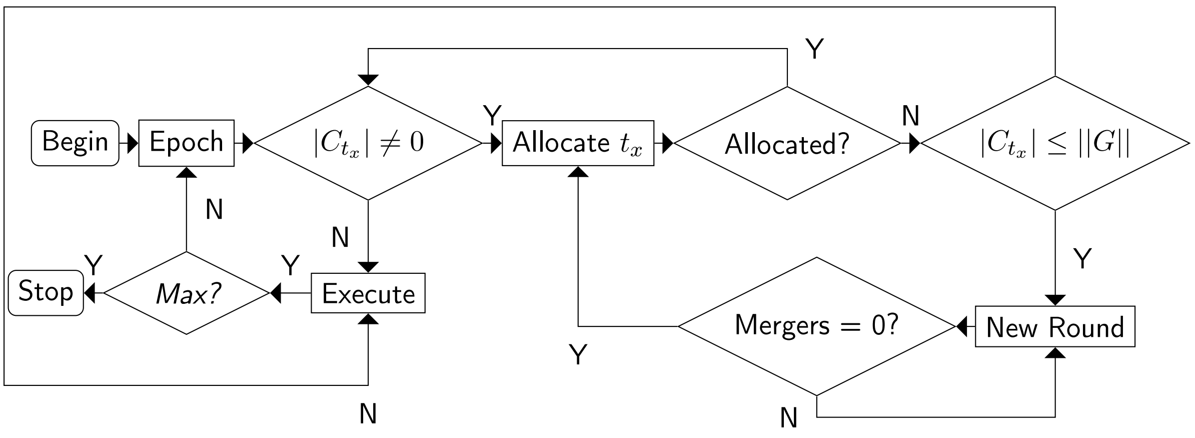


Figure 6.10.: Execution flow for *McD*.

unallocated cores $||G|| < |C_{t_x}|$, which is a contradiction to our original statement total number of unallocated cores $||G|| \geq |C_{t_x}|$; hence proved.

□

It is important to note that *McD* only claims optimality concerning system performance in the final state (defragmented) obtained. The problem for reaching one state (fragmented) to another state (defragmented) in a minimum number of steps (optimal number of task migrations) on many-core is similar to optimally solving the sliding-puzzle problem [109], which is also an NP-hard problem. We do not cover the problem of defragmentation of many-cores in the minimum number of task migration in this dissertation.

6.2.2. Execution Flow

Figure 6.10 shows execution flow of *McD*. Every scheduling epoch, *McD* can be invoked to perform allocation of incoming tasks and defragmentation, if required. Many-core is initially idle, and there is only one gap controlling all unallocated $|C|$ cores. *McD* then checks whether FIFO queue is not empty ($|C_{t_x}| \neq 0$). If it is empty, then *McD* waits for the first task to arrive. If the queue is not empty, then tasks are picked from the queue and are allocated on many-core under ESA constraints on First-Come, First-Serve (FCFS) basis till *McD* can allocate no more tasks. FCFS also ensures there is no task starvation in many-core.

Based on Lemma 7, if $|C_{t_x}| \leq ||G||$ then there is scope for more tasks to be allocated in many-core. However, if there is no contiguous gap of size $\geq |C_{t_x}|$, fragmentation prevents further new allocations. *McD* then invokes series of gap merging rounds to defragment many-core. Process of allocation of incoming tasks is invoked again once the rounds stop. *McD* iteratively repeats the process until queue empties ($|C_{t_x}| = 0$) or task in front of queue is too big to be allocated on many-core ($|C_{t_x}| \geq ||G||$). *McD* then executes allocated tasks for the time defined by scheduling epoch. Tasks which are completed leave system at the end of scheduling epoch, creating new gaps. Many-core halts when the number of completed tasks reaches user-defined *Max*. Otherwise, new scheduling epoch commences on many-core.

6.2.3. Illustrative Example

Figure 6.11 illustrates defragmentation performed under *McD* on a 32-core processor executing seven tasks. Incoming task t_8 requires eight cores. Figure 6.11a shows eight unallocated cores are available on the processor in the initial state, but these unallocated cores are fragmented all over processor in the form of four gaps of size two each. Many-core invokes *McD* for

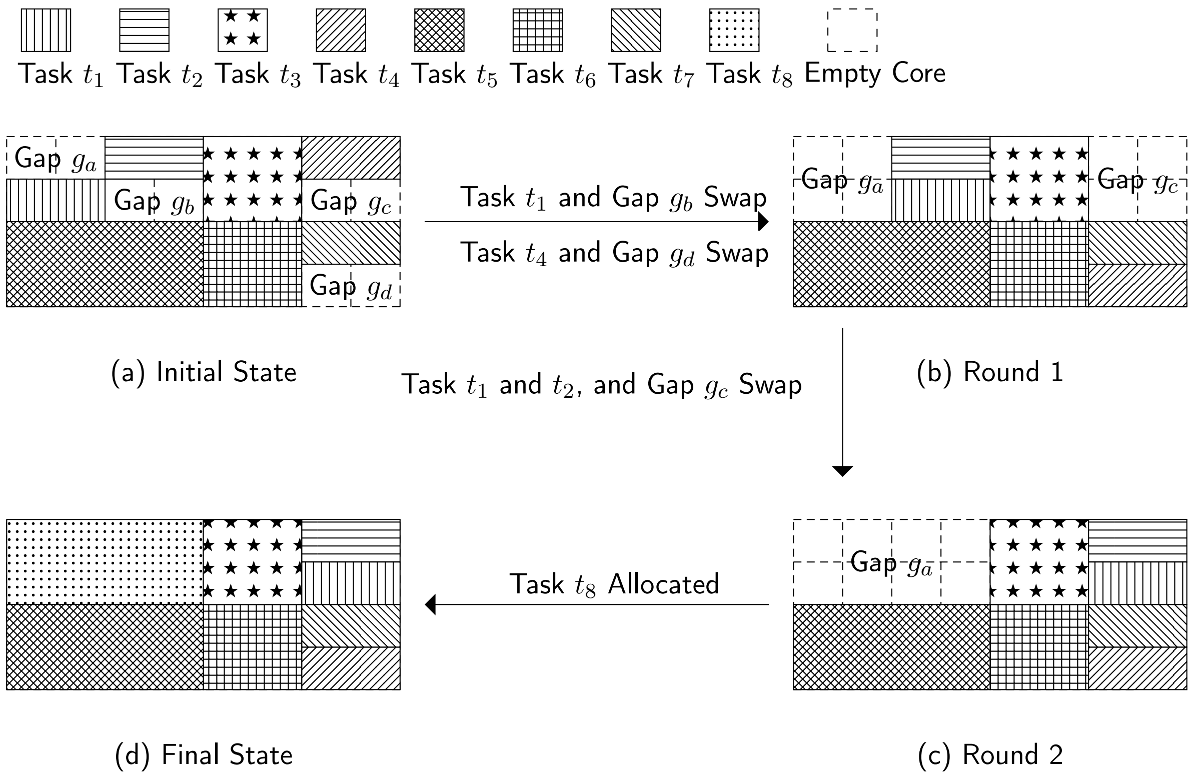


Figure 6.11.: An illustrative example of distributed defragmentation performed under *McD* on a 32-core processor with seven initial tasks.

defragmentation to allocate task t_8 contiguously on the processor. Let gaps g_a, g_b, g_c, and g_d represent four gaps on many-core in the initial state. Gap merging rounds now begin.

Round 1: gap g_a pairs up with gap g_b and simultaneously gaps g_c and g_d pair up. Gap g_a takes control of task t_1 as they share slice in 2^{nd} separation and swaps it with gap g_b. In parallel, gap g_c takes control of task t_4 and swaps it with gap g_d. Figure 6.11b shows the resultant state after swaps. Gaps g_b and g_d now no longer exist.

Round 2: remaining gaps g_a and g_c pair up. Gap g_a takes control of tasks t_1 and t_2 as they share slice in 3^{rd} separation, and swap it with gap t_c. Gap g_c now no longer exist. Figure 6.11c shows the resultant state. Rounds now stop because there are no more gaps to merge. *McD* now allocates task t_8 over gap g_a as shown in Figure 6.11d and all tasks resume execution.

6.2.4. Complexity

On $|C|$-core many-core, there can be at most $O(|C|)$ gaps to merge. This merging will take $O\left(\ln |C|\right)$ rounds under *McD*. In every round, every gap performs at worst $O\left(|C|\right)$ calculations. Thus, in total there is $O\left(|C|^2 \cdot \ln |C|\right)$ processing overhead with per-core processing overhead being $O\left(|C| \cdot \ln |C|\right)$. Every round involves broadcasting at most $O\left(|C|\right)$ messages; hence in worst-case total communication overhead is $O\left(|C| \cdot \ln |C|\right)$. Since *McD* does not require any data structure, space overhead is $O\left(1\right)$.

Table 6.1.: The system configuration of simulated many-core architecture.

Cores	64 x86-64 out-of-order cores
L1 Cache	split I & D, 16 KB, 4-way, 64 B block, 3 cycle access latency
L2 Cache	private 64 KB, 8-way, 64 B block, 8 cycles access latency
Directory	Modified Shared Invalid (MSI) coherence
Network	2-D mesh, 4 cycles per hop latency, 256 bits/cycle bandwidth, XY routing
Memory	1 GB, 80-cycle access latency, 4 MCs on 4 edges
Tasks	*blackscholes*, *bodytrack*, *canneal*, *dedup*, *ferret*, *fluidanimate*, *streamcluster*, *swaptions* and *x264*
Task Model	multi-program, multi-threaded
System Model	one thread per-core using private LLC, FIFO task queue

6.3. Experimental Evaluations

6.3.1. Experimental Setup

We evaluate *McD* using *Sniper* interval simulator [110]. In comparison to time-wise infeasible cycle-accurate simulator like *gem5* [80], *Sniper* allows for multi-program simulations in reasonable time. On the other hand, in comparison to trace-based simulators as used in Chapters 4 and 5 *Sniper* allows for more accurate and realistic multi-program simulations.

Figure 6.1 shows the conceptual block diagram of many-core architecture used in this work. The architecture consists of 8x8 cores (tiles) connected using two-dimensional mesh NoC implementing XY routing with a latency of 4 cycles/hop and links with a bandwidth of 256 bits/cycle. We used 64-core many-core for evaluations as the simulation of multi-program execution in bigger many-cores was difficult due to simulation-time constraints even with *Sniper*.

Each tile consists of out-of-order *Intel Gainestown* core running at 1 GHz implementing *x86-64* ISA with private 4-way associative 16 KB L1 instruction and data caches. Many-core distributes 8-way L2 cache (4 MB) across chip with 64 KB bank of private L2 residing with each tile. Caches are kept coherent using directory-based MSI protocol and use Least Recent Used (LRU) page replacement policy. 1 GB off-chip DRAM accessed using four MCs along four edges of many-core serves as the main memory. Hit latencies of L1 caches, home L2 bank, and main memory are set at 3, 8, and 80 cycles, respectively. We use OS-level page allocation instead of traditional address interleaved cache-directories for managing the L2 banks [111].

McD is equally applicable if LLC was shared by all cores, instead of being private. Conceptually it makes no difference for *McD* if message-passing replaces cache-coherency. Though, changes in topology such as having more than one core per tile can potentially make the problem of many-core fragmentation NP-hard again even under ESA constraints. Therefore, minor changes to underlying system architecture may or may not break *McD*'s optimality and need to be carefully studied in details individually on a case to case basis.

In software, we use multi-threaded tasks from *PARSEC* [85] suite with *sim-small* input. We choose *sim-small* input because next smaller input *sim-dev* input was not representative enough of real-world tasks, while next larger input *sim-medium* took too long to simulate. We also found instruction count of *sim-small* inputs is sufficiently large enough to stress caches on our simulated system in a meaningful way. Among thirteen available tasks in *PARSEC*, we

Table 6.2.: Number of thread spawning count (master and slaves) supported by *PARSEC* tasks.

Tasks	Threads															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
blackscholes	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
bodytrack	-	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x
canneal	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
dedup	-	-	-	x	-	-	x	-	-	x	-	-	x	-	-	x
ferret	-	-	-	-	-	-	x	-	-	-	x	-	-	-	x	-
fluidanimate	-	x	x	-	x	-	-	-	x	-	-	-	-	-	-	-
streamcluster	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
swaptions	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x264	x	-	x	x	-	-	-	-	-	-	-	-	-	-	-	-

discard two tasks *freqmine* and *vips* due to unresolved *PIN* errors. *PIN* [112] is closed-source binary instrumentation tool from *Intel* used inside *Sniper* simulator which prevents debugging of its error. Furthermore, we discard two tasks *facesim* and *raytrace* discarded due to lack of *sim-small* input. Table 6.1 summarizes configuration of simulated many-core. Table 6.2 notes the number of different threads we can use to execute different *PARSEC* tasks. All tasks were limited to produce a maximum of 16 threads.

6.3.2. Implementation Details

We integrate *McD* with default *Pinned* scheduler of *Sniper*. We implement *McD* as multi-threaded distributed task written in *C* with master-slave thread model. We spawn *McD*'s master thread at the start of the simulation and then put it to sleep. At the end of each scheduling epoch (10 ms of simulated system time), time-trigger interrupt wakes main *McD* thread in any random unallocated core. Master thread then checks the status of many-core like the number of unallocated cores and the number of tasks in FIFO queue and then determines if defragmentation is required. If defragmentation is indeed required, *McD* then determines one unallocated core within each gap which is responsible for gap's merging related calculation. For each gap, master thread spawns a slave thread on the chosen unallocated core that has the gap's responsibility. The master thread itself also holds responsibility for one of the gap.

Threads synchronize with each other using memory, and all coherency traffic travels via NoC. Migration of *PARSEC* threads is performed by *McD* threads using custom extensions to *Sniper*'s magic instructions. Suspension, resumption, and placement of *McD* threads are done by *Sniper* scheduler using default native instructions. When all *McD* threads except master thread have terminated, defragmentation is complete. Master thread then allocates new tasks from FIFO queue on the many-core and then goes back to sleep. It is important to note that all *McD* threads mostly operate on unallocated cores and minimal context switching with *PARSEC* threads is required. *McD* threads operate by manipulating thread-to-core affinities of *PARSEC* threads and bulk of actual thread migration and context switching heavy-lifting is left to default *Sniper* scheduler to be performed internally. This implementation is designed to work particularly with *Sniper* and may not be the best design for real-world many-core.

Tasks from *PARSEC* run in a master-slave configuration, where one master thread is invoked first which later spawns more threads. Therefore, in one-thread per-core execution model employed for our simulated many-core minimum number of cores we allocate to a task, in general, is two. The exception to this is *bodytrack*, *ferret*, and *x264* with the minimum number of possible core allocations being three, seven and one, respectively. Figure 6.12 shows performance

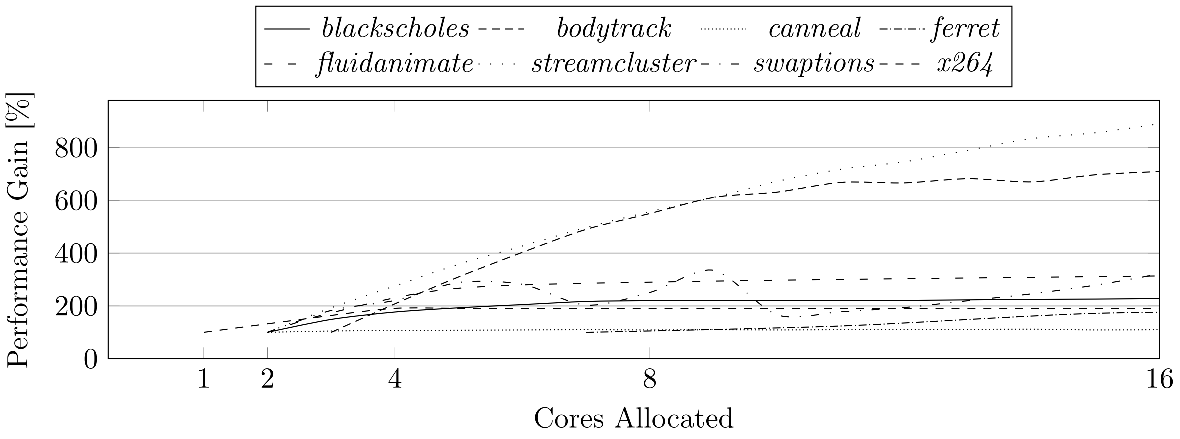


Figure 6.12.: Performance gains in multi-threaded tasks on varisized allocations.

gain in different tasks when allocated varisized ESA polyomino shape allocations with respect to the smallest size ESA polyomino shape allocation permissible. The observed performance gain, in general, starts to saturate with the increase in the size of the allocation. Performance gain saturates because of saturation in exploitable TLP in tasks, increase in inter-thread communication overheads and also because of the inability of some of the tasks to spawn enough threads to utilize all of the allocated cores. Thus, in our experiments we limit the number of cores allocated to task to level it can still draw benefits. We also set the upper limit on the size of allocation to sixteen, which is reasonable given the size of our simulated many-core.

For each reported result, we execute multiple multi-program workloads. Each workload comprises of twenty tasks with each task projecting random core requirement. Simulation-time constraints prevent evaluation of larger size or number of workloads. The arrival time of tasks in workload follows Poisson distribution. Due to peculiar nature of defragmentation problem, many-core will not require defragmentation if core requirements of tasks are too small in comparison to the size of many-core as most tasks then would fit inside comfortably. Similarly, if core requirements of tasks are too large, we will also not require defragmentation as few tasks will occupy entire many-core. Thus, we also set the lower limit and upper limit of core allocation to task to four and sixteen cores, respectively. We empirically found limits to be reasonable given the size of our simulated many-core.

6.3.3. Optimizations

We implement a couple of optimizations over default *Sniper* to improve NoC-traffic isolation in cache-coherent many-core architecture as shown in Figure 6.13. *Sniper* by default uses address interleaving for placing data on distributed L2 cache and separating accesses from MCs to main memory. Each private L2 bank stores directory entries for a predefined range of addresses. Hence, cache-directory responsible for cache blocks of given task can be present in any of the banks. Additionally, banks can use all MCs for fetching data from main memory when they encounter a miss event. This can result in tasks accessing all the banks and MCs providing no isolation. Without isolation, there are no clear benefits from defragmentation. Figure 6.13a shows two tasks t_1 and t_2 on 64-core many-core end up accessing all the banks irrespective of their location under default *Sniper*.

Inspired by D-NUCA design in [111], we ensure that banks of cores on which threads of a multi-threaded task are executing serve task's main memory accesses. Authors in [111] utilize interleaving at page granularity for allocating pages to distributed L2 cache. Whenever task requests a new page, OS chooses free pages from main memory that are serviceable by requesting

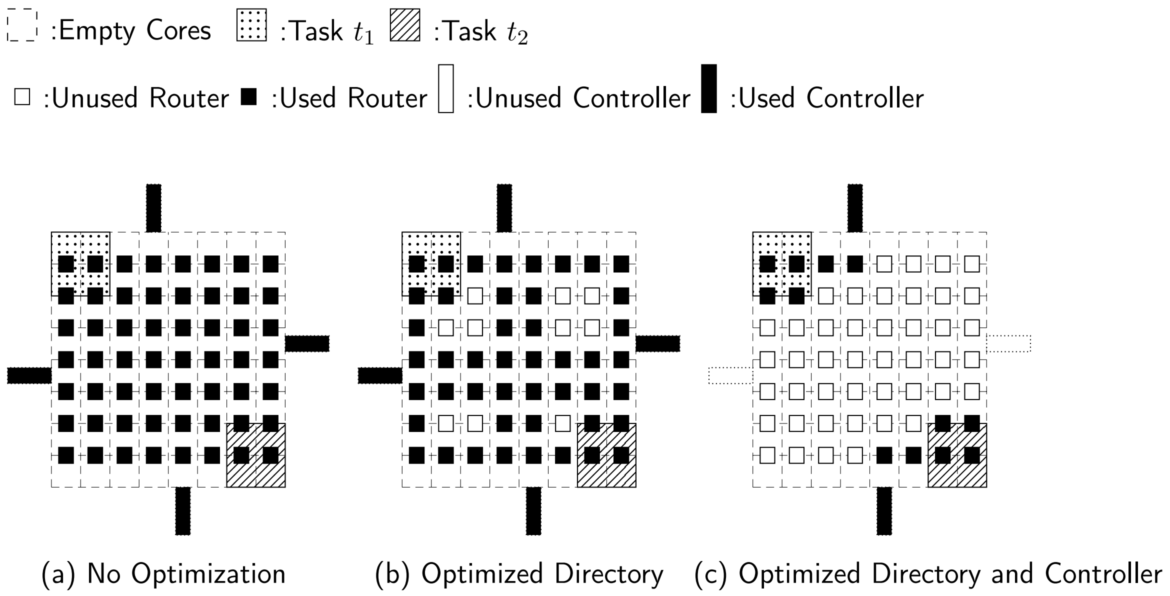


Figure 6.13.: Optimizations for achieving NoC-traffic isolation on cache-coherent many-core.

core's task allocation. Due to lack of OS in *Sniper* simulator, we were forced to achieve the equivalent effect by modifying address lookup function in *Sniper* code-base. Figure 6.13b shows advantage of using this optimization regarding bank accesses, but we still have traffic going to all MCs which breaks NoC isolation.

For preventing tasks from accessing all MCs, we make the entire address space accessible to all controllers. Parallel accesses to the same address are synchronized using an off-chip priority queue as mentioned in [113]. We also ensure that NoC-traffic on a L2 cache miss always goes to the closest MC amongst all controllers. The benefits can be seen in Figure 6.13 (c), where both allocated tasks achieve almost complete NoC isolation. To obtain maximum benefits from these optimizations, amongst all the cores allocated to a task, we place the master thread of that task on the core which is closest to any of the MCs.

In this work, we perform all simulations in the multi-program mode with full modeling of cache-contention, NoC-contention, memory-contention, and performance penalties from task migrations. Single simulation of twenty task workload took approximately on average ten hours to complete on *Intel Core i7* processor.

6.3.4. Performance Metric

Average response time is the standard performance metric [32] for open many-cores, which a defragmenter must minimize. The response time of a task is the time between its arrival and departure time. It is the sum of waiting and servicing time. Waiting time is time task spends in the queue before allocation. Servicing time is time task needs after allocation to complete execution. Average response time for the workload is mean of response times of all its tasks.

6.3.5. Basic Comparative Baselines

To show the effectiveness of *McD*, we choose to compare against two straightforward approaches *Contig* and *Non-Contig* representative of the schedulers that performs contiguous and non-contiguous allocation without defragmentation, respectively.

Contig exhaustively searches each core of many-core for contiguous compact allocation of incoming task. It then allocates the task to the first set of contiguous cores that can satisfy task's requirement. Since *Contig* always allocates incoming task in only best possible shape, it always results in optimal servicing time for the task. However, since it waits passively for contiguous cores in ideal shape to become available, it results in suboptimal waiting time.

Non-Contig, on the contrary, allocates the incoming task to any of unallocated core on many-core with no regards to allocation contiguity. *Non-Contig* results in optimal waiting time for the task as it allocates the task as soon as required number of unallocated cores are available irrespective of their locations. However, since allocation is neither compact nor contiguous, it results in suboptimal servicing time.

Contig and *Non-Contig* represent two extreme points in performance spectrum each guaranteeing optimality for one aspect of performance myopically, while disregarding other. In multi-program execution, waiting and servicing time are not entirely independent. For example, a task that holds cores longer than necessary under suboptimal servicing time ends up adding additional delay to waiting time for all tasks in the queue. *McD* defragmenter introduced in this work can optimize both aspects of performance (waiting and servicing time) together, but only for ESA enforcing many-core. Comparison of *McD* against *Contig* and *Non-Contig* results in deeper insights than comparison with previously proposed heuristic schedulers for preventing fragmentation. Given lack of guarantees on either waiting time or servicing time in heuristics, it is difficult to say what part of performance spectrum they represent. Heuristics are also very sensitive to input workloads and can perform unexpectedly good or bad.

6.3.6. Heuristic Baselines

For complete coverage, we also compare against heuristic approaches designed to address fragmentation. We believe *SHiC* [104], *CASqA* [105], and *DeFrag* [114] are state-of-the-art heuristics for fragmentation-aware contiguous allocations, fragmentation-aware non-contiguous allocations, and defragmentation, respectively. All compared heuristics were designed originally to operate with profiled tasks whose task-graphs (thread-spawning and inter-thread communication patterns) were assumed to be deterministic and predictable. Such task-graphs are not readily available for real-world representative *PARSEC* tasks. Furthermore, it is also not trivial to predict when a task will spawn a particular thread in multi-program execution. We neither assume nor have complete profile information of all tasks to implement heuristics strictly in their original forms. Hence, we were required to slightly adapt heuristics to make them work on our infrastructure. Originally all compared heuristics were evaluated on a trace-based simulator. We reimplemented them on more real-world representative *Sniper*.

SHiC [104] uses a stochastic approach in an attempt to allocate incoming tasks contiguously with minimal fragmentation. It employs smart hill-climbing for finding suitable unallocated core candidate to perform contiguous allocation around in consideration with already allocated tasks to improve overall contiguity. In each iteration of hill-climbing, *SHiC* selects a random unallocated core and calculates "square-factor" around that core. Square-factor is the number of unallocated cores in largest square of unallocated cores that can be made around selected core plus number of unallocated cores in next largest incomplete square of unallocated cores. *SHiC* marks selected unallocated core as a possible candidate for allocation if its square-factor is equal to incoming task's core requirement. Otherwise, *SHiC* performs a random walk from the selected core towards one of eight adjacent cores of the selected core that has lower or higher square-factor depending upon whether selected core has higher or lower square-factor than the incoming task's core requirement, respectively. The random walk terminates after $N/2$ steps if it fails to find a candidate. Hill-climbing itself terminates after $2 + \sqrt{APPS}$ iteration, where $APPS$ is the number of tasks currently allocated on many-core. If hill-climbing finds multiple candidates for allocating incoming task, one closest to edge of many-core is selected. *SHiC*, thereafter, performs compact contiguous allocation around finally selected candidate. Authors of *SHiC* have shown it to be superior to several previously proposed similar heuristics.

CASqA [105] is an extension of *SHiC* for non-contiguous allocation. It allows the user to adjust contiguousness of allocation using a threshold. However in this chapter, we set the threshold to a value that allows for unbounded non-contiguousness. *CASqA* uses the same stochastic hill-climbing algorithm as *SHiC* to find the first unallocated core candidate to perform rest of the core allocations around. It then starts exploring squares with incrementally increasing radius for more unallocated cores and stops when enough cores are found.

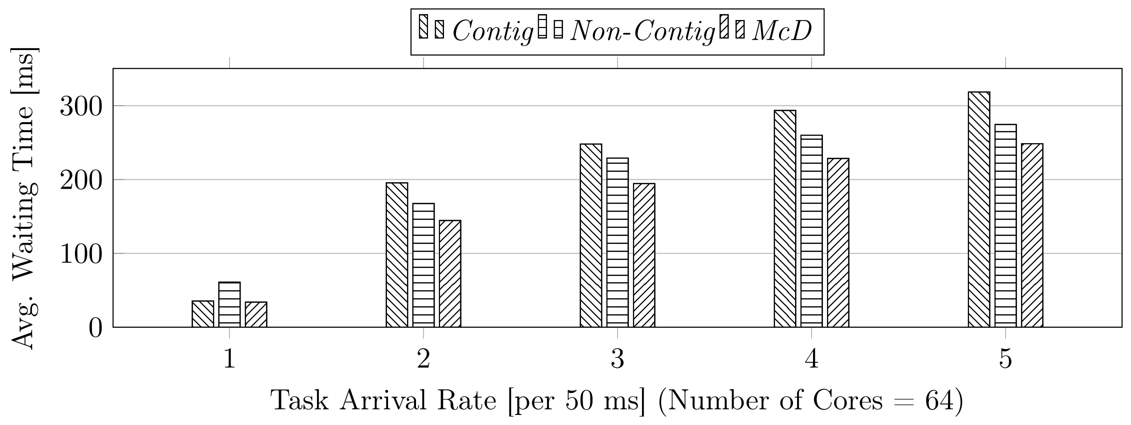
DeFrag's [114] decision as to whether to perform defragmentation after task leaves is based on fragmentation metric. Authors of [114] define the difference between the expected number of unallocated cores required by incoming task and size of the largest contiguous set of unallocated cores available as the fragmentation metric. To avoid excessive task migration overhead involved in defragmentation, *DeFrag* invokes defragmentation only when fragmentation metric is positive. As and when *DeFrag* invokes defragmentation, all the unallocated cores calculate distance from all other unallocated cores. *DeFrag* selects the unallocated core with minimum total distance as center core. *DeFrag* then finds a convex contiguous region of size equal to the total number of unallocated cores around the selected central core. Unallocated cores then travel hop by hop to the closest position in the convex contiguous region, performing thread migrations on busy cores in their paths. Authors' initially proposed algorithm to determine "Minimal-Cost Migration Path" requires complete task profiles, which we neither assume nor possess. We instead choose shortest path algorithm to find migration path of the unallocated core to the convex contiguous region. Finally, the incoming task is allocated compactly in the convex contiguous region provided it is large enough.

6.3.7. Performance Under Power-of-Two Constraint

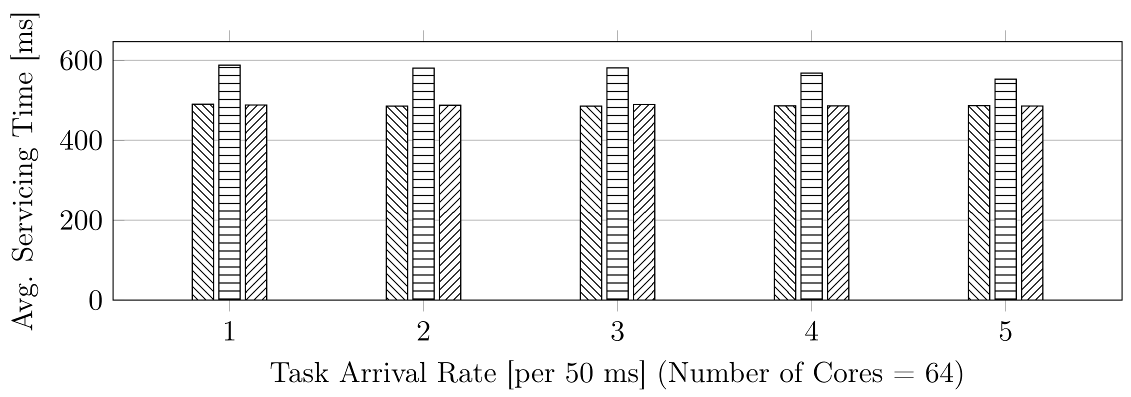
We begin by evaluating performance when tasks in workloads are only allowed to project requirement of 2^n cores as stipulated under ESA number constraint. We execute workloads with different arrival rates under various approaches on open 64-core many-core. For an open many-core, increase in arrival rate translates to increase in its load.

Figures 6.14a, 6.14b, and 6.14c record average waiting, servicing, and response time under different arrival rates, respectively. *McD* always outperforms comparative baselines in both waiting and servicing time. Hence, *McD* always results in superior response time. Initially at lower arrival rates when many-core is underloaded, allocated tasks are sparsely distributed over many-cores, and most of the incoming tasks can be allocated efficiently without waiting by all approaches. Importance of defragmentation increases as the load on many-core increases. Figure 6.14 shows *McD* provides greater performance gains when many-core is substantially loaded. Performance gains from *McD* saturate in overloaded many-core with very high arrival rates. Improved performance under *McD* comes from its ability to create a compact contiguous space for every incoming task as soon as possible resulting in minimum possible waiting time. It also ensures all tasks are always executed efficiently with least possible communication overheads resulting in minimal servicing time. Figure 6.14 shows *McD* can result in up to 8.81% and 19.53% additional performance in comparison to *Contig* and *Non-Contig*, respectively.

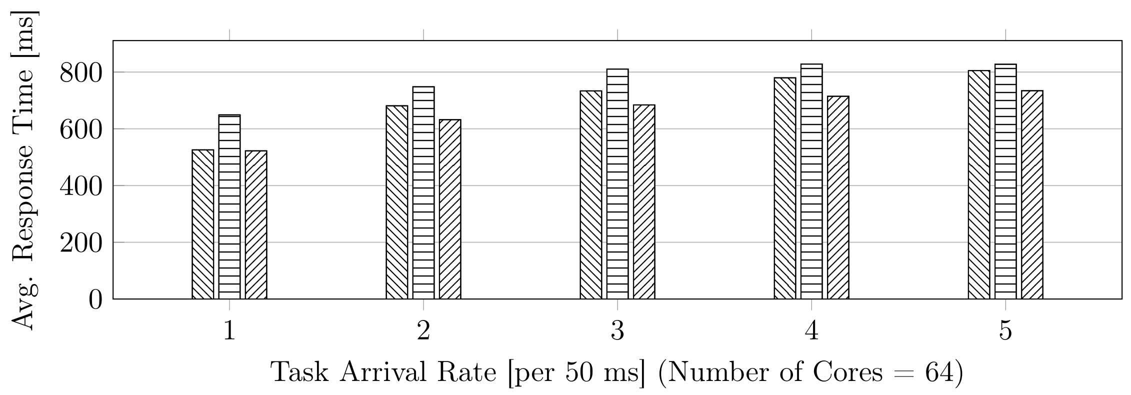
In Figure 6.15a, we explain observed performance gain under *McD* using insights from relevant performance counter for a randomly selected workload. We normalize observations under all approaches against observations made under *McD* so that all of them can be shown concisely on the same graph. We observed that NoC packets transmitted remains nearly same under all approaches. Still, we observed that packet delay due to NoC latency (*Queue-Delay*) and delay due to NoC congestion (*Contention-Delay*) is several times higher for *Non-Contig*. These delays significantly degrade performance under *Non-Contig*. The number of instructions processed by



(a) Waiting Time



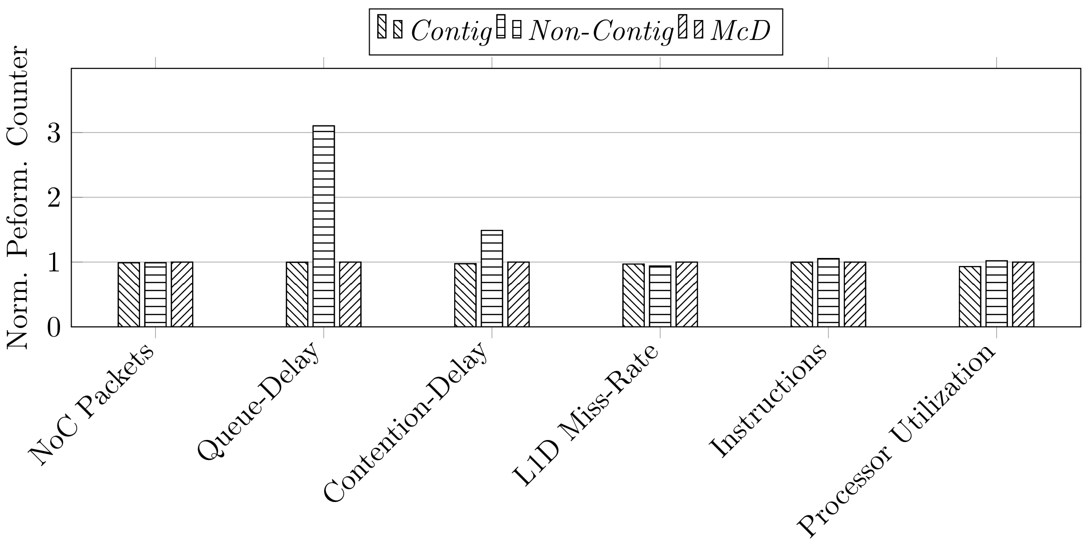
(b) Servicing Time



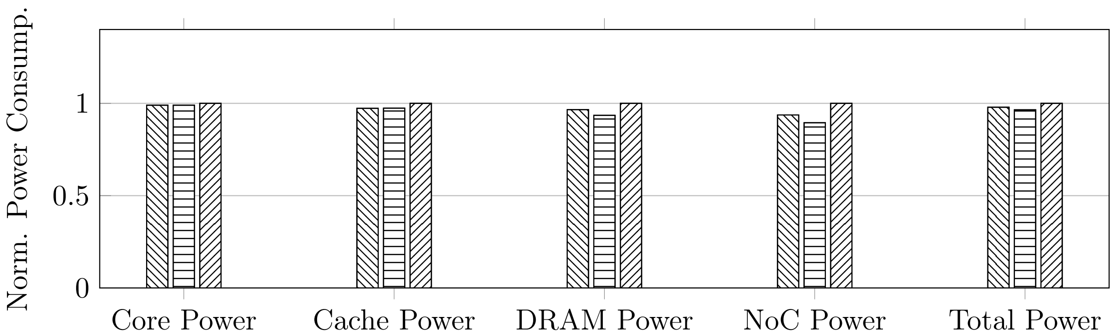
(c) Response Time

Figure 6.14.: Performance comparison between *McD* with *Contig* and *Non-Contig* under power of two ESA number constraint.

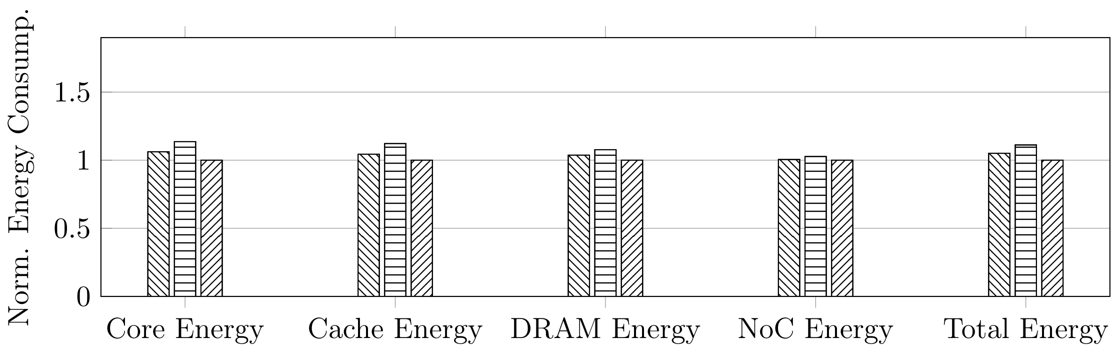
Non-Contig is significantly higher because of additional processing by tasks actively waiting longer for thread-synchronizations to complete. This processing also results in higher processor utilization under *Non-Contig*, but this increased utilization in practice is detrimental instead of beneficial for overall many-core performance. Reduced performance under *Contig* is mainly due to lower processor utilization as it keeps tasks waiting longer in the queue. This utilization drop results in lower congestion in NoC links, but reduced congestion still cannot compensate



(a) Performance Counters



(b) Power Counters



(c) Energy Counters

Figure 6.15.: Observed values (normalized against *McD*) for relevant counters for given workload when executed under different approaches.

for performance drop due to low processor utilization. The L1 data cache-miss rate is higher for *McD* than other approaches because of involved defragmentation related thread migrations.

Figure 6.15b and Figure 6.15c show selected workloads normalized power and energy consumption, respectively. In comparison to baselines, *McD* pushes to execute more load in parallel; as a result, we see all system components having higher power consumption. Still, executing more load in parallel allows it to finish execution faster resulting in lower energy consumption

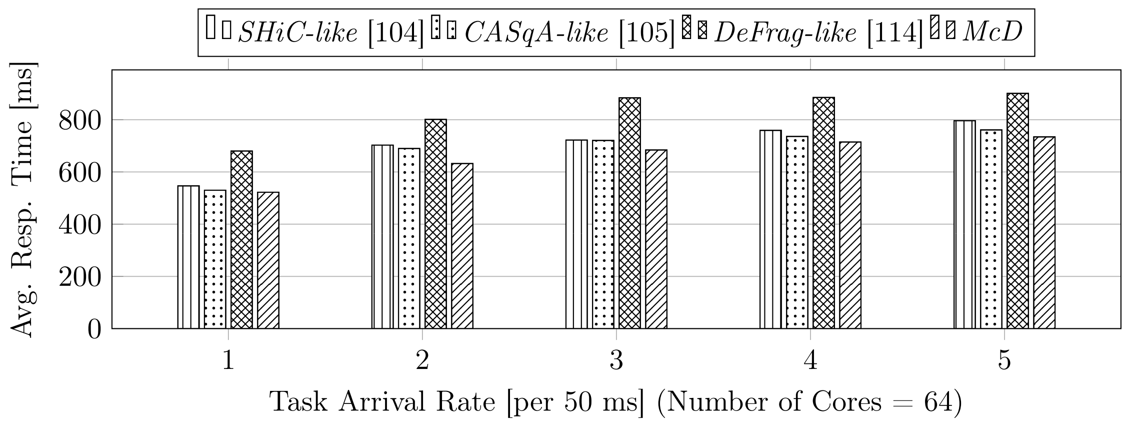


Figure 6.16.: Performance comparison between *McD* with *SHiC*, *CASqA*, and *DeFrag* under power of two ESA number constraint.

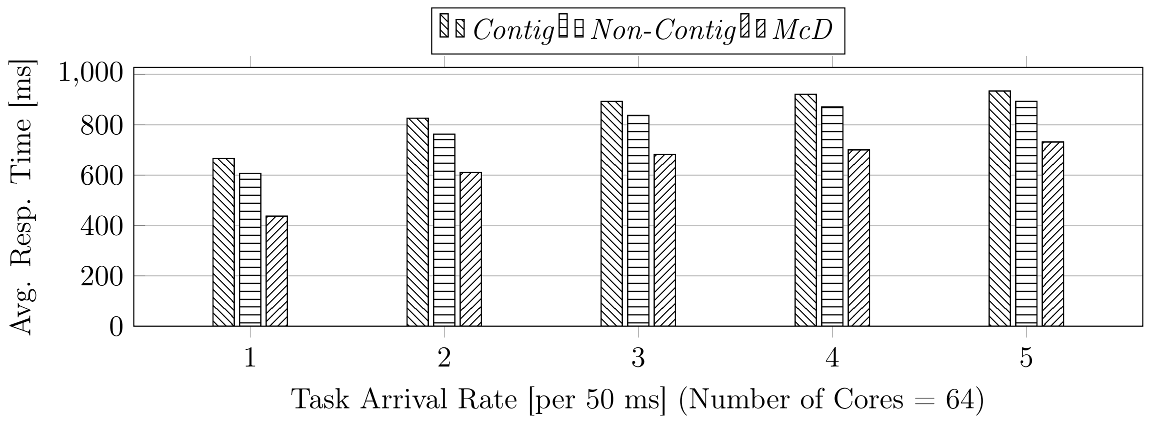


Figure 6.17.: Constraint-free performance comparison between *McD* with *Contig* and *Non-Contig* when tasks are capable of spawning additional threads.

for all many-core components. Overall *McD* results in 2.06% increase in total power consumption while reducing total energy consumption by 4.85% of many-core.

Figure 6.16 shows performance comparison between *McD* and adapted versions of state-of-the-art heuristics designed to tackle fragmentation. *SHiC-like*, *CASqA-like*, and *DeFrag-like* symbolically represent reimplemented versions of *SHiC*, *CASqA*, and *DeFrag*, respectively. We observe that *McD* can result in up to 12.54%, 8.35% and 24.09% improved performance in comparison to *SHiC-like*, *CASqA-like*, and *DeFrag-like*, respectively. *SHiC-like* and *CASqA-like* perform worse because of same reasons as *Contig* and *Non-Contig*; a combination of suboptimal servicing and waiting time. Further given their stochastic nature, their performance does not just vary with input but also based on seed used for randomization. We also found that hop by hop thread migration approach used by *DeFrag-like* is expensive as it leads to substantial displacement of existing tasks and also does not preserve their contiguity resulting in inferior performance. On the other hand, under ESA *McD* is always optimal irrespective of input.

6.3.8. Constraint-Free Performance with Scalable Tasks

McD is ideally designed to perform optimally under ESA constraint which stipulates core requirement of all tasks in powers of two. *McD* needs to buffer the number of cores allocated to task to next higher power of two without ESA constraints. For example, if a task comes with

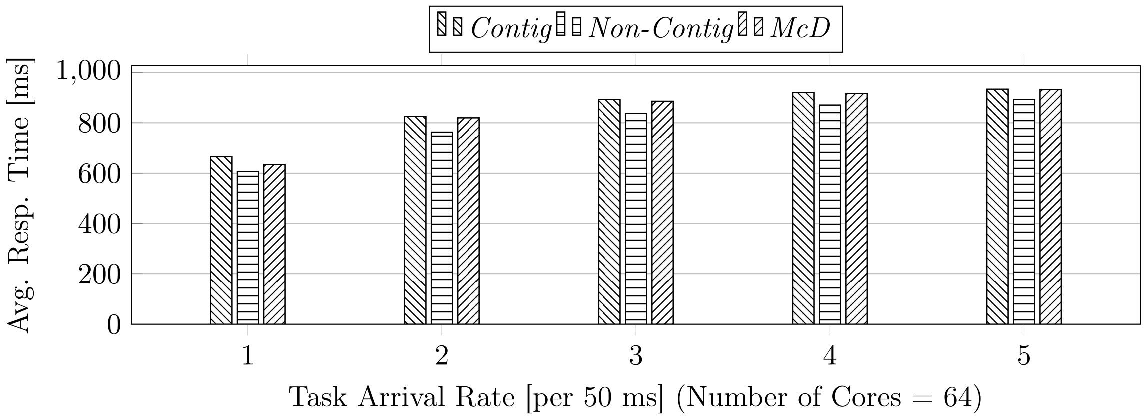


Figure 6.18.: Constraint-free performance comparison between *McD* with *Contig* and *Non-Contig* when tasks are not capable of spawning additional threads.

the fixed requirement of seven cores it must be allocated eight cores. Buffering can lead to the problem of many-core underutilization due to intra-task fragmentation or internal fragmentation if tasks do not use the buffered cores. *McD* designed to minimize inter-task fragmentation or external fragmentation is not able to compensate for this internal defragmentation. None of the comparative baselines require any buffering.

To prevent system underutilization, *McD* allows tasks to spawn threads even on buffered cores. This spawning is permissible because our *PARSEC* tasks are flexible in terms of threads they spawn. The number of threads spawn by *PARSEC* task is fixed once its main thread starts but before execution begins, the maximum number of threads it is allowed to spawn can be passed as parameter to its main thread. Most of the tasks support many different values of maximum thread count parameters, which *McD* can exploit.

Since execution time of tasks is in general monotonically nondecreasing with the number of allocated cores, all tasks will execute faster resulting in a more responsive open many-core. Figure 6.17 shows *McD* performs better than comparative baselines with scalable tasks even when the power of two core requirement is not enforced. Note that randomized workload used in Figure 6.14 and Figure 6.17 are different and hence numbers are not directly comparable.

6.3.9. Constraint-Free Performance with Non-Scalable Tasks

McD can be severely handicapped if tasks are not able (or not allowed) to spawn additional threads on buffered cores. Empty buffered cores can then cause substantial system underutilization, and even our basic comparative baselines are capable of outperforming *McD*. Figure 6.18 shows limitations of *McD* wherein *Non-Contig* now outperforms *McD* by 4.35% because tasks are not allowed to spawn additional threads. Performance gap will widen even further against heuristic baselines. This result brings forth drawbacks of constraint-optimal like *McD* in general, where the price of maintaining optimality may be too high. Therefore, we must not use *McD* with tasks which are incapable of scaling up their thread count.

6.3.10. Scalability

Simulation-time for many-core is directly proportional to the number of instructions simulated in *Sniper*. To best of our knowledge, there is no simulator other than trace-based simulators that can simulate a thousand-core many-core under heavy-load in a reasonable amount of time. This simulation-time constraint makes it difficult to obtain overhead numbers directly from a

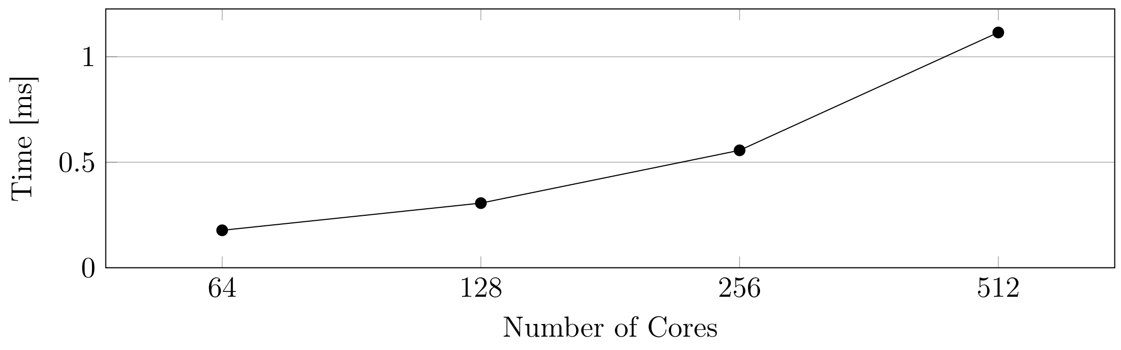


Figure 6.19.: Worst-case problem-solving time taken by *McD* on varisized many-cores.

multi-program simulation of *PARSEC* tasks on a realistic simulator like *Sniper* to demonstrate scalability benefits *McD* can potentially offer in the real-world.

On the other hand, stand-alone execution of *McD* algorithm for large-size input is still time-wise feasible. Hence, we execute *McD* with varisized worst-case representative inputs as distributed task over simulated many-cores with 64 cores or more in *Sniper* and report problem-solving time. This execution time incorporates both communication overhead of *McD* threads communicating through memory via NoC as well as their processing overheads. We believe this is closest we can get to obtaining real-world overheads of *McD* on large size many-cores. This overhead is directly comparable to the response time of *PARSEC* task themselves.

Figure 6.19 shows time it takes for *McD* to perform worse-case many-core defragmentation for 64-core to 512-core many-core. It takes *McD* 1.115 ms to solve worst-case defragmentation problem on 512-core many-core. For scheduling epoch of 10 ms used in this work, this results in the worst-case overhead of 11.15% on 512-core many-core. Worst-case defragmentation overhead on 64-core many-core stands at acceptable 1.77%.

6.4. Summary

In this chapter, we have addressed the problem of many-core defragmentation, known to be NP-hard. To make problem tractable, we simplified it to a problem that can be solved optimally in polynomial time by introducing the concept of ESA for many-cores. ESA puts constraints on allocations on many-core allowing for its optimal distributed defragmentation in polynomial time. We also introduced defragmenter called *McD*, which exploits ESA. *McD* disburses defragmentation related processing overhead across all cores in many-core allowing it to scale up as the number of cores in many-cores continues to increase.

Our experiments show that defragmentation under *McD* increases performance as well as reduces the energy consumption of many-core with minimal increase in its power consumption. Since *McD* is proven optimal, it provides maximum possible performance under ESA constraints which cannot be surpassed by any other algorithm. Though we also observed, performance gains from *McD* without ESA constraints were limited.

McD only works with many-cores with D-NUCA caches. Many-cores also come with S-NUCA caches. There is no benefit of co-locating threads from a task on many-core with S-NUCA caches and hence there is no benefit in defragmenting it. Still, a scheduler aware of S-NUCA design can exploit knowledge of the design to improve performance. We introduce a scheduler for many-cores with S-NUCA caches in next chapter.

7. Many-Core Task Scheduler for S-NUCA Caches

We present a scheduler for many-core with S-NUCA caches in this chapter.[1] We use a many-core architecture similar to the tiled many-core architecture used in Chapter 6, but with one significant difference beside the use of S-NUCA. Cores now share banks of distributed L2 LLC cache which were previously private. Figure 6.1 shows the abstract block diagram of many-core.

Access to banks from cores does not have uniform access latency due to physically distributed LLC. Therefore, such a cache is called Non-Uniform Cache Access (NUCA) cache. NoC-based many-cores primarily employ NUCA caches as bus-based physically consolidated caches used in multi-cores do not scale up well in many-cores [115].

Many-cores can employ one of several different types of memory-to-cache address mapping policies [116]. S-NUCA is a static policy for NUCA caches, wherein the mapping of memory addresses to banks is interleaved over available cache lines statically at design-time; . S-NUCA due to its inflexibility is not as efficient at run-time as OS managed flexible policies such as D-NUCA. We explored the use of D-NUCA in Chapter 6. On another hand, we can implement S-NUCA efficiently in the hardware independent of OS [117].

Figure 7.1 shows how S-NUCA cache accesses are distributed amongst different banks of 64-core many-core when we execute an instance of the four-threaded *streamcluster* on many-core in isolation. We observe that execution results in the access of all banks of many-core without any regard to which cores we pin the threads. This access pattern is also not fixed for given task in multi-program execution as banks accessed by a task not just depends upon the interaction of the task with LLC but also on interactions of other tasks with LLC. This behavior makes profiling or predicting the pattern of S-NUCA cache accesses of the task across all banks difficult in practice [118].

The standard approach for allocating task's threads on many-core is to allocate them in spatially compact square-like shapes [119]. Similar resource allocation problems in grid computing [120] inspire this approach, where the focus is to minimize network hops resulting from shape of allocations on a two-dimensional grid [121]. Chapter 6 shows how we apply this approach successfully to many-cores with D-NUCA caches. Unfortunately, this approach does not work on many-core with S-NUCA caches, where optimizing for physical proximity within threads from the same task can potentially have no apparent benefits.

Even though it is difficult to determine set of cores to pin task's threads under S-NUCA that can potentially result in the best performance for a task, the probability that core on an average would result in higher performance is higher if the core has lower AMD associated with it than its peers. Average of all rectilinear distances between a core and every other core in many-core defines AMD. This behavior under S-NUCA introduces inherent design-time performance heterogeneity in different cores of many-core based on their spatiality even if all cores themselves are perfectly homogeneous. Scheduler oblivious to this heterogeneity would inadvertently make poor allocations to many-core with S-NUCA caches while executing multi-threaded multi-program workloads.

[1]The work presented in this chapter was originally published in [10] ©2018 *EDAA*.

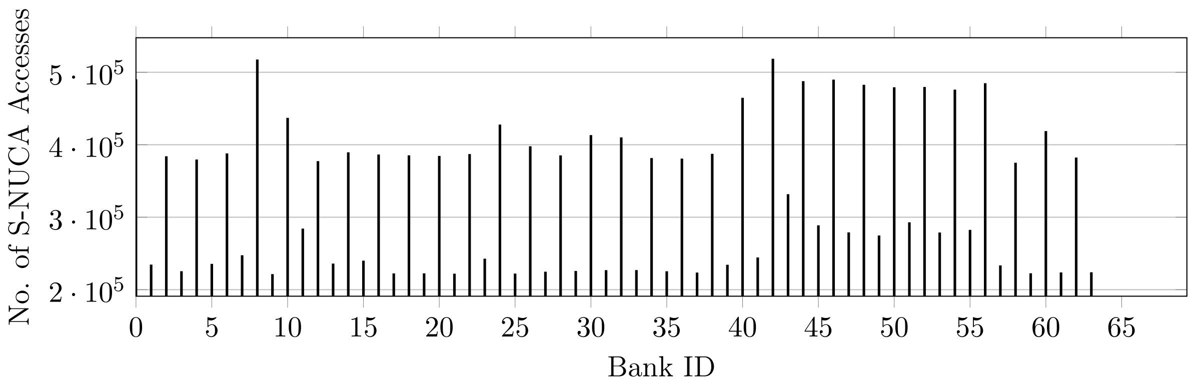


Figure 7.1.: Distribution of S-NUCA cache accesses for four-threaded instance of *streamcluster* amongst the LLC banks of 64-core many-core.

7.0.1. Novel Contributions

We characterize performance heterogeneity in cores of many-core with S-NUCA caches while executing multi-threaded workloads. Based on our observations made during characterization, we introduce a scheduler called *S-NUCA Many-Core Scheduler (SNMS)* which exploits this heterogeneity to extract more performance in comparison to state-of-the-art generic many-core scheduler while executing multi-threaded multi-program workloads.

7.1. Execution Characteristics

We begin by characterizing the performance of multi-threaded tasks on many-core with S-NUCA caches. Observations made in this section will form the foundation for the design of our proposed scheduler in next section.

By topological design, not all cores of many-core are equidistant from each other, and inherently some of the cores have lower AMDs than others. Figure 7.2 shows AMDs of all cores in 64-core (8x8) many-core. Cores closer to center by design have lower AMD than cores farther away from the center. Figure 7.3 shows performance gains for four-threaded instances of different tasks when we pin their four threads across cores with lowest AMD of four vis-a-vis when we pin their four threads across cores with highest AMD of seven. All tasks execute significantly faster on cores with lower AMD compared to cores with higher AMD.

It is also crucial that all the concurrently executing threads of a multi-threaded task which synchronize over a barrier experience near-equal performance, otherwise the slowest thread will become a bottleneck limiting the overall performance gains [122]. We observe this behavior in master-slave thread design of *PARSEC* [85] tasks which represent tasks from both embedded and HPC domains. In *PARSEC* task, the master thread is invoked first which then spawns multiple slave threads. Once all slave threads finish, only then master thread terminates itself to complete task execution. Figure 7.4 shows speedups observed in eight-threaded instances of *streamcluster* when we distribute seven of its slave threads between low-performance (high AMD) cores and high-performance (low AMD) cores in different combinations. We observe that there is a statistically significant jump in the performance gains for *streamcluster*only when all its slave threads execute on the high-performance cores.

The relationship between single master thread and multiple slave threads is of different nature but is equally important. Master and slave threads are joined together in a sequential relationship. Slow execution of any one of two can potentially prolong total execution time of a task. Thread which executes slowly becomes the bottleneck limiting gains as described by

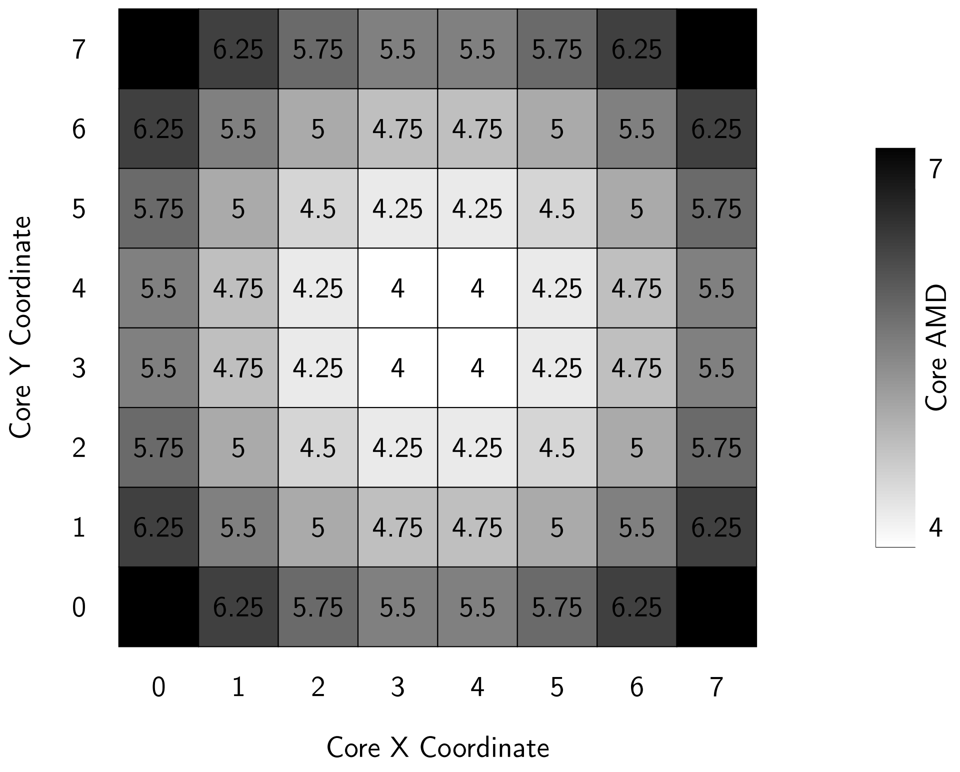


Figure 7.2.: Design-time AMDs of different cores of 64-core (8x8) many-core with S-NUCA caches.

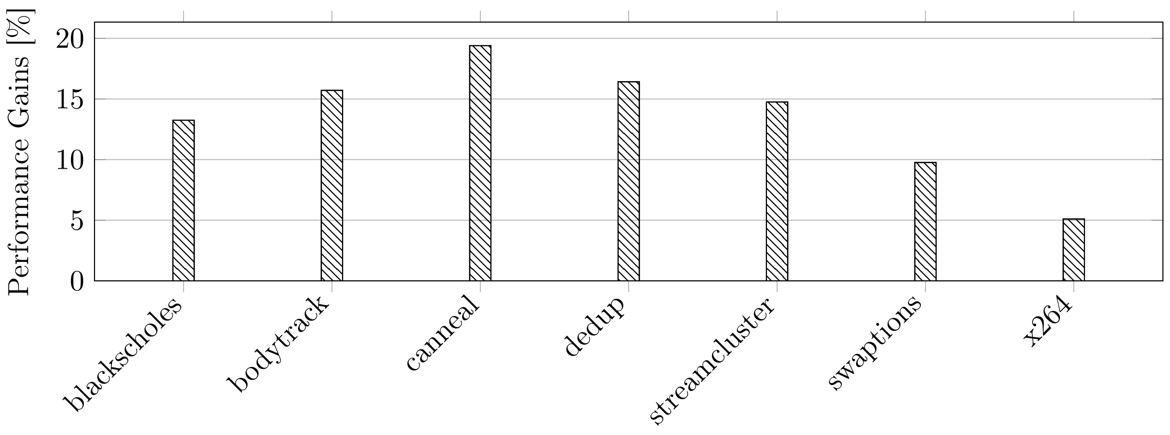


Figure 7.3.: Performance gains in four-threaded instances of tasks when we pin their four threads across cores with the lowest performance in comparison to when we pin their threads across cores with the highest performance of 64-core (8x8) many-core with S-NUCA caches.

Amdahl's law [122]. Figure 7.5 shows performance gain in different four-threaded instances of tasks when we place only their master thread on high-performance cores and when we place only their slave threads on high-performance cores. Performance gains are measured against baseline performance when we place all their threads on low-performance cores. Tasks perform differently based on the importance of master thread and slave threads in their overall performance. Amongst seven tested tasks capable of producing four-threaded instances, *blackscholes* and *canneal* are most sensitive to the performance of master thread. All remaining tasks are most sensitive to the performance of slave threads.

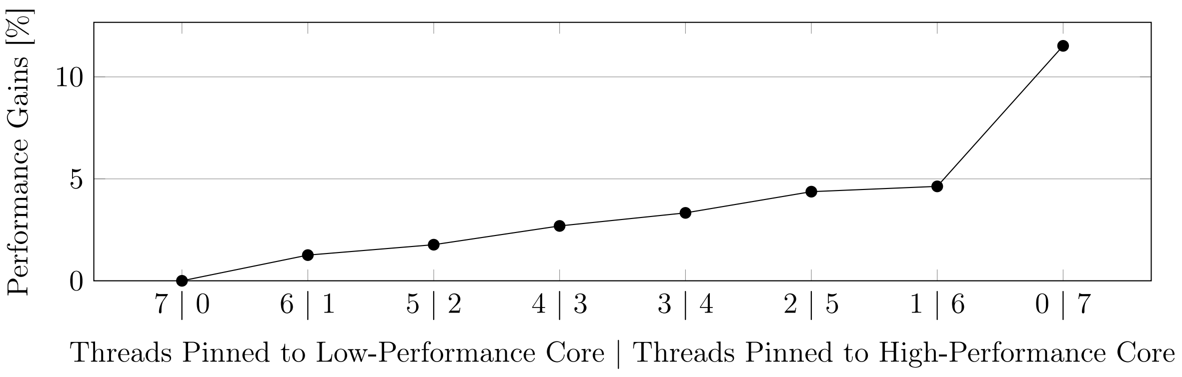


Figure 7.4.: Performance gains in eight-threaded instances of *streamcluster* when we pin their seven slave threads in different combinations on high-performance and low-performance cores.

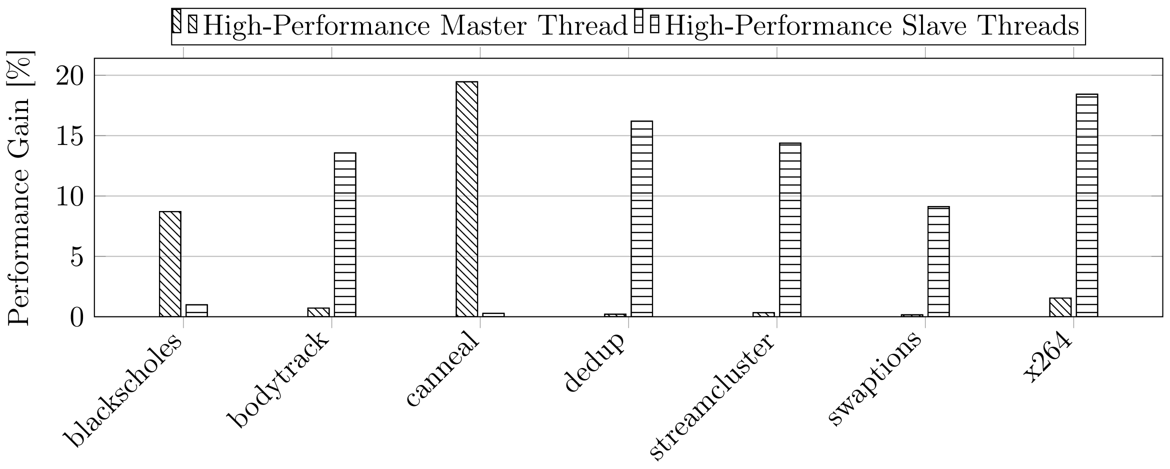


Figure 7.5.: Performance gains in four-threaded instances of tasks when we pin their master thread on low-performance core or when we pin their slave threads on low-performance cores against baseline performance when we pin all their threads on low-performance cores.

7.1.1. Characterization Recap

Based on our observations, we conclude that performance of cores negatively correlates with their topology induced AMDs. We also conclude that to prevent performance degrading bottlenecks; it is best to pin all slave threads of a multi-threaded task to cores that have same or near-similar AMD. Depending upon the sensitivity of master and slave threads to overall task performance we can allocate them to cores with different AMDs. It is best to allocate all master and slave threads to cores with near-similar AMDs if sensitivity information is not available to prevent formation of any bottleneck.

7.2. Shortcomings of State-of-the-art

CASqA [105] is state-of-the-art generic many-core scheduler when operating with rigid tasks[2] executing under one-thread per-core model [69]. *CASqA* attempts to allocate threads of an incoming rigid task in contiguous square-like shape around initial node selected using a stochastic hill-climbing algorithm [104]. Furthermore, it prefers allocation close to edges of many-core to

[2]Please refer Chapter 2.2 for the definition of a rigid task.

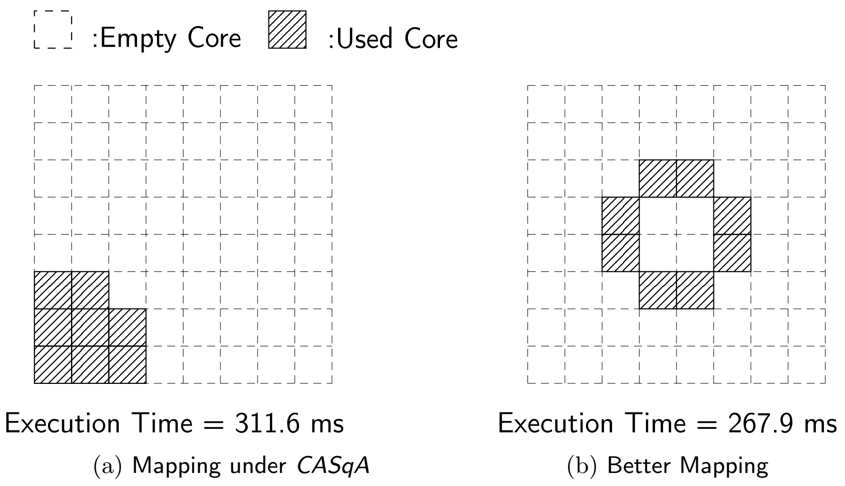

:Empty Core
:Used Core
Execution Time = 311.6 ms
(a) Mapping under *CASqA*
Execution Time = 267.9 ms
(b) Better Mapping

Figure 7.6.: Better performing allocation for eight-thread instance of *streamcluster* on idle 64-core many-core with S-NUCA caches in comparison to allocation under state-of-the-art *CASqA*.

minimize fragmentation which can reduce the potential for future compact contiguous allocations. *CASqA* breaks spatial contiguity when there are not enough contiguous cores available to perform square-like non-contiguous allocation to improve throughput by increasing system utilization. Relaxation in contiguity can be adjusted using a user-defined parameter under *CASqA*. We, however, only cover *CASqA* version that permits unlimited non-contiguous allocations in this chapter. Furthermore, *CASqA* is designed to operate with tasks that come with task-graphs. We modify *CASqA* to operate with master-slave tasks such as *PARSEC* tasks.

We now show short-comings of *CASqA* on many-core with S-NUCA caches using an illustrative example in Figure 7.6. Figure 7.6a shows how eight-threaded instance of the *streamcluster* could be possibly allocated by *CASqA* in square-like shape on idle 64-core many-core using its stochastic algorithm. Figure 7.6b shows alternative thread allocation more suitable for many-core with S-NUCA caches for eight threads of *streamcluster* in the same scenario. Experiments show that execution time of *streamcluster* decreased by 14.02% under non-compact non-contiguous allocation shown in Figure 7.6b in comparison to compact contiguous allocation shown in Figure 7.6a. Note that result only holds for many-cores with S-NUCA caches.

Observations made in Section 7.1 can explain the performance gain in Figure 7.6. Performance of *streamcluster* in both allocations is determined by its bottleneck thread. Core with highest AMD allocated to the task, in turn, determines bottleneck thread. Highest AMD of core allocated in allocation shown in Figure 7.6a is 7 whereas all cores allocated in allocation shown in Figure 7.6b have the same AMD of 4.25. Hence, *streamcluster* executes faster with allocation shown in Figure 7.6b than allocation shown in Figure 7.6a.

7.3. Scheduler

Chapter 3 describes common notations used to describe *SNMS*. We use open many-core with FIFO queue similar to Chapter 6 in this chapter. We assume all tasks to be rigid. $|C_{t_x}|$ denotes core requirement of incoming task t_x in front of the queue. Since many-core operates with one-thread per-core model, *SNMS* can allocate task t_x only when the number of unallocated cores in many-core is more than its core requirement $|C_{t_x}|$.

A denotes set of $|A|$ AMD classes into which we can classify the unallocated cores in $|C|$; indexed by a_m. $|a_m|$ represents the number of available unallocated cores of class a_m. $[a_m]$ means the numerical AMD value associated with cores of class a_m. We also define comparison operation between the classes and say $a_m > a_{m'}$ if the performance of cores in class a_m is more than the performance of cores in class $a_{m'}$. Since the performance of core is inversely proportional to its AMD, it implies $[a_m] < [a_{m'}]$.

M denotes set of subsets of AMD classes A; indexed by m_n representing all possible allocations (mappings) using different combinations of classes in A that can all satisfy core requirement $|C_{t_x}|$ of task t_x.

$$m_n \in M \iff \left(\sum_{a_m \in m_n} |a_m| \right) \geq |C_{t_x}| \tag{7.1}$$

Solution-set M theoretically contains all possible allocations for task t_x though in practice we do not need to enumerate them all at run-time. We still need to guarantee that we find best amongst them. We measure the quality of allocation using two different functions. First function $P(m_n)$ captures the performance of allocation m_n. Based on observations made in Section 7.1, the performance of allocation negatively correlates to AMD of the core with highest AMD amongst all cores contained in allocation.

$$P(m_n) = 1 / \max_{a_m \in m_n} [a_m] \tag{7.2}$$

Second function $D(m_n)$ captures AMD dispersion in allocation m_n. Difference between maximum and minimum AMD of cores contained in allocation defines its dispersion. Dispersion has value zero if cores contained in allocation are all from the same class.

$$D(m_n) = \max_{a_m \in m_n} [a_m] - \min_{a_{m'} \in m_n} [a_{m'}] \tag{7.3}$$

We choose to prioritize performance over dispersion. Therefore, we give preference to task ready for execution over a task that is yet to execute. An advance heuristic can also only partially prioritize performance over dispersion to derive more overall many-core performance. Performance $P(m_n)$ and dispersion $D(m_n)$ default to values $-\infty$ and ∞, respectively if allocation m_n is empty. Under *SNMS*, allocation m_n is superior to another allocation $m_{n'}$ if former has either higher performance or has lower dispersion with same performance than latter.

$$\begin{aligned} m_n > m_{n'} \iff & P(m_n) > P(m_{n'}) \vee \\ & \big(D(m_n) < D(m_{n'}) \wedge P(m_n) = P(m_{n'}) \big) \end{aligned} \tag{7.4}$$

7.3.1. Algorithm

SNMS uses a standard Branch and Bound (BnB) algorithm [123] to determine the allocation for an incoming task on many-core. Complete BnB algorithm would have been computationally infeasible on many-core at run-time because of large problem size and NP-hard complexity of allocation problem under consideration [103]. Still, we can effectively deploy it in this work because observations made in Section 7.1 substantially reduce search-space. Without our observations, each core of many-core is potentially unique design point for BnB to evaluate. With the help of our observations, we can classify cores based on their AMDs. For example, we can divide 64 cores of many-core shown in Figure 7.2 into nine unique classes - AMD 4 to 7. Since all cores in each class are potentially equivalent to *SNMS* with respect to their potential, worst-case search depth of BnB reduces to nine from 64. BnB depth of nine is still computationally feasible at run-time on many-cores.

The goal of *SNMS* is to find the best allocation for incoming task t_x with core requirement $|C_{t_x}|$ given allocations of already allocated tasks. Figure 7.7 shows search-space tree of empty

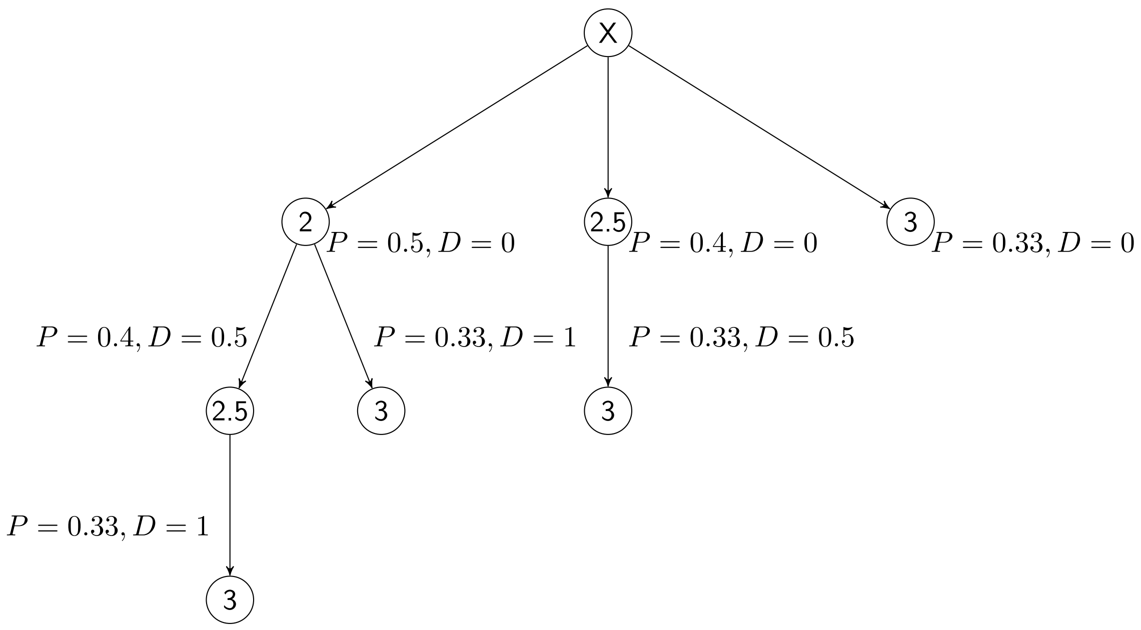


Figure 7.7.: Search-space tree for 16-core (4x4) processor with three unique AMD-based core classes.

16-core (4x4) processor with cores divided into three unique AMD-based classes – 2, 2.5 and 3. Value of node in the tree represents numeric value of AMD class *SNMS* considers for containment in possible allocation for task t_x. The path to the node represents allocation (partial or complete) in which *SNMS* uses at least one core from class represented by the node, and also uses at least one core from class represented by all its parent. Each node in search-space tree shown in Figure 7.7 is accompanied by the respective value of performance and dispersion function if the allocation is to include node and its parents but not its children.

In search-space tree shown in Figure 7.7, by design child nodes always go from lower to higher AMD values. AMD of the parent node is always lower than all its children nodes in the tree. As a result, performance always decreases, and dispersion always increases as we traverse down the tree. Therefore, search for the better solution in the tree can be bounded by not exploring node's children if the performance of node falls below already known allocation of higher performance (Equation 7.4). Search can also be bounded if the performance of node is same as known allocation but has higher dispersion associated with it than known allocation.

Algorithm 1 shows pseudo-code for *SNMS* that uses depth first search to find the best allocation m^* through recursively extending partial allocation $m^{\#}$, which gets bounded by the value of performance and dispersion function as soon as *SNMS* finds the first valid allocation. *SNMS* only needs to iterate over AMD classes that have at least one unallocated core. If best allocation m^* has more than $|C_{t_x}|$ cores, then lower performing cores from class with higher AMD in m^* are allocated first to task t_x till it has $|C_{t_x}|$ cores allocated.

7.4. Experimental Evaluations

7.4.1. Experimental Setup

We evaluate *SNMS* with nearly the same experimental setup as setup used to evaluate *McD* in Section 6.3.1. Crucial difference being the use of S-NUCA caches instead of D-NUCA caches. All cores now share L2 cache banks of LLC instead of banks being private. *Sniper* uses S-NUCA by default when simulating NUCA architectures. We implemented *SNMS* and *CASqA* in *C++* by forking original code of *pinned* scheduler shipped with *Sniper*.

Algorithm 1 Proposed algorithm for *SNMS*.

```
Input: SNMS $(|C_{t_x}|, m^{\#})$;
Output: $m^*$;                              ▷ $m^*$ is a global variable initialized to NULL.
 1: Stack $S = \{\}$;
 2: for $a_m \in A \ \nabla[a_m]$ s.t. $|a_m| \neq 0 \wedge [a_m] > \max_{a_{m'} \in m^{\#}}[a_{m'}]$ do
 3:     Push $a_m$ into $S$;
 4: end for
 5: while $S$ is not empty do
 6:     $NEXT$ = Pop $S$
 7:     $m^{\#} = m^{\#} \cup NEXT$;
 8:     if $P(m^{\#}) > P(m^*) \vee \big(D(m^{\#}) < D(m^*) \wedge P(m^{\#}) = P(m^*)\big)$ then
 9:         if $\sum_{a_m \in m^{\#}} |a_m| \geq |C_{t_x}|$ then          ▷ A valid best solution yet.
10:             $m^* = m^{\#}$;
11:         else
12:             SNMS $(|C_{t_x}|, m^{\#})$;          ▷ Recursively search deeper for a solution.
13:         end if
14:     end if          ▷ Search depth bounded by no further recursive exploration.
15:     $m^{\#} = m^{\#}$ - $NEXT$
16: end while
17: return $m^*$;
```

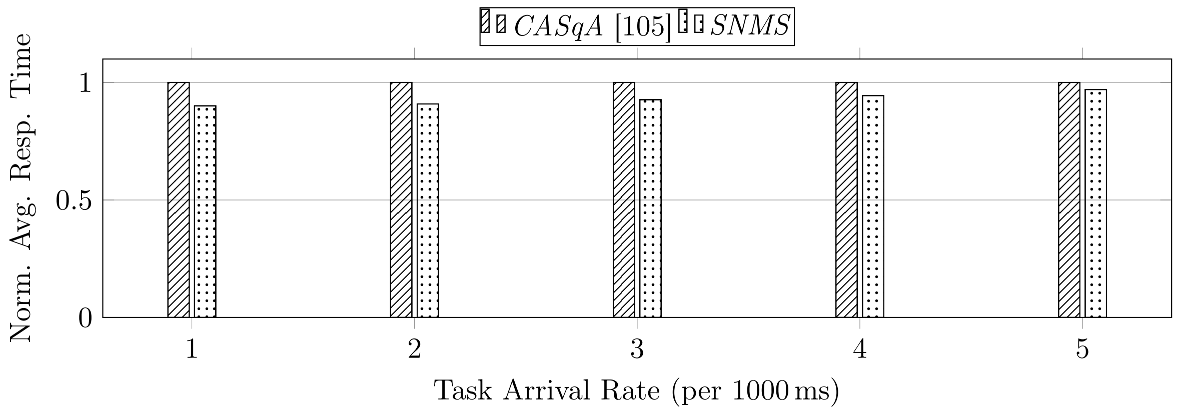


Figure 7.8.: Performance of 64-core open many-core with S-NUCA caches with multi-program workload arriving at different arrival rates under different schedulers.

7.4.2. Performance

We execute same multi-program workloads on 64-core open many-core with S-NUCA cache under two schedulers *CASqA* and *SNMS* to evaluate their efficacies. Workloads are repeated with different values of arrival rate parameter to simulate different levels of many-core load. A load induced by workload on many-core increases with increase in arrival rate. Average response time experienced by tasks composing workload when managed by scheduler serves as the performance metric. Lower average response time reflects higher scheduler performance.

Figure 7.8 shows performance under *SNMS* is superior to *CASqA* under all loads, and *SNMS* can result in up to 9.93% increase in performance. We observe that performance gains under *SNMS* over *CASqA* decrease with increase in many-core load. As a load of many-core increases, low-performance cores even under *SNMS* must be allocated to some task in order to prevent under-utilization of many-core and hence potential to improve overall average response time

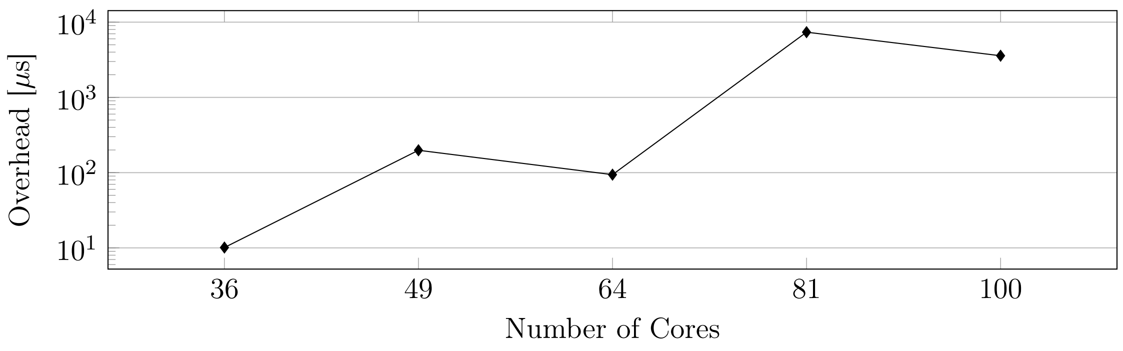


Figure 7.9.: Measured worst-case task scheduling overheads for *SNMS* on varisized many-cores.

decreases. Experiments corroborate our claim that scheduler aware of S-NUCA design can improve the performance of many-core with S-NUCA caches.

7.4.3. Overhead

Figure 7.9 shows worst-case scheduling overheads of *SNMS* in real-world time for 6x6 36-core many-core to 10x10 100-core many-core on a logarithmic scale. Average overhead in real-world would be much smaller than reported worst-case overheads. Worst-case problem-solving time of *SNMS* on 64-core (8x8) many-core is minuscule 94 μs translating into the worst-case overhead of just 0.09% at run-time for scheduling epoch of 10 ms. Worst-case problem-solving time rises sharply on 81-core (9x9) many-core to 7364.1 μs translating into the unsustainable worst-case overhead of 73.64% at run-time. Scheduling using light-weight heuristics [124] would be more suitable for 81-core many-core than BnB algorithm used in this work.

This experiment supports our argument that *SNMS* is practical on 64-core many-core. We also note that *SNMS* maybe not be fast enough on many-cores of larger size. Interestingly, the overhead of *SNMS* on a 64-core many-core is less than overhead on a 49-core many-core. This observation holds because the number of unique AMD classes in 8x8 many-core is nine against 10 in 7x7 many-core. As a result, search-space for *SNMS* on 64-core many-core is smaller than search-space on 49-core many-core resulting in a lower overhead on the 64-core many-core.

7.5. Summary

In this work, we proposed scheduler called *SNMS* for task scheduling on many-core with S-NUCA caches. We characterized performance heterogeneity introduced in cores of many-core by executing multi-threaded workloads on them. We then presented BnB algorithm that enables *SNMS* to exploit this heterogeneity for extracting up to 9.93% more multi-program performance in comparison to the state-of-the-art generic many-core scheduler on 64-core many-core. Proof-of-concept simulation predicts that *SNMS* will operate efficiently in practice on 64-core many-core with negligible 0.09% worst-case scheduling overhead.

All schedulers presented till now have ignored power consumption of many-core. With the failure of Dennard Scaling [17] and emergence of Dark Silicon [19], not all cores in many-core can operate at their peak performance. DVFS technology allows cores to trade-off performance with power consumption. Many-core must operate within a strict power budget, but there is some flexibility in distributing the power budget amongst cores. We study the problem of power budgeting in many-cores in next chapter.

8. Probabilistic Many-Core Task Governor

We present scheduler (governor) for many-core power budgeting in this chapter.[1] Due to the failure of Dennard Scaling [17] power density of many-cores has been increasing with every reduction in technology node. Limited power dissipation capacity of many-core, thereby, requires adherence to strict power budget called Thermal Design Power (TDP) [125].

Many-cores processors are capable of executing scads of multi-thread tasks in parallel. Unfortunately, TDP restricts many-core from executing all its tasks at peak performance simultaneously. Continuous operation at power consumption beyond TDP may cause severe damage to the processor. Furthermore, tasks go through various execution phases during their lifetime that determines how well they can exploit part of TDP allocated to them [126]. Therefore, it is necessary to ensure proper rationing of TDP amongst tasks. Governors are OS sub-routines responsible for the judicious and safe use of TDP. DVFS is knob available to governors for performing phase-aware power budgeting between tasks [127]. DVFS allows different cores of many-core to operate at different frequencies and voltages. When operating at higher DVFS level, cores execute threads of task faster provided task is in processing intensive phase but at the cost of more power consumption. Governor will end up only wasting power if it increases core frequencies using DVFS when a task is memory-bound.

DVFS-speedup[2] is metric used for measuring performance gain obtained from DVFS. Figure 8.1 shows how DVFS-speedup of a single-threaded version of *bodytrack* changes as it goes through different phases of its execution. In many-core restricted by TDP, a task should operate at higher DVFS level only when it can derive considerable DVFS-speedup as it may deprive other tasks operating in parallel from raising their performance. Furthermore, for the sake of fairness, all tasks must be given equal opportunity to use higher DVFS level.

Power has always been prime design constraint in processors [128]. Authors in [59] recently proposed core-level power budget for processors, but till date, most commercial processors operate with chip-level power budget TDP [125]. Use of governors for keeping processor operating close to TDP has been well-studied for many-cores [119]. Still, continuous trend of adding more cores to processors [21] warrants more scalable techniques for power budgeting.

Centralized bounded state-search power budgeting techniques for multi-cores [129] are too slow when applied to many-cores. Authors of [130] proposed multiple light-weight power budgeting heuristics for many-cores amongst which greedy algorithm called *SortedWS* provided high performance with low overhead. Orthogonally, authors in [35] proposed distributed approach for power budgeting for improved scalability. Still, all previous techniques remain non-probabilistic inherently limiting their scalability.

To best of our knowledge, probabilistic power budgeting remains unexplored concerning multi/many-cores. In other domains, probabilistic models for power budgeting have been applied to solve large size problems in data centers [131] and wireless sensor networks [132].

We make an argument in this work that constant monitoring of task phases is not required when a large number of independent tasks are running on many-core as total DVFS-speedup $\alpha(S)$ for any given many-core state S is autonomously self-stabilizing. Figure 8.2 shows this behavior where instantaneous total DVFS-speedup of 256-task (1024-thread) running using higher DVFS level on 1024-core many-core has very low standard deviation from average.

[1]The work presented in this chapter was originally published in [7] ©2017 *EDAA*.

[2]Refer Chapter 3 for the definition of DVFS-speedup

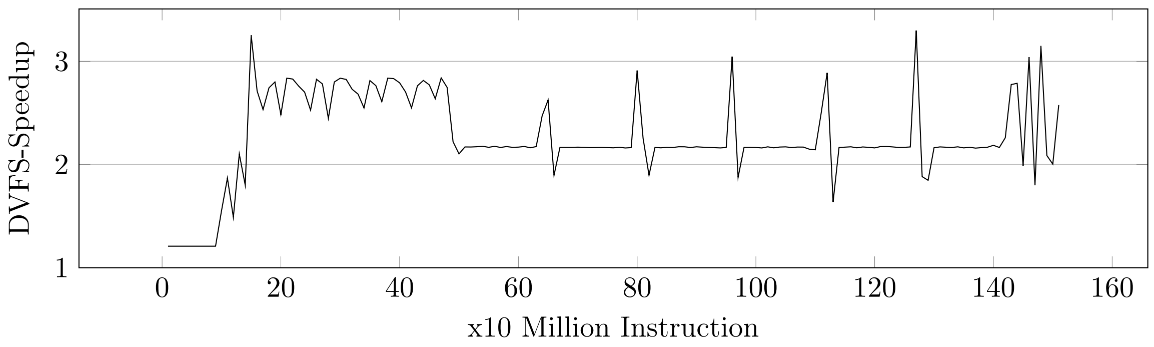


Figure 8.1.: Execution profile of single-threaded *bodytrack* showing variation in DVFS-speedup.

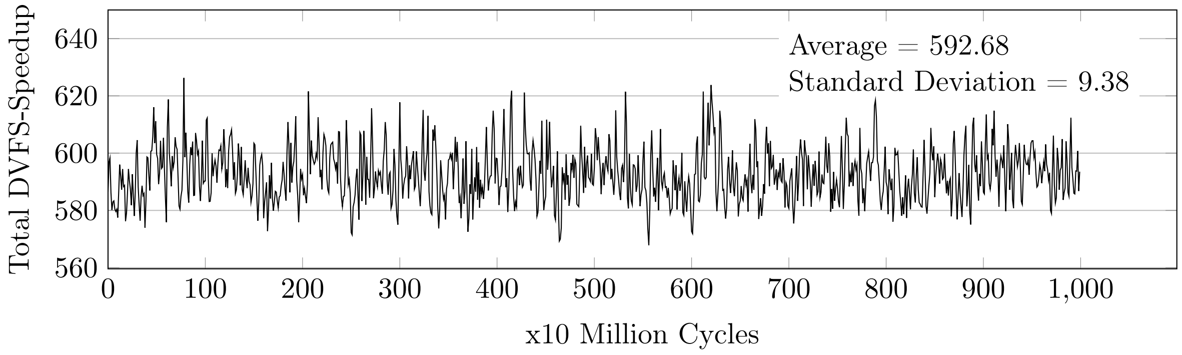


Figure 8.2.: Total DVFS-speedup over time when 1024 threads are using higher DVFS level.

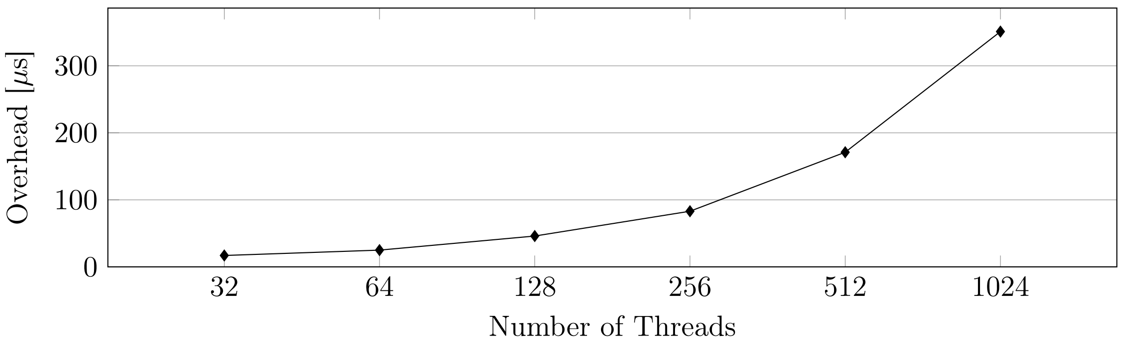


Figure 8.3.: Expected overhead for collecting and processing phase information of all tasks while running varisized workloads using a non-probabilistic greedy algorithm *SortedWS* [130].

Reason being that even though locally all tasks are going through different execution phases, there is no synchronization among their phases. While some tasks transition from low to high speedup phase at any given time, near-equal number of tasks perform reverse transition; resulting in stable global behavior. This behavior can be exploited by a probabilistic governor for many-cores to provide near-equal performance in comparison to non-probabilistic governors but with significantly lower computational and communication overheads.

8.0.1. Motivational Example

Our proof-of-concept cycle-accurate simulations predicts that even most light-weight of non-probabilistic governors will have unacceptable overheads when deployed in many-cores. *SortedWS* [130] is greedy non-probabilistic governor which collects phase-correlated power consumption information from all tasks at central core. The central core then sorts information and then utilize it to take power-budgeting decisions. Figure 8.3 shows how overhead to collect and process phase information at central core using *SortedWS* increases with increase in the number of tasks running on many-core. Figure 8.3 shows for many-core with 1024 threads, collecting and processing phase information requires 351 μs. For governor operating at default 10 ms scheduling granularity of current multi-core OS [33], this translates into an overhead of 3.51%. In context, default governors in contemporary multi-core OS operate with average overhead of only around 10 μs or 0.1% [133]. Governor introduced in this chapter aims to achieve a similar level of overhead for many-cores using probabilistic approach.

8.0.2. Novel Contributions

We introduce an alternative probabilistic governor called *Probabilistic Many-Core Governor (PMG)* for power budgeting under TDP on many-cores. Our alternative approach for power budgeting has potential to reduce associated scheduling overheads significantly. Furthermore, mathematical foundations of *PMG* also allow for concrete guarantees on TDP violations.

Operations of *PMG* is entirely different from a non-probabilistic governor. While non-probabilistic governor uses dynamic phase information obtained online to make decisions, *PMG* uses static probabilistic phase profiles collected offline to make similar decisions. Decisions made by non-probabilistic governor regularly change as tasks go through different phases. On the other hand, decisions made by *PMG* change only when task composition changes its constitution by arrival or departure of a task. A non-probabilistic governor is well-suited when the number of tasks is small whereas probabilistic governor works well when the number of tasks is large. In fact, due to the law of large numbers [134] results provided by *PMG* become more accurate as the number of tasks increases. Therefore, *PMG* is particularly well suited for the many-core paradigm. Contrarily, we should not use *PMG* when the number of tasks is small.

8.0.3. Limitations

Results guaranteed by *PMG* are also probabilistic. While non-probabilistic governor can always guarantee non-violation of TDP while extracting high performance, *PMG* only provides a high probability that many-core will operate similarly for any given population of tasks. It is important to note that for a sufficiently loaded many-core probability of TDP violation is always non-zero under *PMG* if it employs DVFS for boosting performance substantially. Hence, *PMG* is also not suitable for hard real-time or mission-critical systems.

8.1. Scheduler

We present details of *PMG* in this section. Chapter 3 describes common notations used to describe *PMG*. Tasks are assumed to be rigid. In this chapter, we assume that cores of many-core have only two DVFS levels namely *Low* and *High*. This assumption limits the ability of *PMG* to target energy-efficient execution on many-cores wherein multiple DVFS levels are available. We explore the use of multiple DVFS in conjunction with QoS tasks in Chapter 9.

PMG gives each task t_i a strategy S_{t_i}, which represents DVFS-speedup threshold. S represents combined strategic profile of all tasks determined by *PMG*. Task t_i chooses to operate at a *Low* or *High* DVFS level depending upon whether instantaneous DVFS-speedup is above or below strategy S_{t_i}, respectively. It is important to note that once strategy S_{t_i} is given to task t_i by *PMG*, there is no further communication between *PMG* and task t_i. Decision to boost performance using DVFS is taken by task t_i independently and locally based on expected

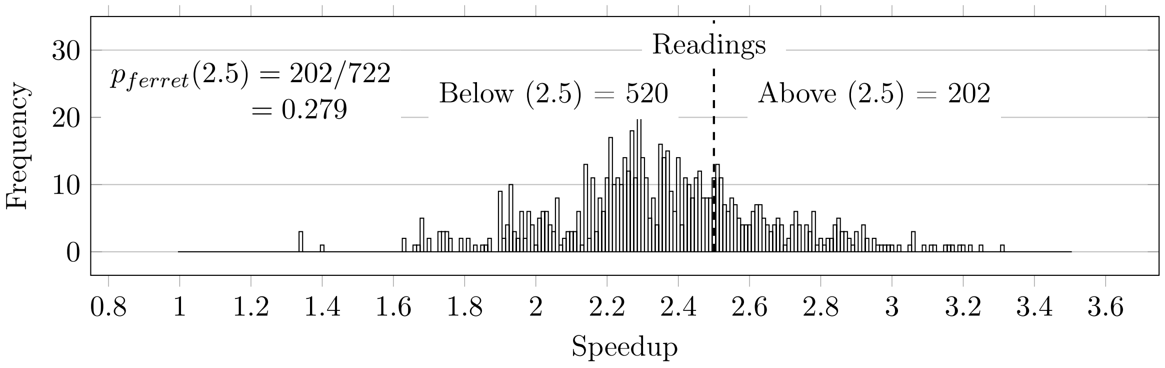


Figure 8.4.: Histogram of DVFS-speedup in single-threaded *ferret* along with calculation of $p_{ferret}(2.5)$.

DVFS-speedup. Expected DVFS-speedup boost can be calculated using locally available profiles or DVFS performance prediction models [135].

8.1.1. Probabilistic Performance Model

Given strategy S_{t_i} for task t_i, let $p_{t_i}(S_{t_i})$ be the probability that task t_i is using *High* DVFS level. Probability $p_{t_i}(S_{t_i}) \in [0, 1]$ represents the fraction of time task t_i under its execution exhibits DVFS-speedup higher than the value of strategy S_{t_i}. Figure 8.4 shows the histogram of DVFS-speedup corresponding to a single-threaded version of *ferret* and also shows a numerical example how we can calculate $p_{ferret}(2.5)$.

Probability $p_{t_i}(S_{t_i})$ is a monotonically non-increasing function of strategy S_{t_i} because as we increase strategy S_{t_i} lesser or equal fraction of task t_i would use *High* DVFS level. We obtain this probability data for discrete values of strategy S_{t_i} for each unique task and store it in a lookup table. We set granularity of DVFS-speedup data discretization at 0.01.

While executing in parallel with given strategy profile S, each task t_i acts as independent Bernoulli trial which uses *High* DVFS level with probability $p_{t_i}(S_{t_i})$ and uses *Low* DVFS level with probability $1 - p_{t_i}(S_{t_i})$. Therefore, our system exhibits Poisson binomial distribution and probability that $K \leq |T|$ tasks will be using *High* DVFS level simultaneously is given by following Probability Mass Function (PMF) [136].

$$Pr(K) = \sum_{A \in F_K} \prod_{x \in A} p_{t_x}(S_{t_x}) \prod_{y \in A^C} (1 - p_{t_y}(S_{t_y})) \tag{8.1}$$

where F_K is set of combinations of K tasks selected from set of T tasks. A represents one such combination whereas A^C represents the complement combination of combination A.

We choose the maximum number of tasks in scheduling epoch that can accelerate without violation of TDP as metric for optimization. This metric has a positive correlation with standard non-probabilistic performance metrics like throughput used in Chapter 4. Standard performance metrics [32] like throughput and response time are not suitable metrics to target directly for probabilistic governor. During execution, we want the number of tasks that boosted themselves using DVFS in scheduling epochs under TDP to be maximum. Therefore, we want to optimize Equation (8.1) to peak at highest value for K feasible under TDP.

8.1.2. Binomial Simplification

Even obtaining PMF distribution for any given strategy profile S using Equation (8.1) has $O(|T|!)$ complexity, making direct optimization computationally infeasible when the number of tasks $|T| >> 1$. To make the problem tractable, we propose simplification that converts

Poisson binomial distribution into binomial distribution, which is well-studied and much more mathematically tractable discrete probability distribution.

By design instead of choosing unique strategy S_{t_i} and thereby resulting unique probability $p_{t_i}(S_{t_i})$ for every task t_i, *PMG* instead selects single global *High* DVFS level probability p. *PMG* then gives each task t_i a strategy S_{t_i} such that $p_{t_i}(S_{t_i}) \approxeq p$. This assignment also introduces fairness into many-core because each task has now the equal probability of using DVFS to boost its performance. On the other hand, when operating with QoS tasks, this simplification can result in wastage of energy as heterogeneity in speedup behaviors then remains unexploited. Therefore, we do not use this simplification in Chapter 9 when we work with QoS tasks.

Under above simplification, each task executing in parallel acts as independent Bernoulli trial which uses *High* DVFS level with probability p and uses *Low* DVFS level with probability $1-p$. Our system now exhibits binomial distribution and the following PMF gives the probability that $K \leq |T|$ tasks will be using *High* DVFS level together.

$$Pr(K) = \binom{|T|}{K} p^K (1-p)^{|T|-K} \tag{8.2}$$

We aim to maximize Equation (8.2) concerning given K using probability p. Since natural logarithm is a positive function, maximizing log of Equation (8.2) is same as maximizing equation itself. By taking log of Equation (8.2) we get

$$\log(Pr(K)) = \log \binom{|T|}{K} + K \log(p) + (|T| - K) \log(1-p)$$

Derivating with respect to probability p and equating to zero we get

$$\frac{K}{p} - \frac{(|T| - K)}{1-p} = 0 \implies p = \frac{K}{|T|} \tag{8.3}$$

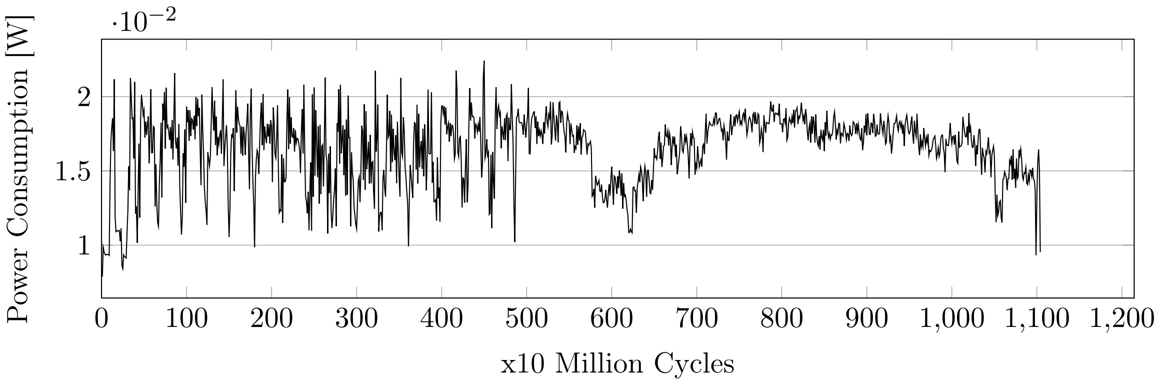
Therefore, if we know target K, then we can use Equation (8.3) to find probability p which maximizes the probability of many-core boosting K tasks using *High* DVFS level in given scheduling epoch. *PMG* then determines strategy profile S for target probability p using the profiles. The values of Mean (μ) and standard deviation (σ) of targeted binomial distribution are given by following equations.

$$\mu = \sum_{K=1}^{|T|} K Pr(K) = |T|p \tag{8.4}$$

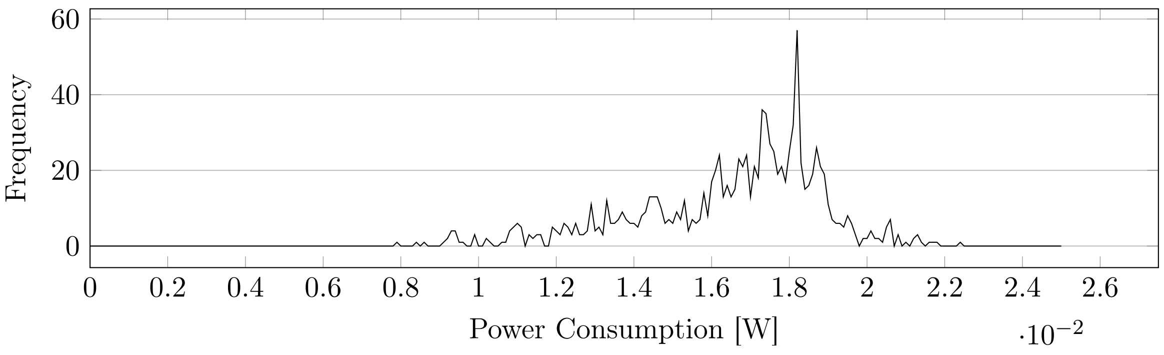
$$\sigma = \sqrt{\sum_{K=1}^{|T|} K^2 Pr(K) - \mu^2} = \sqrt{|T|p(1-p)} \tag{8.5}$$

8.1.3. Probabilistic Power Consumption Model

The power consumption of task similar to its DVFS-speedup also varies over time. Figure 8.5a shows how power consumption of *ferret* varies over time when running in *Low* DVFS level. Figure 8.5b shows the corresponding distribution of *ferret*'s power consumption in *Low* DVFS level obtained by transformation from discrete time domain to discrete frequency domain. Non-uniform real-world distribution as shown in Figure 8.5b is computationally infeasible to aggregate. Hence, we need to make some approximations to get total power consumption.



(a) Power Consumption in Time Domain



(b) Power Consumption Real Distribution

Figure 8.5.: Probabilistic power consumption of *ferret*.

We now attempt to predict probabilistic power consumption of the many-core for a given *High* DVFS level probability p. For each task t_i, let $W_{t_i}^L$ and $W_{t_i}^H$ be its expected power consumption in *Low* and *High* DVFS level, respectively. This power includes both the static and dynamic power consumption of a task at those levels. Let W^L and W^H be the average power consumption of all tasks in many-core at *Low* and *High* DVFS level, respectively. We can make a quick rough estimation of their values at run-time using the following equations by assuming power consumption of individual tasks to be additive.

$$W^L = \frac{\sum_{i=1}^{|T|} W_{t_i}^L}{|T|} \qquad W^H = \frac{\sum_{i=1}^{|T|} W_{t_i}^H}{|T|} \tag{8.6}$$

Since the number of tasks that boost up using DVFS follows binomial distribution, the total power consumption of many-core will also thereby exhibit a normal distribution [137]. Let $W(x)$ represent the distribution of power consumption given by the following equation.

$$W(x) = \frac{1}{\sqrt{2(\sigma_W)^2 \pi}} e^{-\frac{(x-\mu_W)^2}{2(\sigma_W)^2}} \tag{8.7}$$

where μ_W and σ_W represent the mean and standard deviation of the normal distribution.

To obtain total power consumption distribution, we reason that at for any given value of p, Np number of tasks are expected to use *High* DVFS level consuming W^H power each. Similarly, we can expect $N(1-p)$ number of tasks to use *Low* DVFS level consuming W^L power each. Therefore, following relation between K and x holds.

$$\forall_{K \in [0,|T|]} \exists x = W^H K + W^L(|T| - K) \mid Pr(K) = W(x) \tag{8.8}$$

We can derive the equations for μ_W as followed.

$$\begin{aligned}
\mu_W &= \sum_{K=1}^{|T|} \left((W^H - W^L)K + W^L|T|\right) Pr(K) \quad [\because \text{Equation (8.8)}] \\
&= \sum_{K=1}^{|T|} (W^H - W^L)KPr(K) + \sum_{K=1}^{|T|} W^L|T|Pr(K) \\
&= (W^H - W^L)\mu + W^L|T| \quad [\because \text{Equation (8.4) and} \sum_{K=1}^{|T|} Pr(K) = 1] \qquad (8.9)
\end{aligned}$$

Using Equation(8.8), we can derive the equations for σ_W as followed.

$$\begin{aligned}
\sigma_W &= \sqrt{\sum_{K=1}^{|T|} ((W^H - W^L)K + W^L|T|)^2 \, Pr(K) - \mu_W{}^2} \\
&= \sqrt{\sum_{K=1}^{|T|} \Big((W^H - W^L)^2K^2 + W^{L^2}|T|^2 + 2(W^H - W^L)KW^L|T|\Big) Pr(K) - \mu_W{}^2} \\
&= \sqrt{\textstyle\sum_{K=1}^{|T|} (W^H - W^L)^2K^2Pr(K) + \sum_{K=1}^{|T|} W^{L^2}|T|^2Pr(K) + \sum_{K=1}^{|T|} 2(W^H - W^L)KW^L|T|Pr(K) - \mu_W{}^2} \\
&= \sqrt{(W^H - W^L)^2(\sigma^+\mu^2) + W^{L^2}|T|^2 + 2(W^H - W^L)|T|W_L\mu - \mu_W{}^2} \quad [\because \text{Equations (8.4) and (8.5)}] \\
&= \sqrt{(W^H - W^L)^2(\sigma^+\mu^2) + W^{L^2}|T|^2 + 2(W^H - W^L)|T|W_L\mu - ((W^H - W^L)\mu + W^L|T|)^2} \quad [\because \text{Equation (8.9)}] \\
&= \sigma(W^H - W^L) \qquad (8.10)
\end{aligned}$$

The assumption of normality in distribution is a strong assumption but is necessary to provide formal mathematical analysis. Fortunately, the error introduced by this assumption becomes less severe as the number of independent tasks on the many-core increases due to the central limit theorem [134]. Central limit theorem applied in our context states that distribution of the arithmetic sum of power consumption of independent tasks approaches normal distribution as the number of tasks approaches infinity irrespective of power consumption distribution of individual tasks. Therefore, feasibility and accuracy of *PMG* increases with the size of the problem making it especially suitable for many-cores. It is important to note that if tasks are not executing independently or tasks in workload are not diverse enough then many-core would not exhibit any normal distribution. For many-core with interdependent tasks, the covariance between them is not insignificant which also needs to be considered.

8.1.4. Probabilistic TDP Model

The probability of TDP violations under *PMG* when many-core is not severely underloaded can be non-zero if DVFS has to be used aggressively. Therefore, it is essential to quantize risk many-core is taking for given *High* DVFS level probability p and given a set of tasks T. Let $\hat{W}$ symbolically represent TDP of many-core. The probability that many-core will stay within TDP $\hat{W}$ is given using cumulative distribution function $F(\hat{W})$ of the normal distribution.

$$F(\hat{W}) = \int_0^{\hat{W}} W(x)dx = \int_0^{\hat{W}} \frac{1}{\sqrt{2(\sigma_W)^2\pi}} e^{-\frac{(x-\mu_W)^2}{2(\sigma_W)^2}} dx \qquad (8.11)$$

Algorithm 2 Proposed probabilistic power budgeting technique named *PMG*.

```
Input: $|T|, \hat{W}, \hat{Q}$;
Output: $S$;
1: $\forall i \in |T|$ read profiled (or estimate) $W^L_{t_i}$ and $W^H_{t_i}$;
2: Calculate $W^L$ and $W^H$ using Equations (8.6)
3: for $K = 1$ to $|T|$ do
4:     Calculate $p$ for a given $K$ using Equation (8.3);
5:     Calculate $W(x)$ using Equation (8.7);
6:     Calculate $F(\hat{W})$ using Equation (8.11);
7:     Calculate $Q(\hat{W})$ using Equation (8.12);
8:     if $Q(\hat{W}) \geq \hat{Q}$ then
9:         break;
10:    end if
11: end for
12: Calculate $p$ for $K - 1$
13: $\forall i \in |T|$ set $S_{t_i}$ such that $p_{t_i}(S_{t_i}) \approxeq p$;
14: return $S$;
```

No closed form solution exists for calculation of $F(\hat{W})$. However, it can be numerically approximated using Chebyshev fitting [138] in constant time. The probability that TDP is violated is given by Q-function symbolically represented by $Q\ (\hat{W})$.

$$Q\ (\hat{W}) = 1 - F(\hat{W}) = 1 - \int_0^{\hat{W}} \frac{1}{\sqrt{2(\sigma_W)^2\pi}} e^{-\frac{(x-\mu_W)}{2(\sigma_W)^2}} dx \tag{8.12}$$

8.1.5. Power Budgeting

Based on the mathematical foundations laid above, we now present power budgeting algorithm used by *PMG*. Unlike a non-probabilistic governor, *PMG* cannot guarantee non-violation of TDP. However, it provides formal guarantees on the risk of TDP violations. Let TDP threshold $\hat{Q}$ represent the fraction of TDP violating scheduling epochs, many-core designer is willing to tolerate. Accordingly, the expected number of tasks to boost using DVFS K needs to be determined based on the current set of tasks T. Value of K should be as high as possible.

For a given set of tasks T, $K \propto p$ is based on Equation (8.3). Therefore, maximizing *High* DVFS level probability p is same as maximizing K. It is a common observation that all tasks consume less or equal power while using *Low* DVFS level than when using *High* DVFS level. Hence, for any given set of tasks T we know $W^H \geq W^L$. Based on this knowledge and Equation (8.9), we can state $\mu_W \propto p$.,

Being normal distribution, $W(x)$ is unimodal with the peak around μ_W. Since TDP $\hat{W}$ is immutable, increase in μ_W will push more cumulative distribution beyond $\hat{W}$. Therefore, we can conclude $Q\ (\hat{W})$ is monotonically non-decreasing with μ_W. Based on transitivity of above proportionality argumentation, the risk of TDP violation $Q\ (\hat{W})$ is monotonically non-decreasing with task boost target K.

Determination of optimum value of K by *PMG* is thereby simplified to search in a discretized domain $K \in [0, N]$ such that $Q\ (\hat{W}) \approxeq \hat{Q}$. Algorithm 2 summarizes technique used in *PMG* using simple linear search. We can also use binary search for improved efficiency as $Q\ (\hat{W})$ is inherently sorted with the value of K. Our simple approach is sufficient to provide reasonable performance given its carefully designed underlying mathematical formulation.

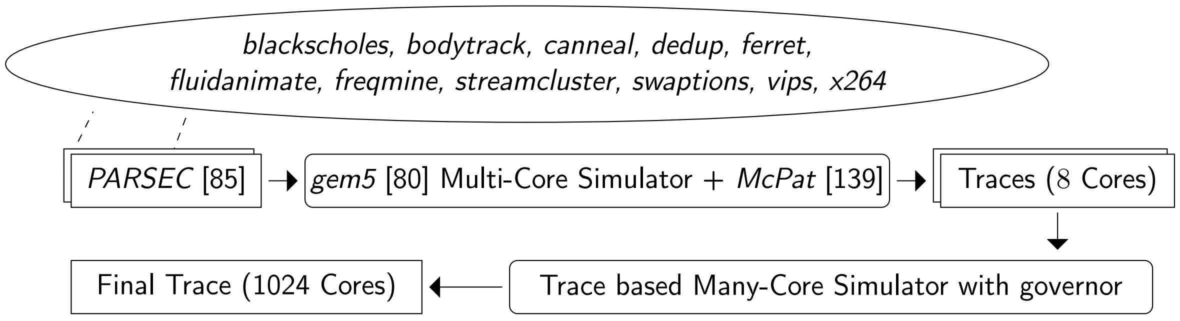

blackscholes, bodytrack, canneal, dedup, ferret,
fluidanimate, freqmine, streamcluster, swaptions, vips, x264
PARSEC [85]
gem5 [80] Multi-Core Simulator + McPat [139]
Traces (8 Cores)
Trace based Many-Core Simulator with governor
Final Trace (1024 Cores)

Figure 8.6.: Experimental Setup for *PMG*.

8.1.6. Complexity

Since *PMG* under any step requires not more than one iteration over tasks, it has linear time computational complexity of $O(|T|)$. Use of centralized lookup tables leads to communication complexity of $O(1)$ for *PMG*. In comparison, probabilistic greedy governor *SortedWS* [130] has computation complexity of $O(|T| \lg |T|)$ and communication complexity of $O(|T|)$. Both techniques will have $O(|T|)$ space complexity. Furthermore, the greedy algorithm needs to be invoked in every scheduling epoch to operate. On the other hand, *PMG* executes when task arrives or leaves many-core changing task composition.

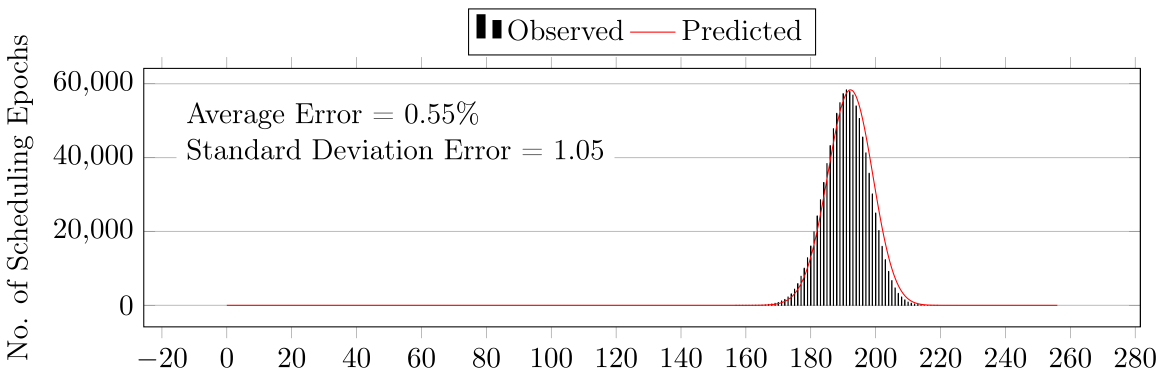
8.2. Experimental Evaluations

8.2.1. Experimental Setup

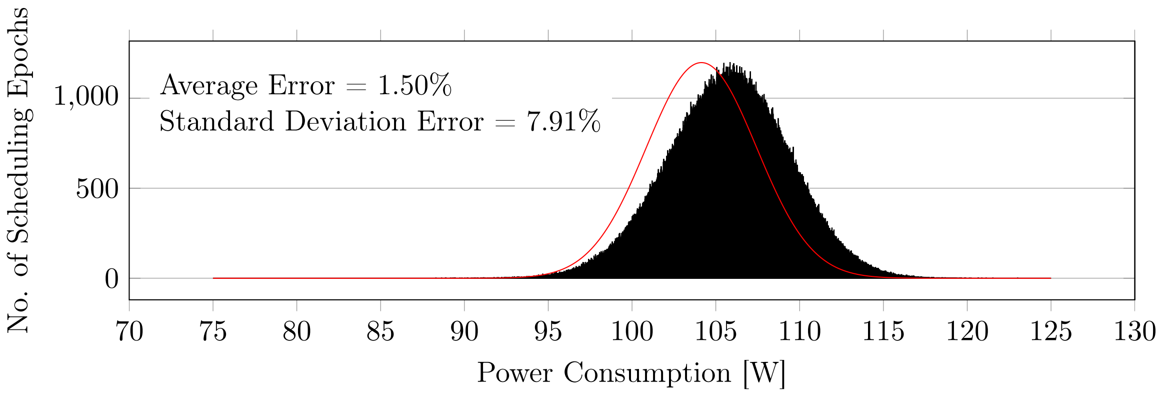
We again fallback to trace-based simulator similar to the one used in Chapter 4. Even interval-simulator like *Sniper* used in Chapter 6 is too slow in simulating thousand core system required to demonstrate the benefits of a probabilistic governor. We use a two-stage simulator for empirical evaluation of *PMG* as shown in Figure 8.6. In stage one, we use *gem5* [80] cycle-accurate simulator bridged with *McPat* [139] power simulator. Simulation-time constraints limit cycle-accurate simulations to maximum eight cores. Each core uses *Alpha* ISA and holds 16 KB L1 data and instruction cache, along with 32 KB private L2 cache. Cores can run at two DVFS frequencies 1 GHz and 3 GHz representing *Low* and *High* DVFS level, respectively. Unused cores are power-gated. We believe our 22-nm low-power in-order cores with small caches are most representative cores for real-world thousand core many-cores.

We pipe cycle-accurate isolated execution traces of tasks with up to eight cores allocated from stage one into a trace-based simulator in stage two. Stage two simulator then combines traces for a final many-core trace with up to thousand cores assuming 2D mesh NoC between cores. NoC has a concentration of 1 router per-core and operates at 1 GHz using 256-bits flit. NoC links have a bandwidth of 1 flit per cycle and latency of 4 cycles per hop. Stage two simulator also implements governors operating at a granularity of 10 ms. For each experiment, we simulate around three hours of a closed system. We initiate each task with a random instructional skew to simulate independent task execution in the closed system.

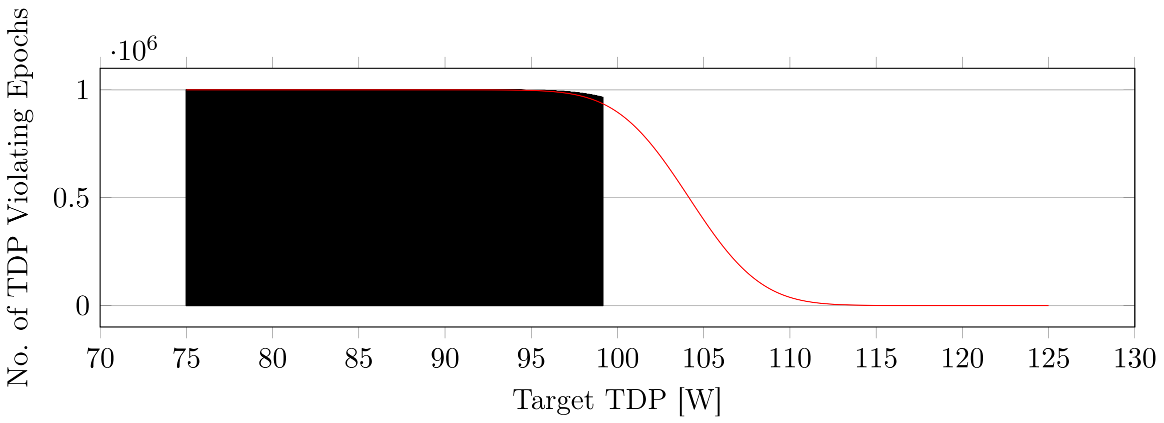
For software, we use eleven multi-threaded tasks as enumerated in Figure 8.6 from *PARSEC* suite [85]. We form multi-program workloads from the random composition of these tasks with each task randomly spawning between one to eight threads. We run tasks in Full System (FS) mode of *gem5* with *sim-small* input. Out of thirteen *PARSEC* tasks, we did not use only two tasks *facesim* and *raytrace* due to lack of *sim-small* input.



(a) Task Boost Distribution



(b) Power Consumption Distribution



(c) TDP Violation (Q-Function) Distribution

Figure 8.7.: Observed and predicted distributions for a 256-task (1024-thread) workload with boost target of 192 tasks ($p = .75$) on a 1024-core many-core.

8.2.2. Comparative Baseline

We compare *PMG* against a scalable non-probabilistic greedy scheduler called *SortedWS* [130] designed for many-cores. *SortedWS* allocates power budget to tasks in decreasing order of instantaneous speedup without TDP violation with the goal of maximizing throughput. *SortedWS* uses the aggregate of DVFS-speedups of all tasks $\alpha(S)$ as the measure of throughput.

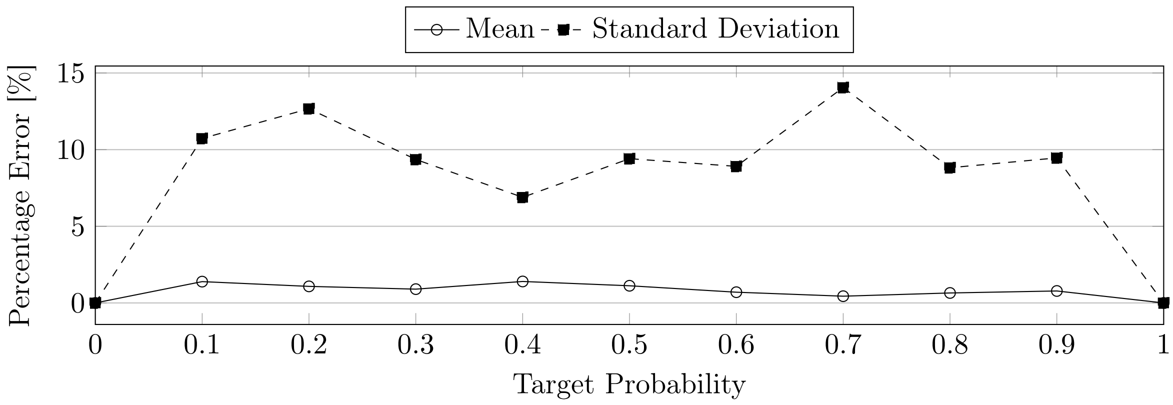


Figure 8.8.: Error in predicting mean and standard deviation for total power consumption distribution at different probability targets for a 256-task (1024-thread) workload.

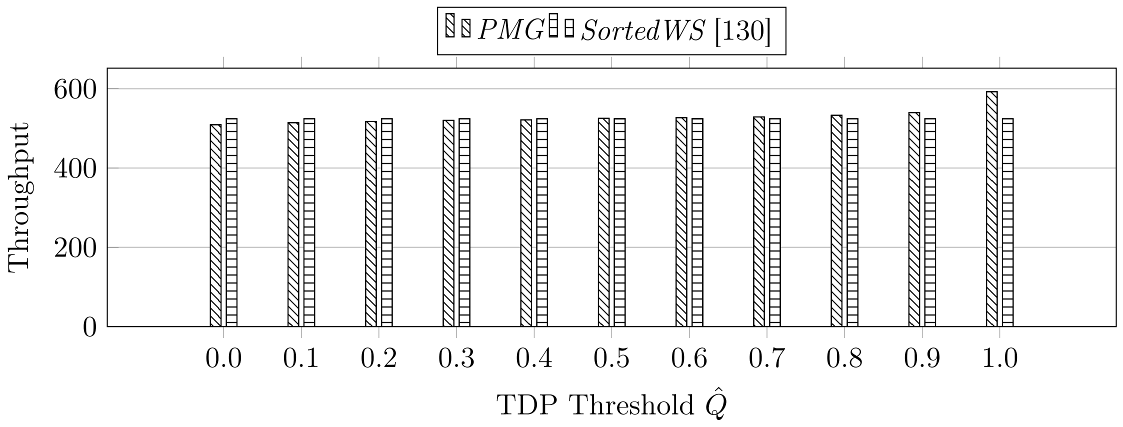


Figure 8.9.: Throughput comparison between *PMG* and *SortedWS* [130] for different values of TDP threshold $\hat{Q}$ with TDP $\hat{W}$ set at 100 W while executing a 256-task (1024-thread) workload.

8.2.3. Probabilistic Modeling Accuracy

We ran a 256-task (1024-thread) workload on 1024-core many-core with a target of 192 tasks to boost using DVFS in given scheduling epoch. Figure 8.7a plots the predicted and observed distribution of DVFS boosted tasks. Figure 8.7b shows corresponding predicted and observed total system power consumption distribution. Results show that we can predict the distribution of both DVFS-speedup and power consumption with high accuracy. Figure 8.7c plots the predicted and observed distribution of TDP violating scheduling epochs with different targeted TDP values. Results show that we can also predict TDP violation distribution.

Figure 8.8 notes accuracy in predicting mean and standard deviation for total power consumption distribution at different probability targets for a 256-task (1024-thread) workload. The figure shows that accuracy is higher at some probability targets than others. This observation attributes to the fact that due to discrete sampling profiles of tasks, we cannot achieve all probability targets with equal precision for all tasks. The task with higher variations in its observed speedups allows for a more precise setting of speedup probability and thereby provides better results when used with *PMG*. The average error in predicting mean and standard deviation of total power consumption is 0.76% and 8.20%, respectively.

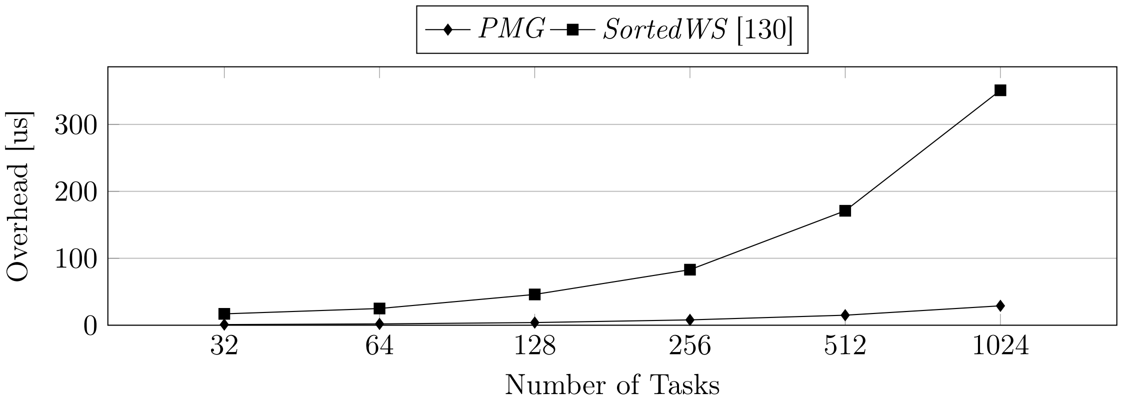


Figure 8.10.: Expected overhead for power budgeting varisized workloads with *PMG*.

8.2.4. Performance Comparison

Figure 8.9 compares throughput for a 256-task (1024-thread) workload under *PMG* and *SortedWS* [130] governor for different values of TDP threshold $\hat{Q}$ with TDP $\hat{W}$ set at 100 W. Average total system speedup per scheduling epoch is selected as throughput metric. Since *SortedWS* does not allow for any TDP violation, it ignores threshold $\hat{Q}$ and produces only one fixed result in Figure 8.9. On the other hand, *PMG* allows for an increase in throughput with an increase in the value of threshold $\hat{Q}$. When we set threshold $\hat{Q}$ to its lowest value zero, there is practically no risk of TDP violation. But even at this setting, it also has 2.85% lower performance than *SortedWS*. When we set the threshold $\hat{Q}$ to one, *PMG* completely ignores TDP, and many-core runs all tasks using *High* DVFS level resulting in maximum system performance. Intermediate values of threshold $\hat{Q}$ allows for a trade-off between performance and TDP violation risk.

8.2.5. Scalability

The motivation behind using probabilistic governor is its ability to scale up with the increase in problem size. We run *PMG* and *SortedWS* cycle-accurately on *gem5* with representative input and report its worst case problem-solving time in Figure 8.10 with workloads of different sizes. Results show that *PMG* can solve the problem of allocating power budgets to 1024 tasks many-core in only 29 μs. For governor operating at default 10 ms scheduling granularity of current multi-core OS [33], this translates into an overhead of 0.29%. In context, non-probabilistic governors in multi-core OS operate with average overhead of only around 10 us or 0.1% [133]. In comparison, *SortedWS* takes 351 us to perform power budgeting for same number of tasks. Therefore, *PMG* provides 97.13% (or 12x) reduction in overhead compared to *SortedWS*.

8.3. Summary

In this work, we proposed governor called *PMG* based on a probabilistic technique for power budgeting on many-cores. *PMG* provides superior scalability in comparison to existing non-probabilistic power budgeting techniques while providing mathematical guarantees on the risk of TDP violations. Proof-of-concept cycle-accurate simulations show that *PMG* results in 97.13% less overhead than non-probabilistic greedy governor on thousand-core many-core. This reduction in overhead comes at 6% loss of performance.

The probabilistic governor presented in this chapter cannot operate with QoS tasks. It also cannot handle multiple discrete DVFS levels due to its fundamental mathematical constructs. We eliminate both these drawbacks in probabilistic governor presented in next chapter.

9. Probabilistic Many-Core Task Governor for QoS Tasks

We present power budgeting scheduler (governor) for many-core with QoS tasks in this chapter.[1] Proposed governor builds upon the governor introduced in Chapter 8. A many-core is capable of parallel execution of hundreds of multi-threaded tasks. Many-core is expected to execute a task at user-defined target QoS when deployed in QoS-aware system. We choose to measure QoS of a task with its IPS, which has a positive correlation with other equivalent QoS metrics [140].

Figure 9.1 denotes changes in IPS of a single-threaded *ferret* on given frequency. The figure shows that frequency is incapable of keeping task's QoS consistent. Many-core can abate this problem with the help of DVFS. As previously also discussed, DVFS allows a change in frequency of cores executing the task to deliver task variable amount of processing. Governor, therefore, can use DVFS to keep QoS of task close to its target QoS as shown in Figure 9.2. Since the number of DVFS frequencies is limited, not every target QoS can be precisely attained. The power consumption of core in general increases with increase in its frequency. Therefore, task's power consumption shows strong positive correlation with its underlying core frequencies. This correlation results in task consuming a variable amount of power over its lifetime as also shown in Figure 9.2. Achieving QoS more than task's target QoS task is not detrimental to the task but wastes power. Governor, therefore, should avoid it.

Limited heat dissipation capacity of many-core forces it to operate under power budget called TDP [141]. Continuous operation beyond TDP leads to a thermal emergency on many-core wherein hardware-triggered Dynamic Thermal Management (DTM) reduces all core frequencies to a minimum. Frequent triggering of DTM leads to significant deterioration in performance. When executing in parallel, individual tasks executing within TDP can violate TDP in totality. Therefore, it is mandated to carefully budget TDP between tasks while keeping their QoS requirements under consideration. OS sub-routine called governor is tasked to manage TDP.

The problem of QoS-aware power budgeting for multi/many-cores has been studied only from non-probabilistic perspective [119]. Note, for the problem of power-budgeting in many-core term stochastic and probabilistic are interchangeable. Non-probabilistic controls – centralized [142] or distributed [143] – have been employed by governors for QoS-aware power budgeting in many-cores. Non-probabilistic governor involves monitoring of tasks for their QoS, power consumption and other similar parameters to make power budgeting decisions. A non-probabilistic governor can operate directly on feedback from power sensors often available at core, cluster or chip level granularity depending upon hardware [142]. Feedback can be processed to make power budgeting decisions using techniques like online learning [143] or greedy-search [142]. Decisions dictate to each task frequency it must use.

Many works use Proportional Integrate Derivative (PID) controller as a non-probabilistic control in their QoS-aware governors [125]. Authors in [144] and [125] employed PID controller for power budgeting in homogeneous and heterogeneous multi-cores, respectively. Recently, PID controller has also been applied to perform power budgeting in many-cores [141]. Gains in PID controller of non-probabilistic governor need to be appropriately tuned for it to work correctly. Since gains tuned for one workload may not hold for another workload, it makes governor based on PID controller challenging to use in practice. We also found PID controller

[1]The work presented in this chapter was originally published in [11] ©2018 *ACM.*

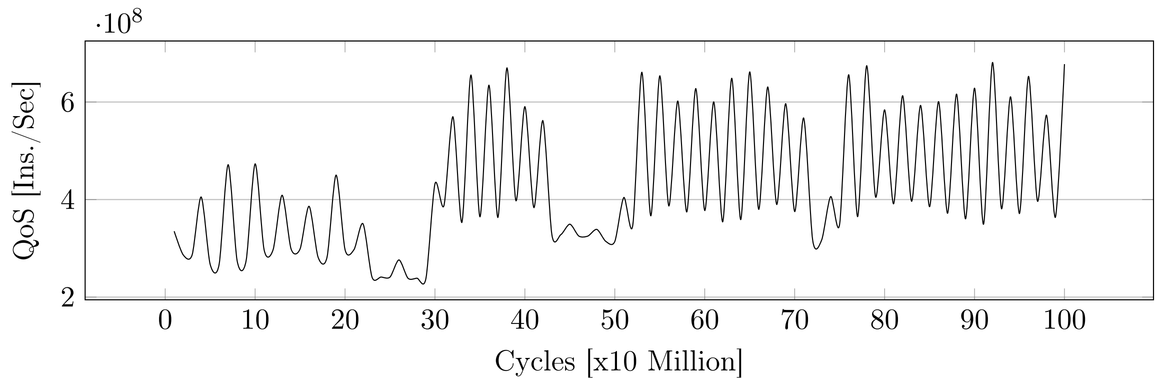


Figure 9.1.: Execution profile of single-threaded *ferret* showing variation in its QoS during execution over a single fixed frequency.

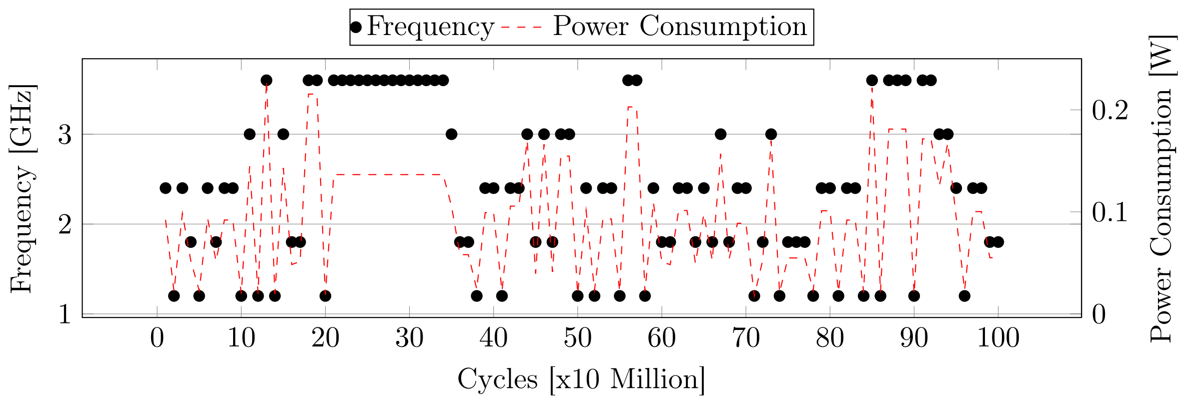


Figure 9.2.: Execution profile of single-threaded *ferret* showing need for different DVFS level to maintain a consistent QoS; resulting in variable power consumption.

to be unstable on many-core as underlying system dynamics have too high entropy for it to manage. The probabilistic governor, on other hand, requires no such fine tuning and does not suffer from any instability. Probabilistic governor, on the flip side, requires no such fine-tuning.

We developed a probabilistic governor for many-cores with the goal of maximizing speedup called *PMG* in Chapter 8. *PMG* due to its fundamental mathematical constructs cannot be applied to QoS tasks and can operate only with two frequency levels – *High* and *Low*. This chapter addresses both these shortcomings by introducing an extension to *PMG* called *Q-PMG*.

As the number of tasks executing on the many-core increases, non-probabilistic governor struggles to keep up due to increased computational overhead and thereby does not scale up. Furthermore, this constant bidirectional communication with hundreds of tasks on many-core involves overhead, which increases with increase in the size of many-core. In this work, we also argue that this constant communication can be replaced with far more infrequent intermittent communications without any significant performance loss using an alternative probabilistic control in the power-budgeting governor.

The probabilistic governor in contrast to non-probabilistic governor centrally optimizes the distribution of many-core's total power consumption over time by manipulating executing tasks' target QoS. Under *Q-PMG*, the task decides on which core frequency to use locally based on its current and set target QoS. Since probabilistic governor operates on distributions, it is required to change QoS of executing tasks only when some task arrives or leaves many-core.

Decisions of probabilistic governor determining target QoS of tasks also has lower computational overhead than similar decisions of non-probabilistic governor directly determining tasks

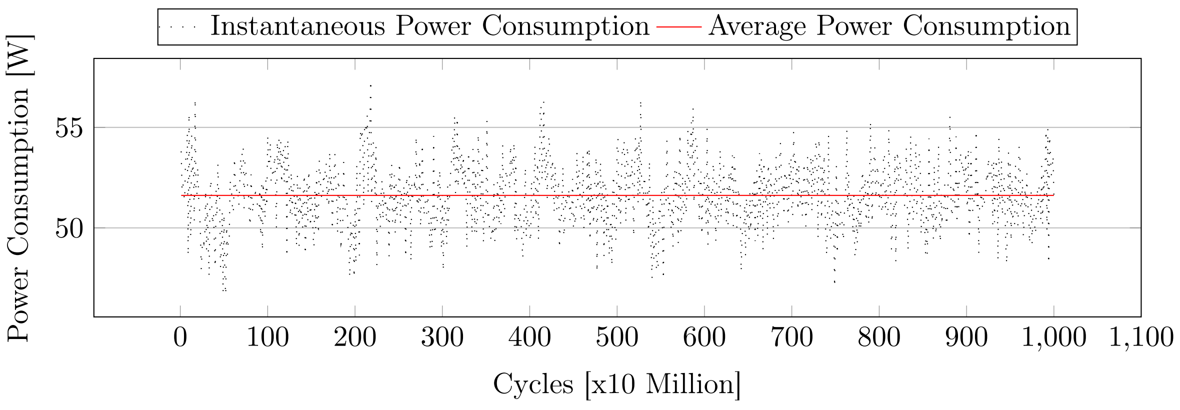


Figure 9.3.: Total power consumption of a many-core when executing a 1024-thread workload.

underlying core frequencies. Since tasks perform DVFS independent of the probabilistic governor, DVFS could be performed at fine granularity to save more power without additional governor-induced scheduling overheads.

The probabilistic governor works on the observation that power consumption of many-core when executing hundreds of independent tasks in parallel is quite stable. Figure 9.3 shows the instantaneous total power consumption of many-core when running 1024-thread workload comprising of 256 independent tasks each with its own QoS and performing independent DVFS stays very close to average total power consumption. We can attribute this observation to the fact that even though power consumption of individual task to maintain its QoS can vary over time; it does not correlate with power consumptions of other tasks. At any given time, some of tasks transit from high-power consumption phase to low-power consumption phase and vice versa without any synchronization. This lack of synchronization results in predictable total power consumption behavior that can be optimized.

9.0.1. Novel Contributions

We propose first probabilistic DVFS-based QoS-aware power budgeting governor for many-cores called *Q-PMG*. *Q-PMG* has computational complexity $O\ (\ln n)$ factor less than non-probabilistic governor while providing equivalent performance. Therefore, *Q-PMG* can potentially scale up much better with the increase in the number of cores in many-cores.

9.1. Scheduler

We present details of *Q-PMG* in this section. Chapter 3 describes common notations used to describe *Q-PMG*. Tasks are assumed to be *rigid* similar to Chapter 8. Let $\hat{\rho}_{t_i}$ be target IPS (QoS) for task t_i measured in Millions of Instruction per Second (MIPS). DVFS is performed by tasks locally at the granularity of scheduling epoch.

9.1.1. Probabilistic DVFS Model

Let $p(t_i, f_k, \hat{\rho}_{t_i})$ represent a probability that without any power budget constraints task t_i is using frequency f_k when its QoS target is $\hat{\rho}_{t_i}$. Mathematically probability $p(t_i, f_k, \hat{\rho}_{t_i})$ represents the fraction of total execution time spent by task t_i in frequency f_k to achieve QoS $\hat{\rho}_{t_i}$ and can be obtained using profiles. Figure 9.4 shows an exemplary calculation of probability $p(t_i, f_k, \hat{\rho}_{t_i})$. Probability is also dependent upon input.

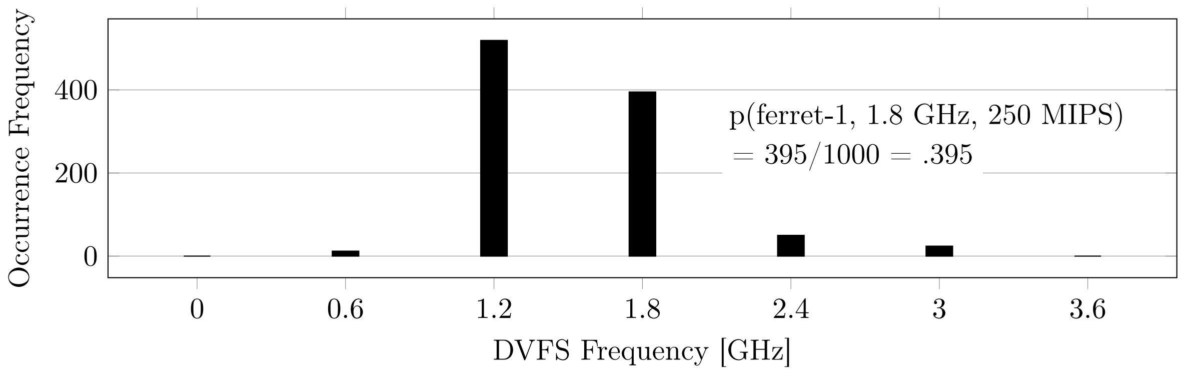


Figure 9.4.: Histogram of DVFS frequency used in single-threaded *ferret* for a given QoS (250 MIPS) along with a sample calculation of probability for a frequency.

Task t_i act as independent Bernoulli trial that uses frequency f_k with probability $p(t_i, f_k, \hat{\rho}_{t_i})$ and uses frequencies other than frequency f_k with probability $1 - p(t_i, f_k, \hat{\rho}_{t_i})$. Since tasks in many-core have different probabilities of using frequency f_k, total usage of frequency f_k shows Poisson binomial distribution. Let μ_{f_k} and σ_{f_k} be mean and standard deviation of Poisson binomial distribution of usage of frequency f_k, respectively.

$$\mu_{f_k} = \sum_{i=1}^{|T|} p(t_i, f_k, \hat{\rho}_{t_i}) \tag{9.1}$$

$$\sigma_{f_k} = \sqrt{\sum_{i=1}^{|T|} (1 - p(t_i, f_k, \hat{\rho}_{t_i}))p(t_i, f_k, \hat{\rho}_{t_i})} \tag{9.2}$$

PMF $Pr_{f_k}(K)$ gives the probability that $K \leq |T|$ tasks would be using frequency f_k [136].

$$Pr_{f_k}(K) = \sum_{A \in F_K} \prod_{t_x \in A} p(t_x, f_k, \hat{\rho}_{t_x}) \prod_{t_y \in A^C} (1 - p(t_y, f_k, \hat{\rho}_{t_y}))$$

where F_K is a set of all combinations of K tasks selected from the set of T tasks. Set A^C is a complement set of A.

The complexity of obtaining PMF $Pr_{f_k}(K)$ has a factorial complexity of $O(|T|!)$. Hence, it is infeasible to directly obtain PMF $Pr_{f_k}(K)$ at runtime when $|T| >> 1$. We handled this complexity in Chapter 8 by forcing all the tasks to use a given frequency with the same probability, thereby, simplifying Poisson binomial distribution into a more mathematically tractable binomial distribution. We cannot apply the same approach to QoS tasks since all of them cannot be forced to conform to same behavior for the sake of simplicity.

We overcome PMF complexity problem in this chapter with the help of central limit theorem [134], which applied here states that PMF Pr_{f_k} will approximately exhibit normal distribution if many-core meets the following two conditions. The first condition is all tasks in many-core should run independent of each other and hence their usage of frequency f_k should exhibit no correlation. Condition holds very well on many-core designs such as *InvasIC* computing [21] described in Section 2.6.1 which support predictable execution [145] where shared-resource contentions do not manifest. This condition is not mandatory for threads of given task, which are inherently correlated.

The second condition is there is large number of tasks executing in parallel that use frequency f_k substantially, which holds on many-core. Therefore, *Q-PMG* becomes more accurate with the

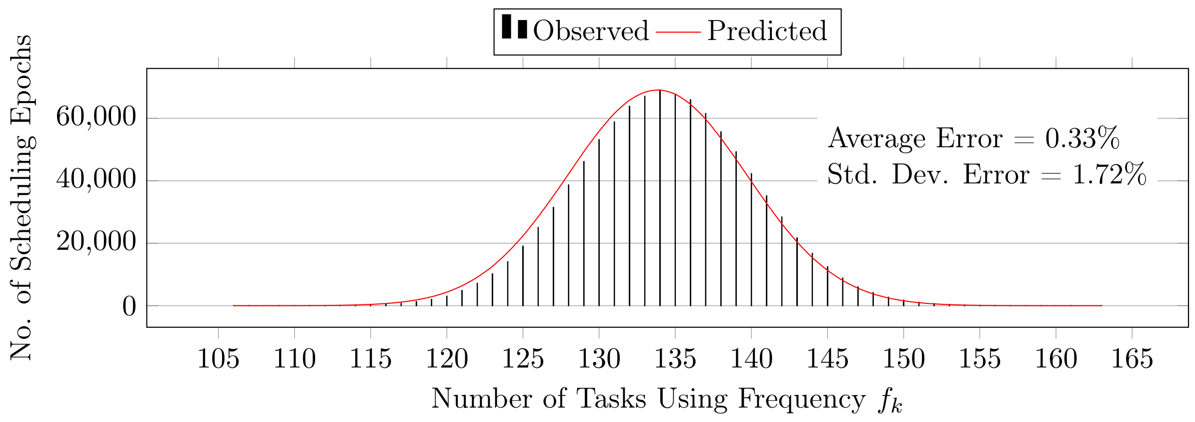


Figure 9.5.: Observed and predicted distribution for usage by tasks of frequency $f_k = 1.8\ GHz$ for a 256-task (1024-thread) workload on a many-core.

increase in the size of many-core and supports the argument for moving from non-probabilistic to probabilistic governors for power budgeting as we transit from multi-cores to many-cores.

We assume that we can approximate discrete PMF $Pr_{f_k}(K)$ by continuous Probability Density Function (PDF) of a normal distribution with mean μ_{f_k} and standard deviation σ_{f_k}.

$$Pr_{f_k}(K) = \frac{1}{\sqrt{2(\sigma_{f_k})^2\pi}} e^{-\frac{(K-\mu_{f_k})^2}{2(\sigma_{f_k})^2}} \tag{9.3}$$

Figure 9.5 shows observed PMF and approximated PDF $Pr_{1.8\ GHz}(K)$ for 256-task (1024-thread) workload. The figure shows that approximation works well in practice.

9.1.2. Probabilistic Power Model

We now need to translate PDF $Pr_{f_k}(K)$ that represents the distribution of usage of frequency f_k to the contribution of that usage to many-core's total power consumption. Using normal approximation of PDF $Pr_{f_k}(K)$, we can find the probability that $K \leq |T|$ tasks would be using frequency f_k, but it does not tell us the composition of those K tasks.

This unknown information makes the translation of PDF $Pr_{f_k}(K)$ to power consumption distribution difficult because different tasks can have different power consumption at the same frequency. Furthermore, a task can also have different power consumption at given frequency depending upon its current execution phase.

A good approximation would be to work out the expected power consumption of task at frequency f_k and assume all K tasks in PDF $Pr_{f_k}(K)$ to have same expected power consumption. Due to the law of large numbers [134], the error introduced by this approximation will reduce with increase in the number of independently executing tasks provided we observe many scheduling epochs. This approximation fits perfectly for design of many-core governors.

Let $\overline{W}(t_i, f_k, \hat{\rho}_{t_i})$ be average power consumption of task t_i at frequency f_k with target QoS set at $\hat{\rho}_{t_i}$. We use probability weighted power consumption of task set T at frequency f_k to obtain expected power consumption of all tasks at that frequency. We then combine it with Equations (9.1) and (9.2) to calculate mean $\mu^W_{f_k}$ and standard deviation $\sigma^W_{f_k}$ of power consumption distribution due use of frequency f_k, respectively.

$$\mu^W_{f_k} = \mu_{f_k} \frac{\sum_{t_i=1}^{|T|} \overline{W}(t_i, f_k, \hat{\rho}_{t_i})\ p(t_i, f_k, \hat{\rho}_{t_i})}{\sum_{t_i=1}^{|T|} p(t_i, f_k, \hat{\rho}_{t_i})} \tag{9.4}$$

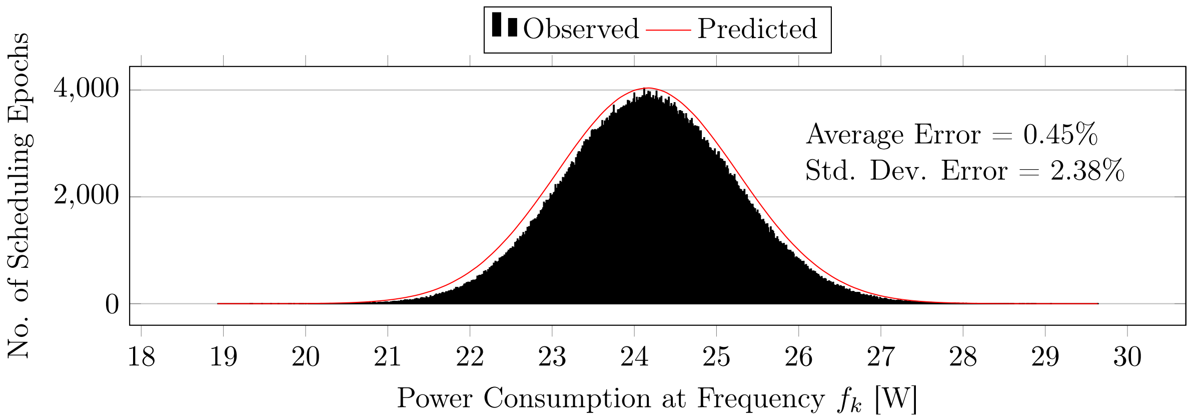


Figure 9.6.: Observed and predicted power consumption distribution due to frequency $f_k = 1.8\ GHz$ for a 1024-thread workload.

$$\sigma^W_{f_k} = \sigma_{f_k} \frac{\sum_{t_i=1}^{|T|} \overline{W}(t_i, f_k, \hat{\rho}_{t_i})\ p(t_i, f_k, \hat{\rho}_{t_i})}{\sum_{t_i=1}^{|T|} p(t_i, f_k, \hat{\rho}_{t_i})} \tag{9.5}$$

PDF $Pr^W_{f_k}(x)$ gives the probability that scheduling epoch will have a power consumption of x Watts due to the usage of frequency f_k.

$$Pr^W_{f_k}(x) = \frac{1}{\sqrt{2(\sigma^W_{f_k})^2\pi}} e^{-\frac{(x-\mu^W_{f_k})^2}{2(\sigma^W_{f_k})^2}} \tag{9.6}$$

Figure 9.6 shows observed PMF and approximated PDF of power consumption distribution at frequency $Pr^W_{1.8\ GHz}(x)$ for 256-task (1024-thread) workload. The figure shows that predicted PDF $Pr^W_{f_k}(x)$ is very close to observed PMF.

We assume power consumption due to individual frequency is a linear combination of power consumptions of the same set of independent tasks. Therefore, all power consumption distributions due to use of frequencies are jointly normal with each other. This relation implies the distribution of their sum which is many-core's total power consumption distribution is also normally distributed. Therefore, we can obtain mean μ^W of total power consumption distribution of many-core by adding means of power consumption distributions due to use of all frequencies.

$$\mu^W = \sum_{j=1}^{|F|} \mu^W_{f_k} \tag{9.7}$$

If power consumption distribution at individual frequencies is assumed to be independent of each other than the standard deviation of total power consumption distribution can be obtained just from the standard deviation of power consumption distribution at individual frequencies.

$$\sigma^W = \sqrt{\sum_{j=1}^{|F|} (\sigma^W_{f_k})^2} \tag{9.8}$$

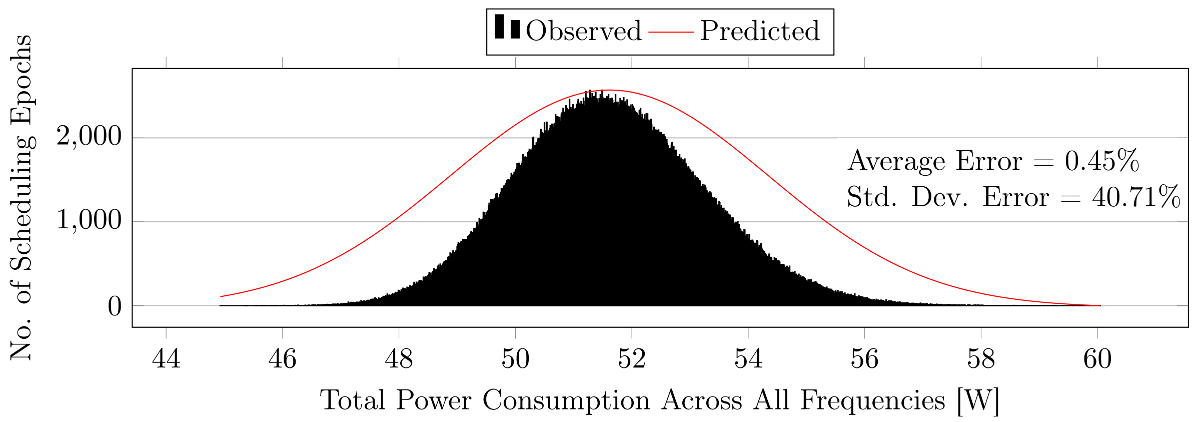


Figure 9.7.: Observed and predicted total power consumption distribution for a 1024-thread workload with assumption that power consumption at individual frequency is independent.

$Pr^W(x)$ gives the probability that in scheduling epoch many-core will have a power consumption of x Watts.

$$Pr^W(x) = \frac{1}{\sqrt{2(\sigma^W)^2\pi}} e^{-\frac{(x-\mu^W)^2}{2(\sigma^W)^2}} \tag{9.9}$$

Figure 9.7 shows observed PMF and predicted PDF of total power consumption distribution $Pr^W(x)$ for a 256-task (1024-thread) workload. The figure shows that our predicted distribution has an unacceptably high error due to imprecise standard deviation.

Error occurs because even though all power consumption distributions at individual frequencies are jointly normal with each other, they are not independent. By design when task switches from one frequency to another, it leads to decrease in power consumption due to the former frequency with a simultaneous increase in power consumption due to latter frequency. Therefore, all power consumption distributions due to use of different frequencies negatively correlate with each other. Therefore, we can obtain standard deviation σ^W of total power consumption distribution by adding variance of power distributions at individual discrete frequencies adjusted with their covariance.

$$\sigma^W = \sqrt{\sum_{j=1}^{|F|} (\sigma^W_{f_k})^2 + 2\sum_{j<j'} \rho^W_{f_k,f_{k'}} \sigma^W_{f_k} \sigma^W_{f_{k'}}} \tag{9.10}$$

where $\rho^W_{f_k,f_{k'}}$ is a correlation coefficient between power consumption at frequencies f_k and $f_{k'}$. We can learn correlation coefficient $\rho^W_{f_k,f_{k'}}$ by taking power samples online. This sampling has very less overhead compared to continuous sampling done online in a non-probabilistic governor and hence does not limit the scalability of *Q-PMG*.

PDF $Pr^W(x)$ gives the probability that in scheduling epoch many-core will have a power consumption of x Watts.

$$Pr^W(x) = \frac{1}{\sqrt{2(\sigma^W)^2\pi}} e^{-\frac{(x-\mu^W)^2}{2(\sigma^W)^2}} \tag{9.11}$$

Figure 9.8 shows observed PMF and predicted PDF of total power consumption distribution $Pr^W(x)$ for 256-task (1024-thread) workload with covariance between frequencies considered. The figure that shows error now is minimal.

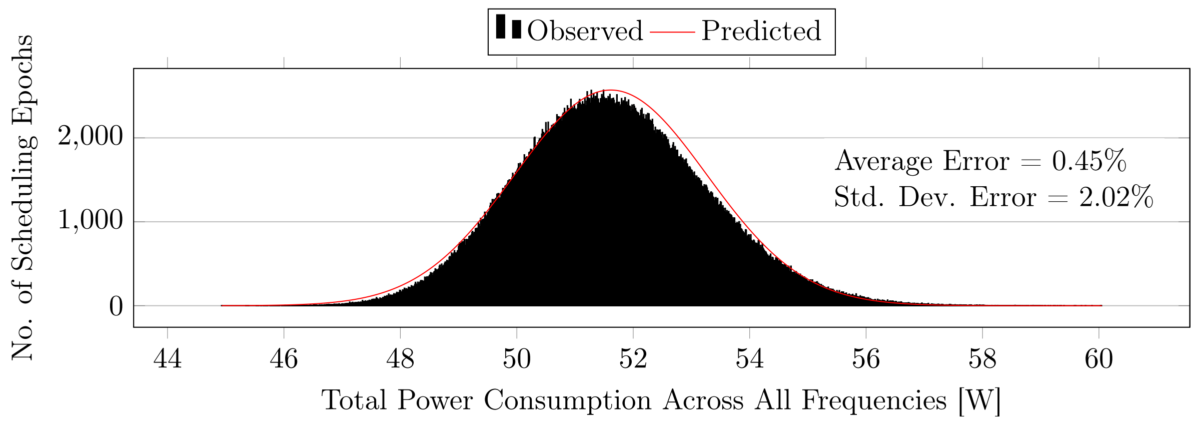


Figure 9.8.: Observed and predicted total power consumption distribution for 1024-thread workload.

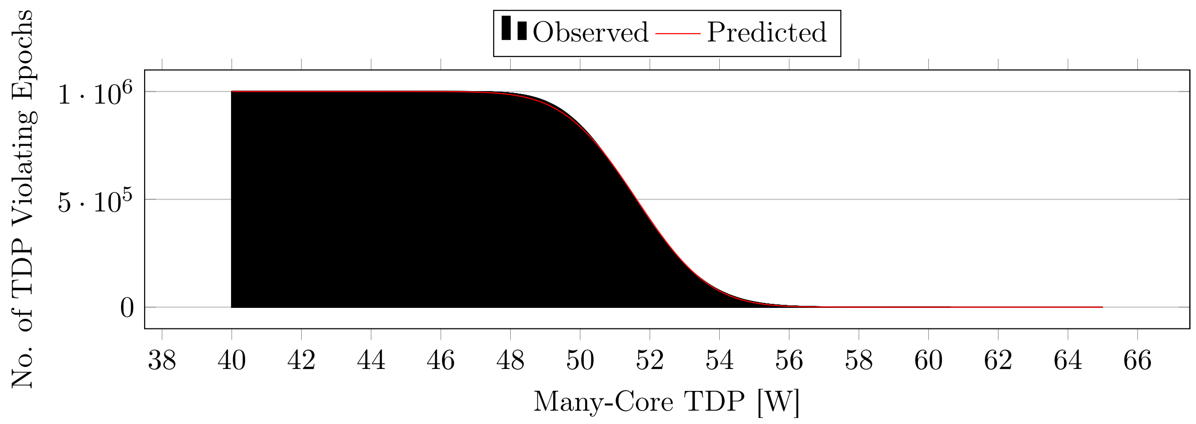


Figure 9.9.: Observed and predicted TDP violation distribution for 1024-thread workload.

9.1.3. Probabilistic TDP Model

Let $\hat{W}$ be TDP of many-core. Q-function $Q(\hat{W})$ gives the probability that scheduling epoch will violate TDP. We use Chebyshev fitting [138] to obtain the approximated value of Q-function $Q(\hat{W})$ numerically at run-time.

$$Q(\hat{W}) = 1 - \int_0^{\hat{W}} Pr^W(x)\ dx \tag{9.12}$$

Figure 9.9 shows observed and predicted (using Q-function $Q(\hat{W})$) distribution of TDP violating scheduling epochs for 256-task (1024-thread) workload. The figure shows that our predicted distribution of TDP violating epochs is accurate.

9.1.4. Power Budgeting Algorithm

Algorithm 3 shows probabilistic power budgeting used in *Q-PMG*. *Q-PMG* cannot give a deterministic guarantee that TDP violation will never happen but it can reduce the probability of TDP violation to such a low value that it practically never occurs. Furthermore, TDP is a soft-constraint, and thermal emergency only occurs when many-core violates TDP for prolonged durations. Few TDP violating epochs spread out over time are generally acceptable. Hardware-triggered frequency throttling via DTM can act as a backup if TDP violations under

Algorithm 3 Stochastic power budgeting used in *Q-PMG*.

```
Input: $T, \hat{W}, \delta_{\hat{W}}$;
Output: $\Delta_I$;
1: $\forall t_i \in T$ Read Profiled Data;
2: for $\Delta_I = 1.0$ to $0.0$ do
3:     for $j = 1$ to $|F|$ do
4:         $\hat{\rho}_{t_i} = \hat{\rho}_{t_i} * \Delta_I \forall t_i \in T$
5:         Calculate $\mu_{f_k}$ using Equation (9.1);
6:         Calculate $\sigma_{f_k}$ using Equation (9.2);
7:         Predict $\mathrm{Pr}_{f_k}(K)$ using Equation (9.3);
8:         Calculate $\mu^W_{f_k}$ using Equation (9.4);
9:         Calculate $\sigma^W_{f_k}$ using Equation (9.5);
10:        Predict $Pr^W_{f_k}(x)$ using Equation (9.6);
11:    end for
12:    Calculate $\mu^W$ using Equation (9.7);
13:    Calculate $\sigma^W$ using Equation (9.10);
14:    Predict $Pr^W(x)$ using Equation (9.11);
15:    Predict $Q(\hat{W})$ using Equation (9.12);
16:    $\hat{\rho}_{t_i} = \hat{\rho}_{t_i} / \Delta_I \forall t_i \in T$
17:    if $Q(\hat{W}) < \delta_{\hat{W}}$ then
18:        break;
19:    end if
20:    $\Delta_I = \Delta_I$ - .01;
21: end for
22: return $\Delta_I$;
```

Q-PMG pushes chip temperature dangerously high. *Q-PMG* also allows for a tradeoff between TDP violation risk with performance using TDP risk threshold $\delta_{\hat{W}}$.

Algorithm 3 takes as input set of tasks executing on many-core T, TDP $\hat{W}$, and threshold $\delta_{\hat{W}}$. It then calculates the risk of TDP violation $Q(\hat{W})$. If the risk is higher than threshold $\delta_{\hat{W}}$, then all tasks are forced to pay equal performance penalty by discounting their target IPS (QoS) by a factor of Δ_I due to power budgeting constraint. Higher value of discount Δ_I signifies higher efficacy of governor. Algorithm 3 is executed only when some task enters or leaves many-core changing composition of tasks under execution on many-core. Note, DVFS is performed locally and independently by tasks themselves without any relation to Algorithm 3.

9.1.5. Complexity

In Algorithm 3 since QoS discount factor Δ_I always take a value between 1.0 and 0.0, the computational complexity of loop in Step 2 is constant. We set the granularity at which this loop iterates at 0.01. The worst-case computational complexity of any step in the frequency loop at Step 3 is O($|T|$), so loop's complexity is O($|F||T|$). Since all other steps have computational complexity less than frequency loop, the worst-case computational complexity of *Q-PMG* is O($|F||T|$). Use of centrally available probabilistic profiles in *Q-PMG* results in worst-case space complexity of O($|T|$). Need to propagate results to tasks and make online observations to learn covariance results in worst-case communication complexity of $O(|T|)$. Note that both worst-case computation and communication complexity is $O(1)$ in scheduling epochs where no task enters or leaves, or *Q-PMG* does not make any online observation.

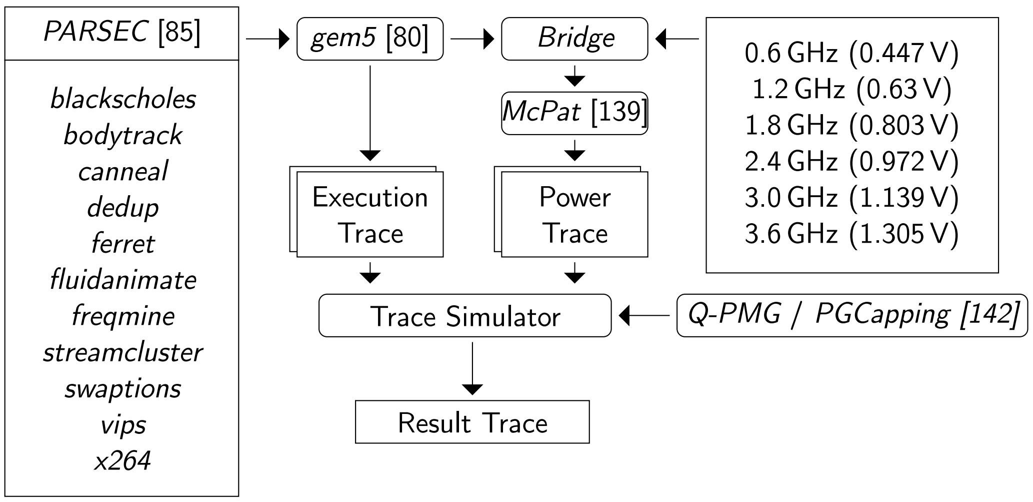


Figure 9.10.: Experimental Setup for *Q-PMG*.

9.2. Experimental Evaluations

9.2.1. Experimental Setup

Experimental setup used in the evaluation of *Q-PMG* is very similar to the setup used in Chapter 8. There are some crucial differences such as availability of multiple DVFS levels as shown in Figure 9.10 and hardware-triggered DTM.

The 22-nm planar CMOS cores have in-order pipeline with low-power design. Core's maximum power consumption is around 0.0065 W (or 0.25 W) at lowest (or highest) frequency of 0.6 GHz (or 3.6 GHz). Ambient temperature is set at 40 °C whereas DTM is triggered when the many-core temperature exceeds 85 °C. We set thermal modeling parameters such that hardware will never trigger DTM if many-core always operates within TDP of 45 W.

9.2.2. Comparative Baseline

We choose to compare *Q-PMG* against *PGCapping* governor [142] both being centralized governors. *PGCapping* uses an efficient non-probabilistic *Quicksearch* greedy algorithm to perform DVFS-based QoS-aware power budgeting for multi-/many-cores. *Quicksearch* similar to *Q-PMG* assumes the availability of per-core DVFS for power budgeting but unlike *Q-PMG* provides an unassailable deterministic guarantee that many-core would never violate TDP.

Quicksearch operates by power/performance ratios. Depending upon whether current power consumption of many-core is above or below TDP, *Quicksearch* calculates ratio of power decrease to performance loss $D_{power-perf}$ or ratio of the performance gain to power increase $D_{perf-power}$ for all cores, respectively. The frequency of core with highest power decrease to performance loss ratio $D_{power-perf}$ (or highest performance gain to power increase ratio $D_{perf-power}$) is decreased (or increased) if power is expected to be above (or below) TDP. *Quicksearch* then recalculates $D_{power-perf}$ (or $D_{perf-power}$) for the task whose frequency it changes. *Quicksearch* algorithm iteratively repeats itself till it meets power constraints.

Since we operate with multi-threaded tasks with the assumption that all cores allocated to task operate at a same frequency, we calculate ratios $D_{power-perf}$ and $D_{perf-power}$ for *PGCapping* at task granularity rather than core granularity. Furthermore, *PGCapping* initially used the product of core utilization and core frequency as a measure of performance (QoS), which we replace with IPS in this work for fair comparison.

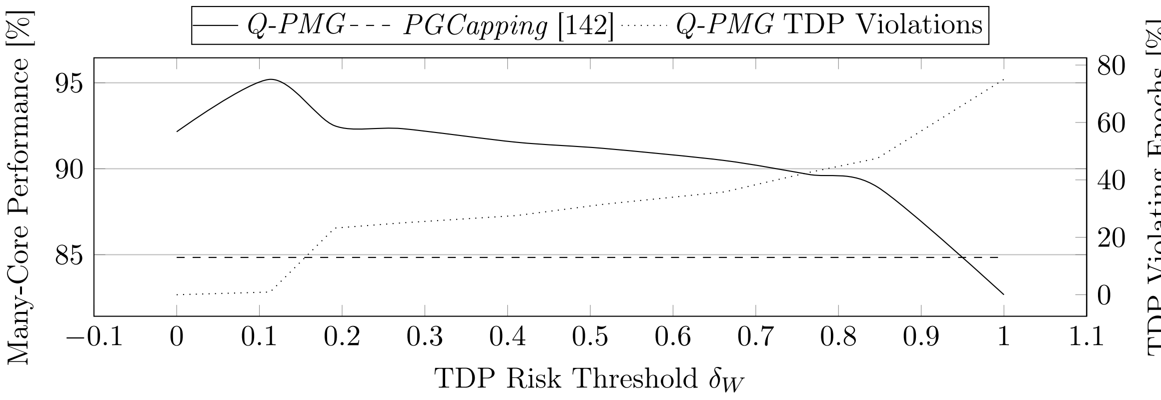


Figure 9.11.: System performance Δ_I comparison between *Q-PMG* and *PGCapping* for different values of TDP risk threshold δ_W when executing 1024-thread workload with TDP $\hat{W}$ set at 45 W.

When *Quicksearch* algorithm is implemented with the help of *quicksort* and *binary search* algorithms, complexity of *Quicksearch* in worst-case works out to be $O(|F||T| \ln |T|)$, which theoretically is a factor of $O\ (ln|T|)$ more than *Q-PMG*.

9.2.3. Stochastic vs. Non-Stochastic Performance

We simulate many-core operating in a closed system [32] to compare the efficacy of different governors. Neither *Q-PMG* nor *PGCapping* is limited to only closed systems. Many-core attains peak performance (100%) when all QoS tasks achieve their target QoS at all times. We cap contribution of tasks to system performance at 100% of its desired QoS.

Figure 9.11 shows how performance measured in percentage of desired QoS sustained for a task on average for 256-task (1024-thread) workload changes with different values of TDP risk threshold δ_W. Since *PGCapping* does not consider δ_W, it results in same performance for all values of δ_W whereas *Q-PMG* allows a tradeoff between performance and TDP risk threshold δ_W. Increase in TDP risk threshold δ_W leads to increase in the percentage of TDP violating epochs under *Q-PMG* as also shown in Figure 9.11.

Ignoring TDP beyond a certain level can lead to performance loss instead of gain as hardware-triggered thermal throttling (DTM) on TDP violations can substantially deteriorate performance. We can see this effect in Figure 9.11 for higher values of TDP risk threshold δ_W where DTM deteriorates many-core performance.

The figure shows that *Q-PMG* results in superior performance compared to *PGCapping* even when TDP risk threshold δ_W is set to 0. *PGCapping* penalizes tasks asymmetrically resulting in several tasks operating far above their target QoS. *Q-PMG* on the contrary, penalizes all tasks fairly in equal proportions which also results in better performance.

9.2.4. Stochastic vs Non-Stochastic Scalability

Power-budgeting governor inherently needs to gather state information from across many-core and send back its decision in the same manner. We use cycle-accurate simulation on *gem5* to determine overhead of this bidirectional communication on varisized many-cores. State information is collected using distributed sub-routine using master-slave thread model.

Subroutine's master thread starts on a random core and spawns slave threads across all other cores. Slave threads then collect per-core statistics such as its power consumption and forward it to master thread. Master thread then processes this information for power budgeting and sends the resultant decision back to slave threads for further propagation to appropriate stakeholders.

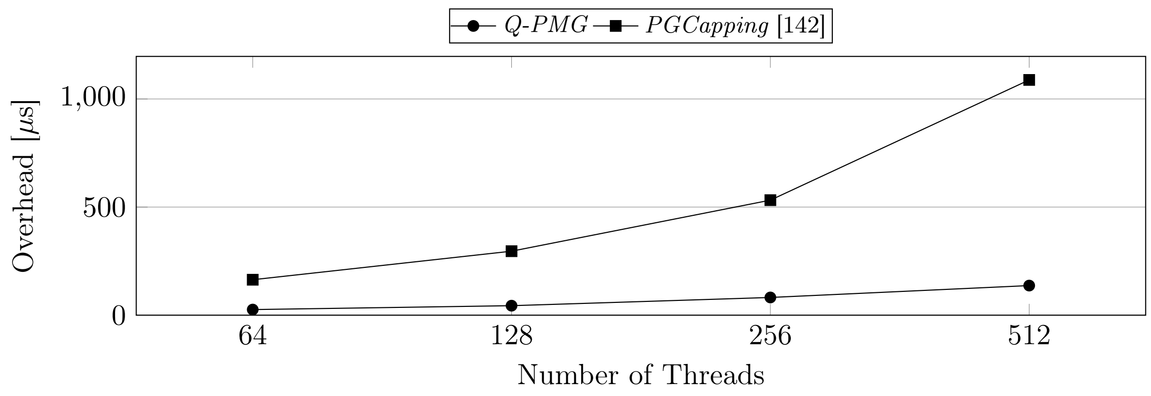


Figure 9.12.: Measured worst-case overheads for *Q-PMG* for varisized workloads.

We are forced to change our ISA from *Alpha* to *ARM* for this experiment because for *Alpha* ISA *gem5* only supports multi-threading (*OpenMP* or *pthread*) in FS mode. Simulation of large-size many-core in FS mode is not time-wise feasible as many-core takes forever even to boot. Large-size many-core with *ARM* ISA, on the other hand, can be simulated in a reasonable time in SE mode of *gem5* with the help of *m5thread* library.

Our proof-of-concept simulation report that time overhead for bidirectional communication between the governor and executing tasks can be as high as 62 μs on 512-core many-core, translating into an overhead of 0.62% for 10 ms scheduling epoch. Note that both *Q-PMG* and *PGCapping* need this bidirectional communication sub-routine to operate, but *Q-PMG* uses it only when task arrives or leaves many-core, while *PGCapping* employs it in every scheduling epoch. Therefore, over long-term overhead of collecting state information and disbursing decisions is much lower in *Q-PMG* than *PGCapping*.

Figure 9.12 shows worst-case scheduling overheads of *Q-PMG* and *PGCapping* for varisized workloads obtained using representative cycle-accurate simulations performed on *gem5*. Overheads reported in Figure 9.12 combines both processing and communication overheads. Our proof-of-concept simulations show *Q-PMG* is highly scalable and has nearly 6.48x less worst-case scheduling overhead for a 256-thread workload in comparison to *PGCapping*.

9.3. Summary

We introduced a QoS-aware probabilistic power budgeting governor for many-cores called *Q-PMG* in this chapter. *Q-PMG* provides strong probabilistic guarantees on the risk of TDP violation while allowing tradeoff of that risk with performance. Compared to a non-probabilistic governor, *Q-PMG* provides equivalent performance but with computational complexity reduced by factor O (ln n). Therefore, *Q-PMG* can scale up more efficiently as the number of cores and tasks executing on the many-core increase.

This chapter presents the last many-core scheduler covered in this dissertation. We conclude this dissertation in next chapter. Conclusion is also followed by description of a toolchain called *HotSniper* in the appendix. *HotSniper* toolchain developed over the course of this dissertation provides researchers with means to evaluate many-core schedulers similar to ones developed in this dissertation on real-world representative many-core architectures.

10. Conclusion

10.1. Dissertation Summary

In this dissertation, we focused on developing schedulers for many-cores with different optimization objectives. The goal was to develop schedulers that are more scalable than multi-core schedulers while making minimal or no compromise on the quality of schedule. We also strove to develop non-heuristic schedulers which provide strong theoretical guarantees on the quality of schedule and hence do not suffer from poor schedules in corner cases. We used four different approaches to achieve our aforementioned goal in this dissertation.

The first approach was to develop distributed schedulers which can distribute their scheduling overheads across multiple or all cores and thereby can scale up with the increase in the number of cores in many-cores. The second approach was to develop centralized schedulers that were based on fast algorithms such as a greedy algorithm. The third approach was to use exact algorithms but at the same time find ways to prune the search-space so that they can be applied at run-time with minimal overheads. The fourth and final approach was to develop centralized probabilistic schedulers that work on optimizing the distributions and hence have drastically reduce overheads compared to non-probabilistic schedulers.

In Chapters 1 and 2, we introduced the many-core computing paradigm along with the necessary background. We elaborated upon as to how many-cores evolved from uni-cores via multi-cores and the essential role many-cores are going to play in embedded systems. We also studied as to why current multi-core schedulers cannot efficiently operate on many-cores. In essence, we justified the need to develop new many-core schedulers even for the most basic of optimization objectives. We also discussed on how many-core computing paradigm is substantially different from multi-thread and multi-grid computing paradigms though they may all look very similar to a non-expert. We also explained three kinds of systems namely fixed, closed and open systems wherein one can deploy a many-core. We then explained the three different kinds of tasks namely rigid, moldable and malleable tasks which one can execute on a many-core. We also explained several different ways to classify many-cores. Finally, we concluded the chapters with details of *InvasIC* computing project in which we participated followed by details of some commercially available many-core platforms and OS.

In Chapter 4, we introduced two new many-core schedulers with the goal of performance maximization. One was a distributed scheduler while the other one was a centralized greedy scheduler. Both schedulers were proven optimal. In Chapter 5, we introduced a distributed many-core scheduler with the goal of fairness maximization. The scheduler is proven optimal in the case of fixed performance. In Chapter 6, we introduced a distributed many-core scheduler capable of improving the performance of the many-core by performing task defragmentation. The scheduler can perform optimal defragmentation in case of tasks being constrained to produce only power-of-two threads. In Chapter 7, we explained as to why the scheduler introduced in Chapter 6 cannot work with a many-core with S-NUCA caches. We then introduced another scheduler that exploits S-NUCA design to reduce the search-space and thereby maximize performance using a centralized BnB algorithm. In Chapter 8, we presented a centralized probabilistic scheduler to maximize the performance of a many-core under fixed power budget. We then extended the scheduler to maximize the performance of a many-core when operating with tasks each of whose performance is upper-bounded by a user-defined QoS.

10.2. Future Work

This dissertation opens venues for further research, which we hope to cover in the near future. Each chapter assumes one of three types of tasks - rigid, moldable or malleable and one of the three types of systems - fixed, closed and open. In total, there can be nine unique combinations of tasks and systems. Schedulers presented in this dissertation can potentially be easily extended to work with many of these combinations, but some of the combinations inherently may require entirely new designs. In the following paragraphs, we iterate through all schedulers presented in this dissertation and present some related unexplored research ideas.

Schedulers presented in Chapter 4 are proven optimal because they address the question of how many cores a task should be assigned to maximize the overall performance of many-core. Schedulers cannot solve the problem optimally if we supplement the question by asking not just how many but also which cores a task should be assigned to maximize performance. Unfortunately, under this formulation, the problem becomes NP-hard. Yet, it remains an open question whether the convex substructures which were the basis of our scheduler designs still hold under the new formulation and if they can help us to design a new optimal scheduler.

Scheduler presented in Chapter 5 is proven optimal regarding fairness maximization but only in the case of a fixed performance. The results presented are of considerable theoretical interest, but as we also argued in the chapter, the results are of limited use in practice. Even though the problem of fairness maximization is NP-hard in the general case, it remains an open question that whether the convex substructure in the problem can somehow also be used to obtain the optimal solution in the case of unconstrained performance.

Scheduler presented in Chapter 6 can optimally defragment a many-core but only when tasks are constrained to produce power-of-two threads. As also argued in the chapter, the cost of maintaining optimality is too high when the many-core operates with tasks that cannot scale or spawn threads on all allocated cores. Even though many-core defragmentation problem is NP-hard in the general case, it remains to be seen whether a many-core can continue to be defragmented optimally with less restrictive constraints on thread spawning than power-of-two. A good starting point would be to modify the thread spawning constraint to Fibonacci numbers which have some elegant spatial property, and the gap between two consecutive numbers in the Fibonacci series is smaller than two consecutive numbers in power-of-two series.

Scheduler presented in Chapter 7 maximizes the performance of a many-core with S-NUCA caches due to its inherent knowledge of the S-NUCA design. Though the results may look completely different if we also take into consideration the power dissipation by active cores. The same also holds true for the scheduler presented in Chapter 6. A task allocation which places threads of a task closer together in a compact pattern may outperform an allocation where threads are further apart when cores in both allocations run at the same frequency. Nevertheless, when power dissipation of cores is considered, cores closer together would also result in a thermal hotspot because of more significant thermal conduction between them. The hotspot may force the cores to run at a lower frequency using DVFS resulting in lower or even negative performance gains than what we obtained in Chapter 7. It looks like neither purely optimizing for performance nor compactness may be optimal in practice, and we can obtain better results by developing a scheduler that is aware of the underlying trade-off.

Probabilistic schedulers presented in Chapters 8 and 9 present an entirely new line of thought in scheduler design for many-cores. It would be interesting to see how far this idea can extend beyond the initial goal of power budgeting presented in this dissertation.

Finally, there are several other optimization goals for many-core schedulers which we did not cover in this dissertation such as thermal management. We also only considered homogeneous (adaptive) many-cores in this dissertation. We hope to develop many-core schedulers for heterogeneous (asymmetric) many-cores in the near future.

A. Toolchain for Interval Thermal Simulations of Many-Cores

We introduce a toolchain for interval thermal simulations of many-cores deployed in open systems in this appendix. There are several types of processor simulators available to explore many-core design. *gem5* [80] is a popular cycle-accurate simulator used for precise processor simulations. Unfortunately, simulation-time in *gem5* for many-cores can often be intractably large. For example, cycle-accurate simulation of single *PARSEC* [85] task can take more than a week to complete within *gem5*.

Simulation-time constraints often force users to employ custom two-stage trace-based many-core simulators as we did in Chapter 4, 5, 8, and 9. In first stage of a trace-based simulator, all tasks are executed individually on many-core using a cycle-accurate simulator to obtain isolated execution traces. These traces are then merged into second stage high-level simulator to simulate simultaneous multi-program execution of traced tasks. A significant drawback of the trace-based simulator is implicit assumption that isolated execution traces obtained in the first stage remain valid in multi-program execution simulated in the second stage. Unfortunately, due to the manifestation of shared-resource contentions during real-world multi-program execution, this assumption no longer holds.

Sniper simulator [110], based initially on *Graphite* simulator [146], was developed to bridge the gap between slow but precise cycle-accurate many-core simulation and fast but imprecise trace-based many-core simulation. Authors of *Sniper* used *PIN* [147] binary instrumentation tool to perform interval simulation of processors. Interval simulations work on the observation that shared-resource contentions in processor only manifest when there are miss events such as cache-misses [148]. This observation allows interval simulators to loosen tight execution entanglement of co-executing threads in their simulation models where synchronization between simulated thread executions only occurs when any one of them encounters miss event. This relaxed synchronization which only occurs at specific discrete intervals results in several folds decrease in simulation-time with minimal loss of accuracy. Both accuracy and simulation-time of interval-based simulations is in between cycle-accurate and trace-based simulations.

We have used *Sniper* to simulate 64-core many-core for 10 seconds of system-time at full utilization in feasible simulation-time of 10 hours in Chapter 6 and 7. *Sniper* is well-suited for system research in processors especially when there is a transition from processors with a small number of cores (multi-cores) to processors with a large number of cores (many-cores) [21]. Many-cores also contain new micro-architectural features such as NoC not present in multi-cores. Therefore, it requires typically more time to simulate many-cores than multi-cores.

Unfortunately, we were not able to perform interval thermal simulations of many-cores in *Sniper* due to lack of integration with a thermal modeling tool. As a result, *Sniper* simulator could not be used to study thermal properties of the many-core design. With the failure of Dennard Scaling [17] and emergence of Dark Silicon [19], thermal-aware hardware-software codesign of many-cores has become an active subject of research especially in embedded systems where lack of active cooling solutions turns high on-chip temperatures into a debilitating constraint on performance. Fast, accurate and real-world representative interval thermal simulations would provide means for thermally-efficient many-core design hastening the adoption of many-cores in different embedded domains.

We were also not able to perform interval-based simulations of many-core deployed in open system [32] using default *Sniper* due to missing support for the feature in code. These shortcomings led to the development of *HotSniper* toolchain introduced in this chapter. *HotSniper* adds support for both open systems (Section A.1) and interval-based thermal simulations (Section A.2) to most-recent *Sniper-6.1*.

Work presented in [149] is closest to this work wherein authors have integrated *HotSpot* [150] temperature modeling tool with *Sniper* many-core simulator as a plugin. *HotSpot* plugin developed in [149] performs thermal simulations after *Sniper* simulator has finished performing processor simulations. As a result, temperatures from thermal simulations produced using *HotSpot* cannot be sent back to *Sniper* as feedback to influence processor simulations itself. In contrast, *HotSniper* toolchain performs *Hotspot* thermal simulations in parallel to *Sniper* processor simulation. Hence, temperature feedback in current scheduling epoch from *HotSpot* is available within *Sniper* to take thermal-aware scheduling decisions in next upcoming epoch.

We are also not aware whether authors released the plugin developed in [149] to the community under an open-source license or not. In contrast, we have already released our code for *HotSniper* as an open-source project on *GitHub* and several external research groups have commenced its use. We have adequately documented code for *HotSniper* and designed it carefully so that its users can integrate code for their algorithms with minimal efforts. Learning-curve of *HotSniper* is also streamlined using “how-to" readme.

A.0.1. Open-Source Contribution

The source code for *HotSniper* is available at https://github.com/anujpathania/hotsniper, and we released it under MIT License for unrestricted use like the original *Sniper* simulator.

A.1. Open System Support

We can deploy multi-/many-cores in three types of embedded systems [32]: fixed, close and open. *Sniper* simulator supports fast and accurate interval simulations of many-cores deployed in both fixed and closed systems out-of-the-box, but it does not support open systems by default. Open systems are the most complex system to model. Therefore, open systems are most suitable to evaluate generic scheduling algorithms.

We introduced support for open systems in *Sniper* by adding a new “*open*" scheduler. We use “*pinned*" scheduler that ships by default in *Sniper* as the template for new *open* scheduler. The user can switch to *open* scheduler using main *Sniper* configuration file. There are other properties associated with the *open* scheduler that can be set using the same file.

A user can easily configure the algorithm used for scheduling, which we initially set to *default* algorithm. The *default* algorithm is a simple illustrative algorithm which pins a thread to be scheduled to first free core - core assigned to no other thread - it finds and thereby employs one-thread-per-core execution model [69] throughout this dissertation. *HotSniper* users are expected to substitute *default* algorithm with their algorithm which they wish to evaluate. The *default* algorithm's code can act as starting template for more complex implementation, where one can pin multiple threads to a single core for context-switched execution.

Length of scheduling epoch can also be configured; whose default value is 10 ms similar to default *Linux* kernel [33]. Scheduling epoch is granularity at which many-core invokes logic of scheduler for task scheduling. A user can set the value of epoch as low as 100 ns at the cost of increased simulation-time overhead. A smaller value of epoch leads to a larger simulation-time because of more frequent invocation of scheduler algorithm that would trigger actions like updating various queues of the open system. The configurable queuing policy has by default FIFO design and serves the task on FCFS basis. Distribution of arrival time of open workloads

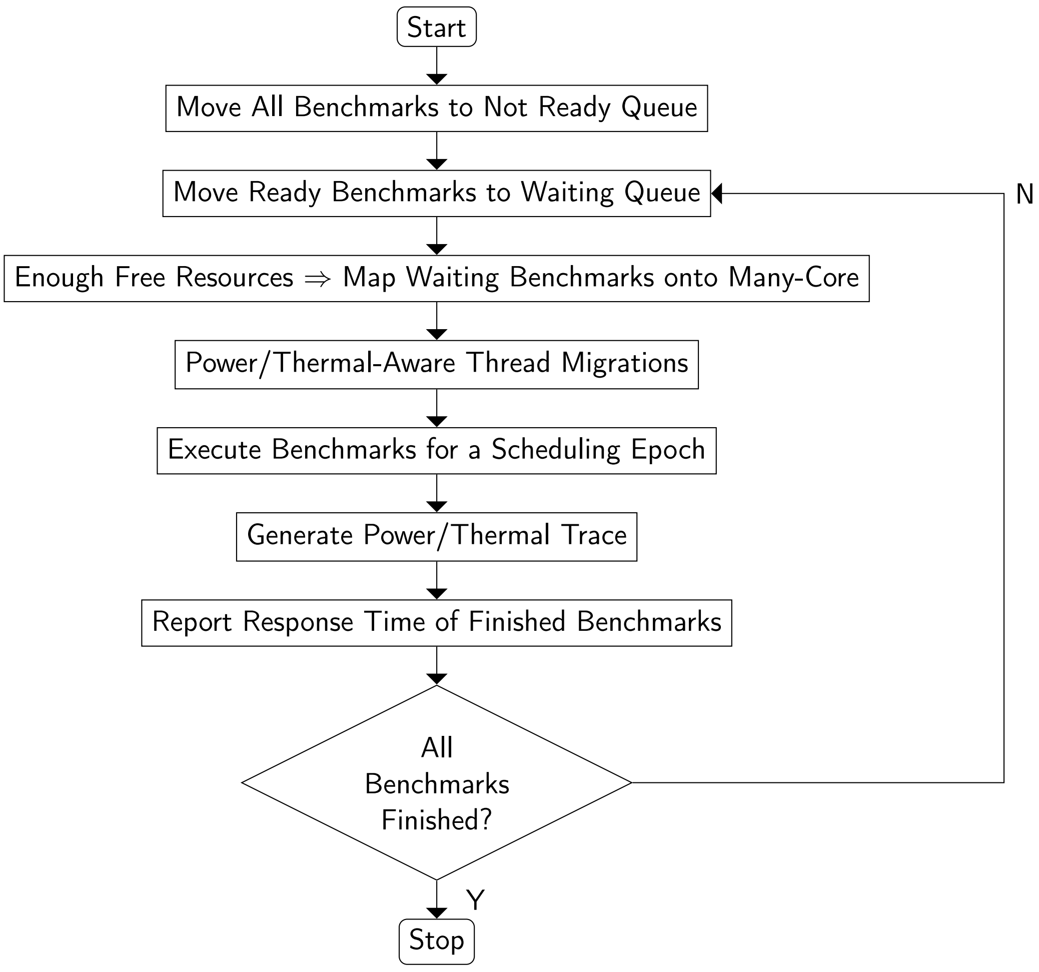

Start
Move All Benchmarks to Not Ready Queue
Move Ready Benchmarks to Waiting Queue
N
Enough Free Resources ⇒ Map Waiting Benchmarks onto Many-Core
Power/Thermal-Aware Thread Migrations
Execute Benchmarks for a Scheduling Epoch
Generate Power/Thermal Trace
Report Response Time of Finished Benchmarks
All Benchmarks Finished?
Y
Stop

Figure A.1.: Execution flow (timeline) for *HotSniper* toolchain.

can also be configured and by default has a uniform distribution. A user can also configure arrival rate and arrival interval for uniform distribution.

Figure A.1 shows execution flow (timeline) for *HotSniper* toolchain. When *Sniper* simulator starts execution, it produces master thread for all tasks to be simulated. Open scheduler immediately puts all master threads to sleep and thereby adds the task to the not ready queue of the open system. At appropriate simulated system-time based on arrival distribution, the task is moved to waiting queue and placed in its respective position in the queue based on queuing policy. Scheduler revives master thread of task at the front of the waiting queue when based on its logic deems it appropriate to start task's execution. The revived master thread then produces slave threads and continues its execution. When the task finishes, *open* scheduler reports its response time, which is the sum of its waiting and service time. Average response time is preferred metric for measuring the performance of workloads in open system [32].

It is important to note that *Sniper* simulator requires at least one thread to be under active execution at all time. Without active thread incrementing the simulated system-time, interval simulation within *Sniper* will fail to progress and go into a deadlock. The *open* scheduler thereby prefetches master thread of task in front of the queue for execution to prevent simulation from crashing (hanging) if the simulated system is expected to go idle. Prefetching forwards simulated system-time accordingly, so that reported response time still remains correct.

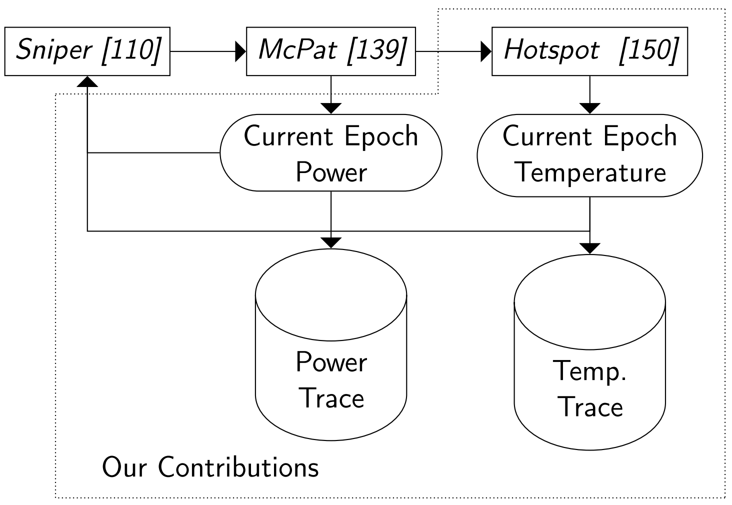


Figure A.2.: *HotSniper* toolchain with current temperature and power consumption feedback.

A.2. Thermal Simulations

We integrate RC-thermal network based *HotSpot-6.0* temperature modeling tool into *Sniper* to enable thermal simulations using feedback-driven toolchain as shown in Figure A.2. *Sniper* contains integrated *McPat-1.0* [139] power modeling framework which can provide power consumptions of various processor components.

In *Sniper*, *McPat* is executes after *Sniper* simulator has completed execution and only reports average power consumption of processor components observed during entire execution. We slightly modified integrated *McPat* to dump power consumption numbers at regular intervals while processor simulation is still ongoing. This dumping allows generation of power-trace which shows the power consumption of various components over simulated execution time. Most of the code required for this dumping is already implicitly present in *Sniper*.

Power consumption data generated after every interval is then fed into *HotSpot* to dump corresponding temperature data which shows the corresponding current transient temperature of each of processor components. This dumping allows us to generate temperature-trace corresponding to power-trace which shows the transient temperature of various processor components over simulated execution time.

Note that users are expected to provide approximate floorplan for the processor they are simulating to generate thermal-trace. Name of floorplan file must be provided using attribute *"floorplan"* in main *Sniper* configuration file. It is also important to mention that there must be block defined in floorplan for every processor component being power traced using *McPat*. *HotSniper* allows for toggling of processor components being traced using *McPat* for power/thermal simulations using the same configuration file.

HotSniper also allows input of most recent power consumption and temperature of processor components into the *open* scheduler. This power and thermal information can then be used to make scheduling decisions for power and thermal management, respectively as also shown in Figure A.1. Decisions will then influence future power and thermal data and thereby feedback loop as shown in Figure A.2 is completed. By default, *HotSniper* generates power and thermal data every 1 ms. Data can be generated at granularity as low as 100 ns at the cost of increase in the simulation-time overhead.

Figure A.3 shows the transient temperature of one of hottest core component – Renaming Unit (RU) – of all four cores of our simulated processor when we execute a four-threaded instance of *blackscholes* with *sim-small* input on the simulated processor. We use *block* [150]

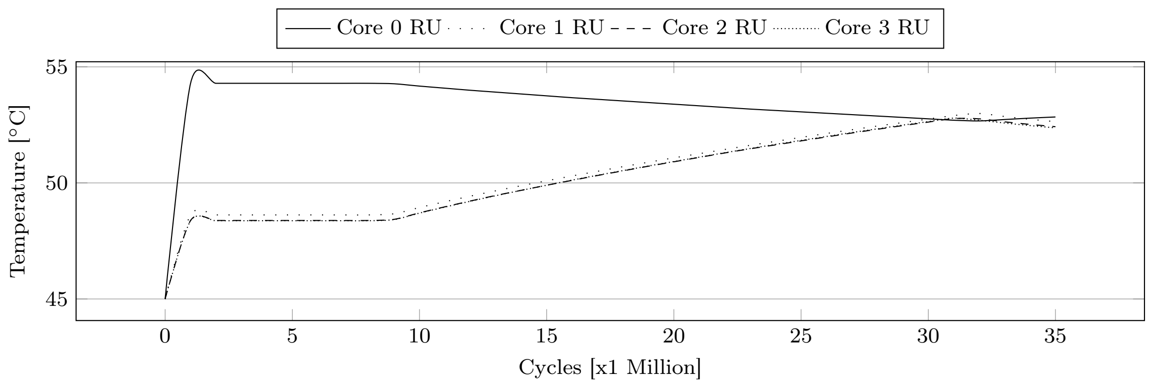


Figure A.3.: Transient temperature of RU of different cores of our simulated processor while executing four-threaded instance of *blackscholes* with *sim-small* input using the *HotSniper* toolchain.

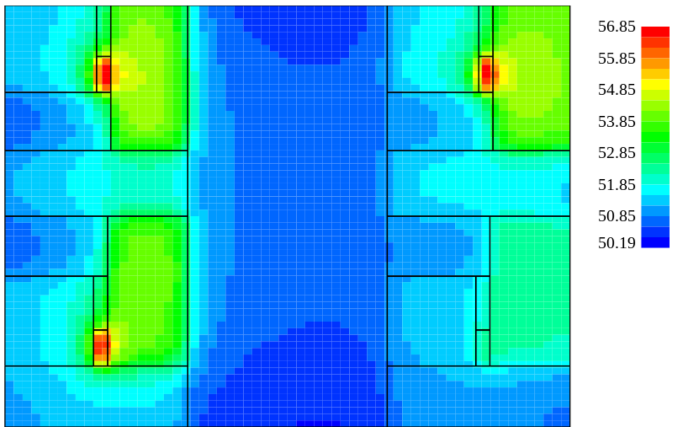


Figure A.4.: Steady-state temperature of our simulated processor while executing a four-threaded instance of *blackscholes* with *sim-small* input using *HotSniper* toolchain.

thermal model of *Hotspot* by default in *HotSniper*. However, one can also use *grid* thermal model [150]. Note that simulation-time with *grid* model is several times larger than simulation-time with *block* model. Transient temperature-trace produced using *grid* model can also be used to generate steady-state temperature using *Hotspot* in-built tools as shown in Figure A.4.

A.3. Experimental Setup

A.3.1. Host System

We evaluate *HotSniper* on *Intel(R) Core(TM) i5-2500K* processor with 6 GB of DRAM running *Ubuntu 15.10* OS. *Sniper* is compiled with *gcc-4.8* and *g++-4.8 GNU* compilers and uses *PIN 2.14-71313* binary instrumentation tool from *Intel*.

A.3.2. Simulated Processor

We use quad-core 45-nm processor with *x86 Gainestown* micro-architecture as the simulated processor. Each core has 32 KB private L1 instruction and data cache, alongside 512 KB private L2 cache. All cores share 8 MB L3 cache. Figure A.5 shows an abstract block diagram of processor's floorplan used for thermal simulations. Each core is composed of Instruction Fetch Unit (IFU), RU, Memory Management Unit (MMU), Execution Unit (EU), and Load Store Unit (LSU) besides L2 cache.

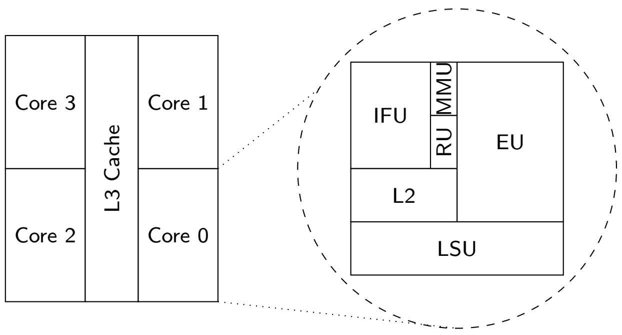


Figure A.5.: Example block diagram of floorplan used for thermal simulation.

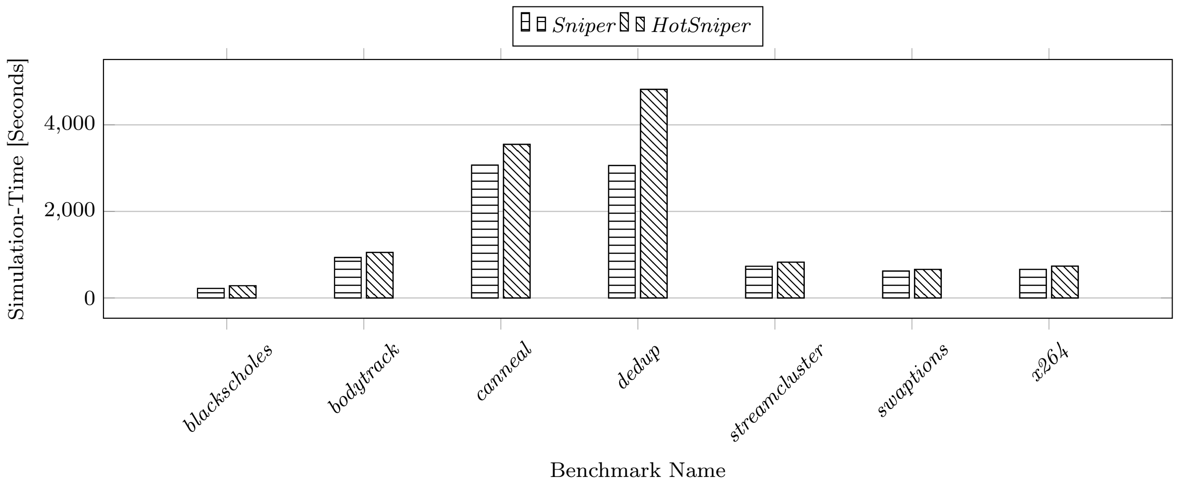


Figure A.6.: Simulation-time of four-threaded instances of various tasks with *sim-small* input for our simulated processor on *Sniper* simulator and *HotSniper* toolchain.

HotSniper is not limited to simulating above architecture only. It can simulate both NoC-based and bus-based multi-/many-core architectures up to 64 cores in a reasonable time.

A.4. Overhead

Enhanced functionality of *HotSniper* comes with additional computations, which adds to the time required to complete many-core simulation. Figure A.6 shows simulation-time of *Sniper* simulator and *HotSniper* toolchain for various four-threaded instances of different *PARSEC* tasks [85] with *sim-small* input on simulated quad-core multi-core. For this experiment, *Hot-Sniper* operates with default 10 ms scheduling epoch and 1 ms power/thermal data generation interval. *Sniper* operates with its default parameters with *McPat* enabled.

Out of 13 *PARSEC* tasks, we use seven tasks - *blackscholes*, *bodytrack*, *canneal*, *dedup*, *streamcluster*, *swaptions* and *x264* - for evaluations. We did not use two tasks - *ferret* and *fluidanimate* - due to lack of ability to generate four-threaded instances. We did not use two tasks - *freqmine* and *vips* were not used due to unresolved errors in *PIN* binary instrumentation tool. We did not use two tasks - *facesim* and *raytrace* due to lack of availability of *sim-small* inputs. Figure A.6 shows that *HotSniper* leads to an acceptable increase of 20.5% in simulation-time on average in our host system.

Simulation-time under *HotSniper* is inversely proportional to the granularity of scheduling epoch. Figure A.7 shows decrease in simulation-time for four-threaded instance of *blackscholes* with *sim-small* input with another similar instance waiting in the queue as we move from

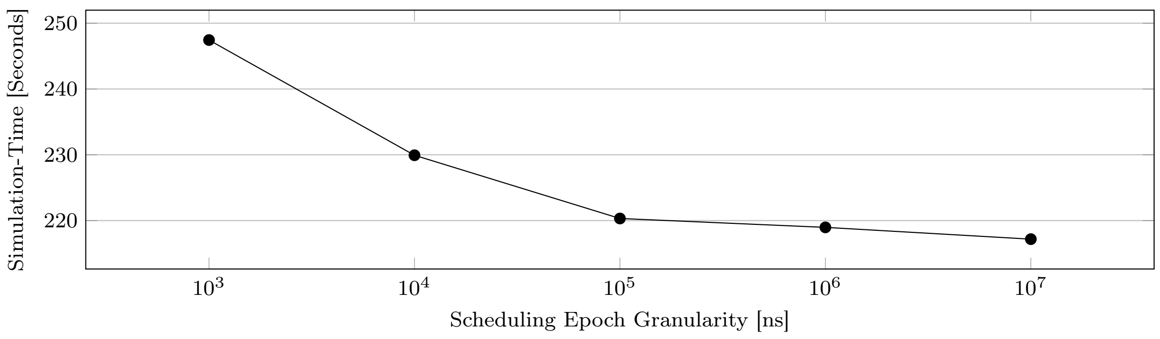


Figure A.7.: Simulation-time for four-threaded *blackscholes* using *HotSniper* with different granularity of scheduling epoch.

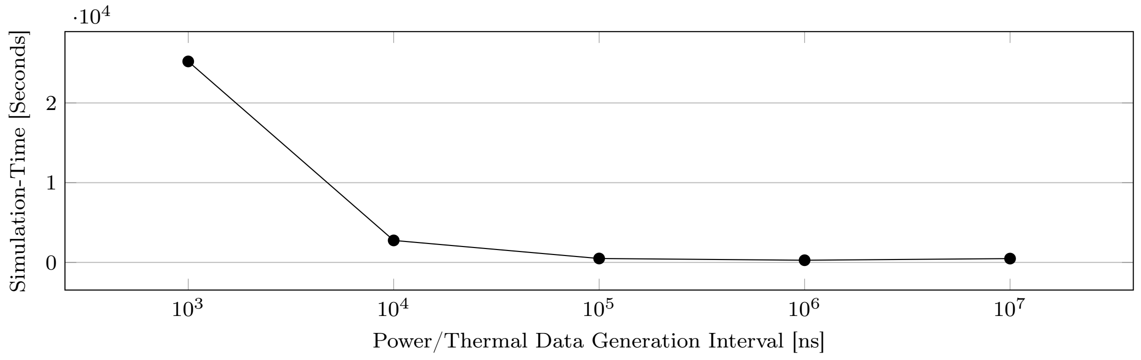


Figure A.8.: Simulation-time for four-threaded *blackscholes* using *HotSniper* with different granularity of power/thermal data generation interval.

scheduling epoch granularity of 1000 ns to 10 ms. We do not generate any power or thermal data in this experiment. Figure A.7 shows that *HotSniper* scales up well with the reduction in granularity of scheduling epoch.

Simulation-time under *HotSniper* is also inversely proportional to the granularity of the interval at which the power/thermal data is generated. Figure A.8 shows the decrease in the simulation-time for four-threaded instance of *blackscholes* with *sim-small* input as we move from interval granularity of 1000 ns to 10 ms. The scheduling epoch is set at default value of 10 ms for this experiment. Figure A.8 shows the simulation-time rises swiftly with reduction in power/thermal data generation interval.

Summary

In this appendix, we have introduced toolchain called *HotSniper* that tightly couples together *Sniper* many-core simulator, *McPat* power modeling framework and *Hotspot* temperature modeling tool. *HotSniper* allows for interval thermal simulations of multi-/many-cores, which was not previously possible. Interval thermal simulations are vital for efficient thermal-aware hardware-software codesign of many-cores especially in the domain of embedded systems.

Our toolchain also supports interval-based simulation of many-cores deployed in open systems, which is not possible out-of-the-box in original *Sniper* simulator. Additional computations introduced by *HotSniper* toolchain increase simulation-time by acceptable 20.50% on average compared to original *Sniper* simulator.

List of Figures

List of Tables

List of Abbreviations

SCC *Single-Chip Cloud Computer. 15, 18, 51, 117*

AMD *Average Manhattan Distance. 53, 54, 73–79, 81, 118, 119*

BnB *Branch and Bound. 78, 81, 107*

CMOS *Complementary Metal-Oxide-Semiconductor. 15, 104*

CPU *Central Processing Unit. 6, 12*

D-NUCA *Dynamic Non-Uniform Cache Access. 51, 63, 71, 73, 79*

DMA *Direct Memory Access. 13*

DP *Dynamic Programming. 7, 22, 30, 32, 34, 37*

DRAM *Dynamic Random-Access Memory. 51, 61, 68, 113*

DTM *Dynamic Thermal Management. 95, 102, 104, 105*

DVFS *Dynamic Voltage and Frequency Scaling. 3, 11, 15, 19, 21, 81, 83–98, 103, 104, 108, 119*

ESA *Exponentially Separable Allocation. 55–59, 61, 63, 65–67, 69, 71, 118*

EU *Execution Unit. 113*

FCFS *First-Come, First-Serve. 59, 110*

FIFO *First-In, First-Out. 58, 59, 61, 62, 77, 110*

FPGA *Field-Programmable Gate Array. 18*

FS *Full System. 91, 106*

GPP *General-Purpose Processor. 1, 6, 12*

GPU *Graphic Processing Unit. 6, 12, 13*

HPC *High-Performance Computing. 14, 74*

IC *Integrated Circuit. 1*

IFU *Instruction Fetch Unit. 113*

ILP *Instruction Level Parallelism. 4, 5, 21, 22, 24, 43*

IPC *Instruction per Cycle. 19, 21, 23, 24, 29–31, 34, 35, 43*

IPS *Instruction per Second.* 19, 95, 97, 103, 104

ISA *Instruction Set Architecture.* 11, 12, 15, 16, 18, 33, 61, 91, 106

LLC *Last-Level Cache.* 4, 5, 11, 51, 52, 61, 73, 74, 79, 118

LRU *Least Recent Used.* 61

LSU *Load Store Unit.* 113

MC *Memory Controller.* 15, 51, 54, 61, 63, 64

MIPS *Millions of Instruction per Second.* 97, 98, 119

MMU *Memory Management Unit.* 113

MPB *Message-Passing Buffer.* 15

MPI *Message-Passing Interface.* 15

MSI *Modified Shared Invalid.* 61

NoC *Network-on-Chip.* 4–6, 11, 13, 15–17, 30, 51, 52, 54, 61–64, 66–68, 71, 73, 91, 109, 114, 118

NUCA *Non-Uniform Cache Access.* 73, 79

OEIS *Online Encyclopedia of Integer Sequences.* 53

OS *Operating System.* vii, 4, 6, 9, 10, 13, 15–18, 51, 54, 61, 63, 64, 73, 83, 85, 94, 95, 107, 113

PDF *Probability Density Function.* 99–101

PE *Processing Element.* 13

PID *Proportional Integrate Derivative.* 95

PMF *Probability Mass Function.* 86, 87, 98–101

QoS *Quality of Service.* vii, ix, 7, 18, 85, 87, 94–99, 103–107, 119

RISC *Reduced Instruction Set Computer.* 12, 15, 16

RPC *Remote Procedure Call.* 13

RU *Renaming Unit.* 112, 113

S-NUCA *Static Non-Uniform Cache Access.* vii, ix, 7, 71, 73–75, 77, 79–81, 107, 108, 118, 119

SE *Syscall Emulation.* 33, 106

SMP *Symmetric Multi-Processing.* 16

TDP *Thermal Design Power.* 83, 85, 86, 89, 90, 92–95, 102–106, 119, 120

TLP *Thread Level Parallelism.* 21, 22, 24, 43, 63

VLIW *Very Long Instruction Word.* 15

Bibliography

[1] A. Pathania, S. Pagani, M. Shafique, and J. Henkel, “Power Management for Mobile Games on Asymmetric Multi-Cores,” in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2015.

[2] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, “Distributed Fair Scheduling for Many-Cores,” in *Conference on Design, Automation & Test in Europe (DATE)*, 2016.

[3] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, “Distributed Scheduling for Many-Cores Using Cooperative Game Theory,” in *Design Automation Conference (DAC)*, 2016.

[4] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, “Optimal Greedy Algorithm for Many-Core Scheduling,” *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

[5] S. Pagani, L. Bauer, Q. Chen, E. Glocker, F. Hannig, A. Herkersdorf, H. Khdr, A. Pathania, U. Schlichtmann, D. Schmitt-Landsiedel *et al.*, “Dark Silicon Management: An Integrated and Coordinated Cross-Layer Approach,” *it-Information Technology (IT)*, 2016.

[6] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, “Defragmentation of Tasks in Many-Core Architecture,” *Transactions on Architecture and Code Optimization (TACO)*, 2017.

[7] A. Pathania, H. Khdr, M. Shafique, T. Mitra, and J. Henkel, “Scalable Probabilistic Power Budgeting for Many-Cores,” in *Conference on Design, Automation & Test in Europe (DATE)*, 2017.

[8] S. Pagani, A. Pathania, M. Shafique, J.-J. Chen, and J. Henkel, “Energy Efficiency for Clustered Heterogeneous Multicores,” *Transactions on Parallel and Distributed Systems (TPDS)*, 2017.

[9] H. Khdr, S. Pagani, E. Sousa, V. Lari, A. Pathania, F. Hannig, M. Shafique, J. Teich, and J. Henkel, “Power Density-Aware Resource Management for Heterogeneous Tiled Multicores,” *Transactions on Computers (TC)*, 2017.

[10] A. Pathania and J. Henkel, “Task Scheduling for Many-Cores with S-NUCA Caches,” in *Conference on Design, Automation & Test in Europe (DATE)*, 2018.

[11] A. Pathania, H. Khdr, M. Shafique, T. Mitra, and J. Henkel, “QoS-Aware Stochastic Power Management for Many-Cores,” in *Design Automation Conference (DAC)*, 2018.

[12] M. Rapp, A. Pathania, and J. Henkel, “Pareto-Optimal Power- and Cache-Aware Task Mapping for Many-Cores with Distributed Shared Last-Level Cache,” in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2018.

[13] V. Venkataramani, A. Pathania, M. Shafique, T. Mitra, and J. Henkel, “Distributed Fair Scheduling for Many-Cores,” in *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2018.

[14] M. Shafique, S. Garg, T. Mitra, S. Parameswaran, and J. Henkel, “Dark Silicon as a Challenge for Hardware/Software Co-Design,” in *Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS)*, 2014.

[15] B. Alcott, “Jevons’ Paradox,” *Ecological Economics*, 2005.

[16] R. R. Schaller, “Moore’s Law: Past, Present and Future,” *IEEE Spectrum*, 1997.

[17] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions,” *Journal of Solid-State Circuits(JSSC)*, 1974.

[18] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation Cores: Reducing the Energy of Mature Computations,” in *Computer Architecture News (CAN)*, 2010.

[19] J. Henkel, H. Khdr, S. Pagani, and M. Shafique, “New Trends in Dark Silicon,” in *Design Automation Conference (DAC)*, 2015.

[20] N. Pinckney, K. Sewell, R. G. Dreslinski, D. Fick, T. Mudge, D. Sylvester, and D. Blaauw, “Assessing the Performance Limits of Parallelized Near-Threshold Computing,” in *Design Automation Conference (DAC)*, 2012.

[21] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel *et al.*, “Invasive Manycore Architectures,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012.

[22] A. Prakash, S. Wang, A. E. Irimiea, and T. Mitra, “Energy-Efficient Execution of Data-Parallel Applications on Heterogeneous Mobile Platforms,” in *International Conference on Computer Design (ICCD)*, 2015.

[23] J. J. Dongarra, H. W. Meuer, E. Strohmaier *et al.*, “TOP500 Supercomputer Sites,” *Supercomputer*, 1997.

[24] C. Tan, A. Kulkarni, V. Venkataramani, M. Karunaratne, T. Mitra, and L.-S. Peh, “LOCUS: Low-Power Customizable Many-Core Architecture for Wearables,” *Transactions on Embedded Computing Systems (TECS)*, 2017.

[25] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng, “General Purpose Computing on Low-Power Embedded GPUs: Has It Come of Age?” in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, 2013.

[26] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, “Integrated CPU-GPU Power Management for 3D Mobile Games,” in *Design Automation Conference (DAC)*, 2014.

[27] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra, “Power-Performance Modelling of Mobile Gaming Workloads on Heterogeneous MPSoCs,” in *Design Automation Conference (DAC)*, 2015.

[28] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, “Many-Core vs. Many-Thread Machines: Stay Away from the Valley,” *Computer Architecture Letters (CAL)*, 2009.

[29] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, “Improving GPGPU Energy-Efficiency Through Concurrent Kernel Execution and DVFS,” in *Symposium on Code Generation and Optimization (CGO)*, 2015.

[30] F. Samie, V. Tsoutsouras, L. Bauer, S. Xydis, D. Soudris, and J. Henkel, “Computation Offloading and Resource Allocation for Low-Power IoT Edge Devices,” in *World Forum on Internet of Things (WF-IoT)*, 2016.

[31] K. J. Naik, A. Jagan, and N. S. Narayana, “A Novel Algorithm for Fault Tolerant Job Scheduling and Load Balancing in Grid Computing Environment,” in *International Conference on Green Computing and Internet of Things (ICGCIoT)*, 2015.

[32] D. G. Feitelson and L. Rudolph, “Metrics and Benchmarking for Parallel Job Scheduling,” in *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1998.

[33] V. Pallipadi and A. Starikovskiy, “The Ondemand Governor,” in *Linux Symposium*, 2006.

[34] T. Mitra, “Heterogeneous Multi-core Architectures,” *Transactions on System LSI Design Methodology (T-SDLM)*, 2015.

[35] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, “Price Theory Based Power Management for Heterogeneous Multi-Cores,” *Computer Architecture News (CAN)*, 2014.

[36] M. Pricopi and T. Mitra, “Bahurupi: A Polymorphic Heterogeneous Multi-Core Architecture,” *Transactions on Architecture and Code Optimization (TACO)*, 2012.

[37] S. Penolazzi, L. Bolognino, and A. Hemani, “Energy and Performance Model of a SPARC Leon3 processor,” in *Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, 2009.

[38] J. Henkel, L. Bauer, M. Hübner, and A. Grudnitsk, “i-Core: A Run-Time Adaptive Processor for Embedded Multi-Core Systems,” in *Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2011.

[39] S. Buchwald, M. Mohr, and A. Zwinkau, “Malleable Invasive Applications,” in *Software Engineering (Workshops)*, 2015.

[40] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel, “DistRM: Distributed Resource Management for On-Chip Many-Core Systems,” in *Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2011.

[41] M. Witterauf, A. Tanase, J. Teich, V. Lari, A. Zwinkau, and G. Snelting, “Adaptive Fault Tolerance Through Invasive Computing,” in *Conference on Adaptive Hardware and Systems (AHS)*, 2015.

[42] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat, “OctoPOS: A Parallel Operating System for Invasive Computing,” in *Workshop on Systems for Future Multi-Core Architectures (SFMA)*, 2011.

[43] M. Mohr, S. Buchwald, A. Zwinkau, C. Erhardt, B. Oechslein, J. Schedel, and D. Lohmann, “Cutting Out the Middleman: OS-Level Support for X10 Activities,” in *Workshop on X10*, 2015.

[44] F. Hannig, V. Lari, S. Boppu, A. Tanase, and O. Reiche, “Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Approach,” *Transactions on Embedded Computing Systems (TECS)*, 2014.

[45] J. Heißwolf, R. König, and J. Becker, “A Scalable NoC Router Design Providing QoS Support Using Weighted Round Robin Scheduling,” in *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2012.

[46] J. Heisswolf, A. Zaib, A. Zwinkau, S. Kobbe, A. Weichslgartner, J. Teich, J. Henkel, G. Snelting, A. Herkersdorf, and J. Becker, “CAP: Communication Aware Programming,” in *Design Automation Conference (DAC)*, 2014.

[47] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang, “SelectDirectory: A Selective Directory for Cache Coherence in Many-Core Architectures,” in *Conference on Design, Automation & Test in Europe (DATE).*

[48] A. Srivatsa, S. Rheindt, T. Wild, and A. Herkersdorf, “Region Based Cache Coherence for Tiled MPSoCs,” in *System-on-Chip Conference (SOCC)*, 2017.

[49] A. Zwinkau, “A Memory Model for X10,” in *Workshop on X10*, 2016.

[50] S. Roloff, A. Pöppl, T. Schwarzer, S. Wildermann, M. Bader, M. Glaß, F. Hannig, and J. Teich, “ActorX10: An Actor Library for X10,” in *Workshop on X10*, 2016.

[51] F. Hannig, S. Roloff, G. Snelting, J. Teich, and A. Zwinkau, “Resource-Aware Programming and Simulation of MPSoC Architectures Through Extension of X10,” in *Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2011.

[52] H.-J. Bungartz, C. Riesinger, M. Schreiber, G. Snelting, and A. Zwinkau, “Invasive Computing in HPC with X10,” in *Workshop on X10*, 2013.

[53] A. Pöppl, M. Bader, T. Schwarzer, and M. Glaß, “SWE-X10: Simulating Shallow Water Waves with Lazy Activation of Patches Using ActorX10,” in *Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, 2016.

[54] J. Paul, W. Stechele, B. Oechslein, C. Erhardt, J. Schedel, D. Lohmann, W. Schröder-Preikschat, M. Kröhnert, T. Asfour, É. Sousa *et al.*, “Resource-Awareness on Heterogeneous MPSoCs for Image Processing,” *Journal of Systems Architecture (JSA)*, 2015.

[55] N. Budhdev, M. C. Chan, and T. Mitra, “PR 3: Power Efficient and Low Latency Baseband Processing for LTE Femtocells,” in *International Conference on Computer Communications (INFOCOM)*, 2018.

[56] S. Kobbe, L. Bauer, and J. Henkel, “Adaptive On-the-Fly Application Performance Modeling for Many Cores,” in *Conference on Design, Automation & Test in Europe (DATE)*, 2015.

[57] S. Pagani, J.-J. Chen, M. Shafique, and J. Henkel, “MatEx: Efficient Transient and Peak Temperature Computation for Compact Thermal Models,” in *Conference on Design, Automation & Test in Europe (DATE)*, 2015.

[58] J. Henkel, H. Bukhari, S. Garg, M. U. K. Khan, H. Khdr, F. Kriebel, U. Ogras, S. Parameswaran, and M. Shafique, “Dark Silicon: From Computation to Communication,” in *Symposium on Networks-on-Chip (NOCS)*, 2015.

[59] S. Pagani, H. Khdr, W. Munawar, J.-J. Chen, M. Shafique, M. Li, and J. Henkel, “TSP: Thermal Safe Power: Efficient Power Budgeting for Many-Core Systems in Dark Silicon,” in *Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS)*, 2014.

[60] H. Khdr, S. Pagani, M. Shafique, and J. Henkel, “Thermal Constrained Resource Management for Mixed ILP-TLP Workloads in Dark Silicon Chips,” in *Design Automation Conference (DAC)*, 2015.

[61] 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 *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2010.

[62] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl *et al.*, “The 48-core SCC Processor: The Programmer’s View,” in *Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[63] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, “Embracing Diversity in the Barrelfish Manycore Operating System,” in *Workshop on Managed Many-Core Systems (MMCS)*, vol. 27, 2008.

[64] S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe, “Early Experience with the Barrelfish OS and the Single-Chip Cloud Computer,” in *MARC Symposium*, 2011.

[65] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris, “Distributed Run-Time Resource Management for Malleable Applications on Many-Core Platforms,” in *Design Automation Conference (DAC)*, 2013.

[66] C. Ramey, “Tile-Gx100 Manycore Processor: Acceleration Interfaces and Architecture,” in *Hot Chips Symposium (HCS)*, 2011.

[67] A. Olofsson, “Epiphany-V: A 1024 Processor 64-bit RISC System-on-Chip,” in *Hot Chips Symposium (HCS)*, 2017.

[68] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich *et al.*, “An Analysis of Linux Scalability to Many Cores,” in *Operating Systems Design and Implementation (OSDI)*, 2010.

[69] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai *et al.*, “Corey: An Operating System for Many Cores,” in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[70] 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 *Symposium on Operating Systems Principles.*

[71] J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, “Resource Management in the Tessellation Manycore OS,” *Workshop on Hot Topics in Parallelism (HotPar)*, vol. 10, 2010.

[72] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable Lightweight Processors," in *International Symposium on Microarchitecture (MICRO)*, 2007.

[73] J. H. Silverman, "A Friendly Introduction to Number Theory," *Am Math Compet*, 2006.

[74] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, and D. Burger, "Multitasking Workload Scheduling on Flexible-Core Chip Multiprocessors," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[75] J. Augustine, N. Chen, E. Elkind, A. Fanelli, N. Gravin, and D. Shiryaev, "Dynamics of Profit-Sharing Games," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[76] V. Vanchinathan, "Performance Modeling of Adaptive Multi-core Architecture," Master's thesis, National University of Singapore (NUS), 2015.

[77] K. Burns and B. Hasselblatt, "The Sharkovsky Theorem: A Natural Direct Proof," *American Mathematical Monthly*, 2011.

[78] A. Stivala, P. J. Stuckey, M. G. de la Banda, M. Hermenegildo, and A. Wirth, "Lock-Free Parallel Dynamic Programming," *Journal of Parallel and Distributed Computing*, 2010.

[79] S. M. Zahedi and B. C. Lee, "REF: Resource Elasticity Fairness with Sharing Incentives for Multiprocessors," *Computer Architecture News (CAN)*, 2014.

[80] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 Simulator," *Computer Architecture News (CAN)*, 2011.

[81] K. M. Lepak, H. W. Cain, and M. H. Lipasti, "Redeeming IPC as a Performance Metric for Multithreaded Programs," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[82] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, 2000.

[83] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *Computer Architecture News (CAN)*, 2006.

[84] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The San Diego Vision Benchmark Suite," in *International Symposium on Workload Characterization (IISWC)*, 2009.

[85] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[86] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *International Symposium on Computer Architecture (ISCA)*, 1995.

[87] M. Pricopi and T. Mitra, "Task Scheduling on Adaptive Multi-Core," *Transactions on Computers (TC)*, 2014.

[88] C. S. Pabla, "Completely Fair Scheduler," *Linux Journal*, 2009.

[89] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias, "ACFS: A Completely Fair Scheduler for Asymmetric Single-ISA Multicore Systems," in *Symposium on Applied Computing (SAC)*, 2015.

[90] C. Wu, J. Li, D. Xu, P.-C. Yew, J. Li, and Z. Wang, "FPS: A Fair-Progress Process Scheduling Policy on Shared-Memory Multiprocessors," *Transactions on Parallel and Distributed Systems (TPDS)*, 2015.

[91] T. Ebi, A. Faruque, M. Abdullah, and J. Henkel, "TAPE: Thermal-Aware Agent-Based Power Economy for Multi/Many-core Architectures," in *International Conference on Computer-Aided Design (ICCAD)*, 2009.

[92] T. Sun, H. An, T. Wang, H. Zhang, and X. Sui, "CRQ-Based Fair Scheduling on Composable Multicore Architectures," in *International Conference on Supercomputing (ICS)*, 2012.

[93] A. Sembrant, D. Black-Schaffer, and E. Hagersten, "Phase Behavior in Serial and Parallel Applications," in *International Symposium on Workload Characterization (IISWC)*, 2012.

[94] H. Vandierendonck and A. Seznec, "Fairness Metrics for Multi-Threaded Processors," *Computer Architecture Letters (CAL)*, 2011.

[95] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multicore Memory Systems," *Transactions on Computer Systems (TOCS)*, 2012.

[96] W. Kubiak, "Completion Time Variance Minimization on a Single Machine is Difficult," *Operations Research Letters*, 1993.

[97] T. Baker, J. Gill, and R. Solovay, "Relativizations of the P=?NP Question," *SIAM Journal on Computing (SICOMP)*, 1975.

[98] A. Toriello and J. P. Vielma, "Fitting Piecewise Linear Continuous Functions," *European Journal of Operational Research (EJOR)*, 2012.

[99] E. Chlamtac and M. Tulsiani, "Convex Relaxations and Integrality Gaps," in *Handbook on Semidefinite, Conic and Polynomial Optimization*, 2012.

[100] K. Murota, "Submodular Function Minimization and Maximization in Discrete Convex Analysis," *Discrete Applied Mathematics*, 1984.

[101] J. Held, "Single-Chip Cloud Computer, an IA Tera-scale Research Processor," in *European Conference on Parallel Processing (Euro-Par)*, 2010.

[102] N. J. Sloane *et al.*, "The On-line Encyclopedia of Integer Sequences," 2003.

[103] E. D. Demaine and M. L. Demaine, "Jigsaw Puzzles, Edge Matching, and Polyomino Packing: Connections and Complexity," *Graphs and Combinatorics*, 2007.

[104] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila, "Smart Hill Climbing for Agile Dynamic Mapping in Many-Core Systems," in *Design Automation Conference*, 2013.

[105] M. Fattah, P. Liljeberg, J. Plosila, and H. Tenhunen, "Adjustable Contiguity of Run-Time Task Allocation in Networked Many-Core Systems," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.

[106] M. A. Al Faruque, R. Krist, and J. Henkel, "ADAM: Run-time Agent-based Distributed Application Mapping for On-Chip Communication," in *Design Automation Conference (DAC)*, 2008.

[107] K. C. Knowlton, "A Fast Storage Allocator," *Communications of the ACM*, 1965.

[108] K. Li and K. H. Cheng, "A Two Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System," in *Computer Science Conference (CSC)*, 1990.

[109] E. D. Demaine and M. Hoffmann, "Pushing Blocks is NP-Complete for Noncrossing Solution Paths," in *Canadian Conference on Computational Geometry (CCCG)*, 2001.

[110] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[111] S. Cho and L. Jin, "Managing Distributed, Shared L2 Caches Through OS-Level Page Allocation," in *International Symposium on Microarchitecture (ISCA)*, 2006.

[112] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education," in *International Symposium on Computer Architecture (ISCA)*, 2004.

[113] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," *Computer Architecture News (CAN)*, 2012.

[114] J. Ng, X. Wang, A. K. Singh, and T. Mak, "DeFrag: Defragmentation for Efficient Runtime Resource Allocation in NoC-Based Many-Core Systems," in *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2015.

[115] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, "Multi-Core Cache Hierarchies," *Synthesis Lectures on Computer Architecture*, 2011.

[116] S. Das and H. K. Kapoor, "Exploration of Migration and Replacement Policies for Dynamic NUCA over Tiled CMPs," in *International Conference on VLSI Design (VLSID)*, 2015.

[117] H. Kim, P. Ghoshal, B. Grot, P. V. Gratz, and D. A. Jiménez, "Reducing Network-on-Chip Energy Consumption Through Spatial Locality Speculation," in *International Symposium on Networks-on-Chip (NOCS)*, 2011.

[118] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry *et al.*, "Scheduling Threads for Constructive Cache Sharing on CMPs," in *Symposium on Parallel Algorithms and Architectures (SPAA)*, 2007.

[119] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends," in *Design Automation Conference*, 2013.

[120] M. A. Bender, D. P. Bunde, E. D. Demaine, S. P. Fekete, V. J. Leung, H. Meijer, and C. A. Phillips, "Communication-Aware Processor Allocation for Supercomputers: Finding Point Sets of Small Average Distance," *Algorithmica*, 2008.

[121] E. D. Demaine, S. P. Fekete, G. Rote, N. Schweer, D. Schymura, and M. Zelke, "Integer Point Sets Minimizing Average Pairwise L1 Distance: What is the Optimal Shape of a Town?" *Computational Geometry*, 2011.

[122] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Spring Joint Computer Conference (SJCC)*, 1967.

[123] E. L. Lawler and D. E. Wood, "Branch-and-Bound Methods: A Survey," *Operations Research*, 1966.

[124] E. Carvalho, N. Calazans, and F. Moraes, "Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs," in *International Workshop on Rapid System Prototyping (RSP)*, 2007.

[125] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical Power Management for Asymmetric Multi-Core in Dark Silicon Era," in *Design Automation Conference (DAC)*, 2013.

[126] A. Das, A. Kumar, B. Veeravalli, R. Shafik, G. Merrett, and B. Al-Hashimi, "Workload Uncertainty Characterization and Adaptive Frequency Scaling for Energy Minimization of Embedded Systems," in *Conference on Design, Automation & Test in Europe (DATE)*, 2015.

[127] K. Kang, J. Kim, S. Yoo, and C.-M. Kyung, "Temperature-Aware Integrated DVFS and Power Gating for Executing Tasks with Runtime Distribution," *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2010.

[128] T. Mudge, "Power: A First-Class Architectural Design Constraint," *Computer*, 2001.

[129] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in *International Symposium on Microarchitecture*.

[130] J. Sartori and R. Kumar, "Three Scalable Approaches to Improving Many-Core Throughput for a Given Peak Power Budget," in *International Conference on High Performance Computing (HiPC)*, 2009.

[131] S. Fan, S. M. Zahedi, and B. C. Lee, "The Computational Sprinting Game," in *Operating Systems Review (OSR)*, 2016.

[132] Z. Li and B. Li, "Probabilistic Power Management for Wireless Ad Hoc Networks," *Mobile Networks and Applications*, 2005.

[133] F. Cerqueira and B. Brandenburg, "A Comparison of Scheduling Latency in Linux, PREEMPT-RT, and LITMUS RT," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2013.

[134] P. S. Mann, *Introductory Statistics*, 2007.

[135] B. Rountree, D. K. Lowenthal, M. Schulz, and B. R. De Supinski, "Practical Performance Prediction under Dynamic Voltage Frequency Scaling," in *International Green Computing Conference (IGCC)*, 2011.

[136] Y. H. Wang, "On the Number of Successes in Independent Trials," *Statistica Sinica*, 1993.

[137] I. Ukhov, P. Eles, and Z. Peng, "Probabilistic Analysis of Power and Temperature under Process Variation for Electronic System Design," *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2014.

[138] W. J. Cody, "Rational Chebyshev Approximations for the Error Function," *Mathematics of Computation*, 1969.

[139] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *International Symposium on Microarchitecture (ISCA)*, 2009.

[140] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," in *International conference on Autonomic Computing (ICCAC)*, 2010.

[141] A.-M. Rahmani, M.-H. Haghbayan, A. Kanduri, A. Y. Weldezion, P. Liljeberg, J. Plosila, A. Jantsch, and H. Tenhunen, "Dynamic Power Management for Many-Core Platforms in the Dark Silicon Era: A Multi-Objective Control Approach," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2015.

[142] K. Ma and X. Wang, "PGCapping: Exploiting Power Gating for Power Capping and Core Lifetime Balancing in CMPs," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[143] Z. Chen and D. Marculescu, "Distributed Reinforcement Learning for Power Limited Many-Core System Performance Optimization," in *Conference on Design, Automation & Test in Europe (DATE)*, 2015.

[144] A. K. Mishra, S. Srikantaiah, M. Kandemir, and C. R. Das, "CPM in CMPs: Coordinated Power Management in Chip-Multiprocessors," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[145] A. Weichslgartner, D. Gangadharan, S. Wildermann, M. Glaß, and J. Teich, "DAARM: Design-Time Application Analysis and Run-Time Mapping for Predictable Execution in Many-Core Systems," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2014.

[146] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[147] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *SIGPLAN Notices*, 2005.

[148] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval Simulation: Raising the Level of Abstraction in Architectural Simulation," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[149] A. Florea, C. Buduleci, R. Chiş, A. Gellert, and L. Vinţan, "Enhancing the Sniper Simulator with Thermal Measurement," in *International Conference on System Theory, Control and Computing (ICSTCC)*, 2014.

[150] M. R. Stan, K. Skadron, M. Barcella, W. Huang, K. Sankaranarayanan, and S. Velusamy, "Hotspot: A Dynamic Compact Thermal Model at the Processor-Architecture Level," *Microelectronics Journal*, 2003.