Prefetching on the Cray-T3E: A Model and its evaluation
Technical Report No. 26/97

Matthias M. Müller*, Thomas M. Warschko and Walter F. Tichy
University of Karlsruhe, Dept. of Informatics
Am Fasanengarten 5, 76 128 Karlsruhe, Germany
e-mail: {muellerm,warschko,tichy}@ira.uka.de

10th December 1997

Abstract

In many parallel applications, network latency causes a dramatic loss in processor utilization. This paper examines software controlled access pipelining (SCAP) as a technique for hiding network latency. An analytic model of SCAP briefly describes basic operation techniques and performance improvements. Results are quantified with benchmarks on the Cray-T3E. The benchmarks used are Jacobi-iteration, parts of the Livermore Loop kernels, and others representing six different parallel algorithm classes. These were parallelized and optimized by hand to show the performance tradeoff of severals pipelining techniques. Our results show that SCAP on the Cray-T3E improves performance compared to a blocking execution by a factor of 2.1 to 38. It also got a performance speed-up against HPF of at least 12% to a factor of 3.1 dependent on the algorithm class.

1 Introduction

As microprocessors get faster and the gap between computation and communication speeds widens, network latency becomes the dominant factor of the execution time of fine-grained parallel programs. Given a 300 MHz clock, a 1.5μs latency corresponds to 450 clock cycles as is the case for the Cray-T3E. Thus, instead of a single communication operation one could perform 450 arithmetic instructions. This situation becomes worse by a factor of up to 10 once software overhead is factored in. If, however, the parallel machine is capable of performing communication and computation concurrently, the loss in efficiency can be reduced by overlapping several communication requests.

Little is known about the effects of latency hiding applied to communication networks in massively parallel computers with distributed memory. This paper reports on experiments on the Cray-T3E that quantify the effects of latency hiding on real programs, namely parallel versions of the Livermore Loops, Jacobi-iteration, and a few others.

The basic latency hiding technique is discussed in Section 2 combined with an overview of the analytic network model. Section 3 introduces the architecture of the T3E and compares it to the assumptions made in Section 2. Section 4 presents parallel algorithm classes and their instantiation by the benchmark set. Performance results are discussed in Section 5. Section 6 presents conclusions and future work.

2 SCAP

2.1 The Technique

In fine-grained parallel applications as in most other parallel applications latency prevents fast access to non-local memory. This work targets latency hiding through both overlapping computation and communication by splitting non-local memory access into prefetch and access.

2.1.1 Network requirements

We distinguish *blocking* and *overlapping* networks (see figure 1):

Blocking Networks: The Processor stalls until the desired remote value arrives. Hence, there is no

*Supported by the Graduiertenkolleg Karlsruhe 'Beherrschbarkeit komplexer Systeme'

way other than task switching to overlap different communication requests. The processor delay equals the latency of the underlying network.

(C_n,$\triangle t_n$)-Overlapping Networks: The network is able to issue a communication request every $\triangle t_n$ cycles. It can handle up to C_n operations per processor in parallel. C_n and $\triangle t_n$ are explained in detail in section 2.2.

Furthermore, the (C_n,$\triangle t_n$)-overlapping network should serve the communication requests of each processor in FIFO order to achieve maximum latency hiding capabilities because the value prefetched first is accessed first.

2.1.2 Basic Operation of SCAP

The basic operation of SCAP is illustrated using the following simple forall-statement:

```
FORALL i = 0..N-1 DO
  A[i] := B[q(i)];
END
```

The program fragment updates array A in parallel, indexing array B with permutation q. A parallelizing compiler maps the problem size N onto P real processors. This technique is called *virtualization*. Hence, each processor emulates $V = \lceil \frac{N}{P} \rceil$ virtual processors within a virtualization loop. Both A and B are distributed over the P processors. Since the value of $q(i)$ can not be determined at compile time the compiler has to insert remote memory accesses. The virtualization of the program fragment is as follows:

```
        /* Forall processors in parallel */
FORALL j = 0..P-1 DO
        /* Simulate V virtual processors */
  FOR k=j*V TO (j+1)*V-1 DO
        /* Calculate remote address */
    a1 := calculate_address(B[q(k)]);
        /* Read remote data element */
    A[k] := remote_read (a1);
  END
END
```

In the worst case, every processor issues V non-local memory accesses. These stall the processor if the network can not serve the desired values fast enough. Hence, execution time of this loop is at least V times the network latency. The following transformation of the loop shows how communication and computation can be overlapped:

```
FORALL j=0...p-1 DO
  FOR k=j*V TO (j+1)*V-1 DO
        /* Calculate remote address */
    a1 := calculate_address ( B[q(k)] );
        /* Start read request */
    prefetch(a1);
  END

  FOR k=j*V TO (j+1)*V-1 DO
        /* Recalculate remote address for simplicity */
    a1 := calculate_address ( B[q(k)] );
        /* Access data element */
    A[k] := access(a1);
  END
END
```

In this transformation, the main loop is split into two instances: a prefetch and an access (or calculation) loop. Instead of stalling on a remote memory access as in blocking networks, the processor issues remote memory requests. After the prefetch loop is executed the calculation loop accesses non-local memory without waiting time (if the data is already present !). In the best case, program speed-up is about $(V-1)$ times the network latency because there is at most one waiting period (arrival of first data item) which is bridged with subsequent communication requests compared to V waiting times in a blocking network. The nature of this speed-up is explained below. However, the double address calculation and the loop cost time, also. To reduce overhead of address calculation we assume a global address space where either network or processor is able to compute local addresses efficiently.

2.2 The analytic network-model

We only give a short overview of the model. A complete discussion is given in [10].
First of all, the execution time T of a parallel algorithm can be written as the sum of computation time and communication time: $T = T_{\text{Computation}} + T_{\text{Communication}}$. While $T_{\text{Computation}}$ stays constant for a fixed algorithm, $T_{\text{Communication}}$ depends on the underlying network. Communication time for blocking networks $T_{\text{Blocking}}(k)$ is

$$T_{\text{Blocking}}(k) = k * T_{\text{Latency}}, \qquad (1)$$

because each of the k non-local memory accesses lasts one network latency.
The case for (C_n,$\triangle t_n$)-overlapping networks is not that easy because each communication request depends on the ones before:

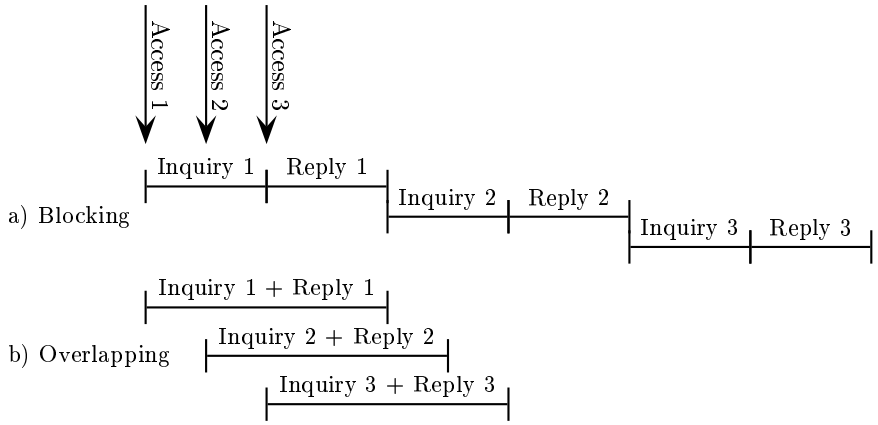

Figure 1: Different network models.

$$T_{\text{Overlapping}}(k) = T_{\text{Prefetch}}(k) + \sum_{i=1}^{k} T_{\text{Wait}_i}, \qquad (2)$$
$$0 \leq T_{\text{Wait}_i} \leq T_{\text{Latency}}.$$

$T_{\text{Prefetch}}(k)$ denotes the time for the prefetch loop and T_{Wait_i} is the waiting time for communication request i. While $T_{\text{Prefetch}}(k)$ depends only on address calculation designation of T_{Wait_i} involves the parameters in table 1.

a) Application parameters

Parameter	Description
$\triangle t_p$	Time spent in one iteration of the prefetch loop
$\triangle t_c$	Time spent in one iteration of the calculation loop

b) Hardware parameters

Parameter	Description
$\triangle t_n$	Network issue time
C_p	Size of processor prefetch buffer
C_n	Capacity of network

Table 1: Hardware and application specific parameters.

$\triangle t_p$ and $\triangle t_c$ vary for different applications and have to be measured for each new program. $\triangle t_n$, C_p, and C_n characterize the network, hence they are fixed for a given architecture. Network capacity C_n indicates the amount of communication requests the network can overlap. C_n and $\triangle t_n$ are connected over T_{Latency} because

$$T_{\text{Latency}} = C_n * \triangle t_n. \qquad (3)$$

The prefetch buffer decouples processor from network such that the processor can issue more prefetch instructions than network can overlap. Consequently, C_p is assumed to be much larger than C_n ($C_p \gg C_n$). The relationship of $\triangle t_n$, $\triangle t_p$, and $\triangle t_c$ on one side and of C_p, C_n, and k on the other side introduces six different network model classes. They are summarized in table 2.

	$\mathbf{0 < \triangle t_n \leq \triangle t_p \leq \triangle t_c}$	$\mathbf{0 < \triangle t_p < \triangle t_n}$
$\mathbf{0 \leq k \leq C_n \ll C_p}$	Class 1	Class 4
$\mathbf{C_n < k \leq C_p}$	Class 2	Class 5
$\mathbf{C_p < k}$	Class 3	Class 6

Table 2: Different processor network models.

The rows indicate the different amount of communication compared to prefetch buffer and network capacity. The columns distinguish different relationships between network issue rate and processor prefetch time. Parameter $\triangle t_c$ covers not only address calculation but also some computation which uses the requested non-local memory content. Consequently, it is $\triangle t_c > \triangle t_p$ which does not affect the first column. However, it incorporates additional classes to the second but they are further of no interest. The discussion of the entire second column and its subclasses can be found in [10].
Now, we can calculate T_{Wait_i} in (2) for the network model classes one to three.

Class 1: ($0 < \triangle t_n \leq \triangle t_p \leq \triangle t_c$, $0 \leq k \leq C_n \ll C_p$, see figure 2)

There are fewer communication requests than the

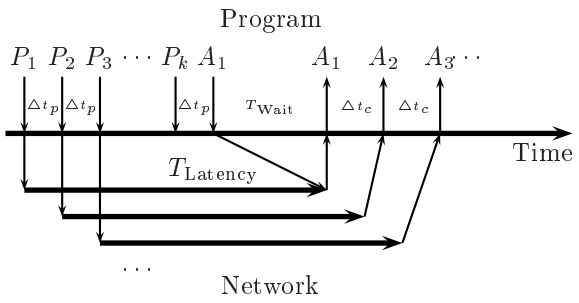

Figure 2: Behavior of SCAP, class 1.

network can overlap ($k \leq C_n$). Hence, after executing the prefetch loop the processor stalls waiting for the first non-local access. All subsequent accesses to non-local values have no delay because of $\triangle t_c \geq \triangle t_n$. Consequently:

$$T_{\text{Wait}_i} = \begin{cases} T_{\text{Latency}} - k * \triangle t_p & \text{if } i = 1 \\ 0 & \text{if } 1 < i \leq k \end{cases} \quad (4)$$

Class 2: ($0 < \triangle t_n \leq \triangle t_p \leq \triangle t_c,\ C_n < k \leq C_p$, see figure 3)

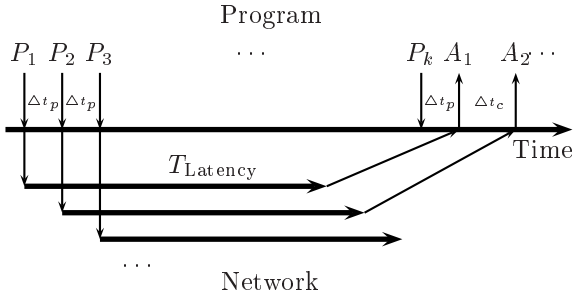

Figure 3: Behavior of SCAP, class 2.

The number of communication requests is larger than the network capacity C_n and smaller than the prefetch buffer size C_p. Hence, there are no waiting times for non-local memory accesses during the calculation loop because of $\triangle t_c \geq \triangle t_n$. Therefore :

$$T_{\text{Wait}_i} = 0, \quad 1 \leq i \leq k \quad (5)$$

Class 3: ($0 < \triangle t_n \leq \triangle t_p \leq \triangle t_c,\ C_p < k$, see figure 4)

The difference to case 2 lies in the number of communication requests which exceeds the size of the prefetch buffer ($k > C_p$). Thus, SCAP changes from a two to a three step loop execution. In the first loop, C_p values are prefetched. In each iteration of the second loop, one entry of the prefetch

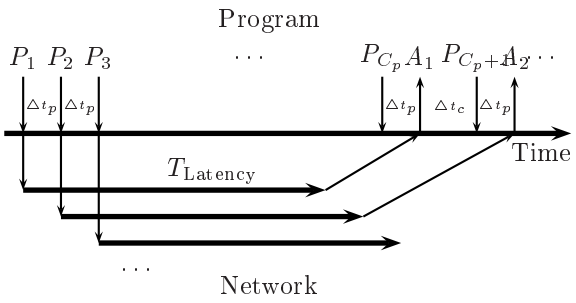

Figure 4: Behavior of SCAP, class 3.

buffer is accessed, the content is used, and another communication request is issued with the empty entry. This loop is executed $k - C_p$ times. The third loop accesses the remaining C_p non-local values.

$\triangle t_c + \triangle t_p \geq \triangle t_n$ because of $\triangle t_c \geq \triangle t_p \geq \triangle t_n$ and therefore, there are no waiting times:

$$T_{\text{Wait}_i} = 0, \quad 1 \leq i \leq k \quad (6)$$

Now, the waiting times are inserted in (2) to calculate the communication time of SCAP, assuming $T_{\text{Prefetch}}(k)$ equals $k * \triangle t_p$.

Class 1: Time of SCAP covers the prefetch loop and waiting time for the first non-local memory access (4):

$$T_{\text{Overlapping},1}(k) = T_{\text{Latency}} \quad (7)$$

Class 2 and 3: As there are no waiting times (see (5) and (6)), time for communication equals the overhead of the prefetch loop:

$$T_{\text{Overlapped},2,3}(k) = k * \triangle t_p \quad (8)$$

The runtime effort of SCAP against blocking execution is summarized below.

Class 1: The difference between T_{Blocking} (1) and $T_{\text{Overlapping},1}$ (7) is as follows:

$$T_{\text{diff}} = (k-1) * T_{\text{Latency}} \quad (9)$$

SCAP runtime is $(k-1) \times T_{\text{Latency}}$ faster than blocking execution.

Classes 2 and 3:

$$T_{\text{diff}} = (1 - \frac{1}{c}) * k * T_{\text{Latency}}, \quad 1 < c < C_n \quad (10)$$

with $c = \frac{T_{\text{Latency}}}{\triangle t_p}$. In both cases the advantage of SCAP is limited by the network capacity C_n. The case $c = 1$ is omitted because this results in a blocking network. In networks with larger capacities, $1 - \frac{1}{c}$ denotes the speed-up in percentage. For example with $c = 2$, SCAP shows a reduction of communication time of 50% (90% with $c = 10$).

After presenting SCAP and its theoretical runtime improvements the next section deals with the architecture of the Cray-T3E and covers its classification in the above mentioned classes.

3 The Cray-T3E

3.1 Architectural Overview

The T3E consists up to 2048 DEC Alpha EV5 21164 processors running at 300 MHz. They are connected with a 3D-torus network. The net is decoupled from the processors at a speed of 75 MHz [8] with overlapped communication. Each link has a bandwidth of approximately 500 MB/s resulting in a 3 GB/s transfer rate for a single node.
The network interface consists of 512 user and 128 system E-registers, memory mapped in the address space of each processor. They are the only way to perform data transfer between distinct nodes in the network. Reads and writes between E-registers and global memory are called *gets* and *puts*. To load a global memory content into the processor, a *get* and a subsequent read of the E-register has to be executed. The latter operation stalls the processor until the value arrives. This is achieved in hardware according to the implicit state of the E-register. On a *put* the local memory of a remote node is modified and the cache is updated [6]. Hence, the T3E implements a *global address space* with locally consistent memory.

3.2 Characteristic Parameters

The model parameters of the T3E from table 1 are given below.

$$\begin{aligned} \triangle t_p &= 160ns \\ \triangle t_c &= 107ns \\ \triangle t_n &= 13.3ns \\ C_p &= 480 \text{ entries} \\ C_n &= 56 \text{ entries} \end{aligned}$$

We derived the first two from measurements. The last three were taken from literature [8]. With these parameters, we got $T_{\text{Latency}} = 1489.6ns$ which differs only 0.5% from measurement.

For the classification of the T3E, there is $\triangle t_n < \triangle t_p$ for all applications. $\triangle t_c < \triangle t_p$ in contrast to the model adds only some waiting times but does not affect the classification. Consequently, the Cray-T3E can be graded into the first three classes of table 2.

4 Benchmarks and their Implementations

The classification of the parallel algorithms used is due to the relationship of communication to computation time and due to the communication pattern. Table 3 gives an overview.

	$T_{\text{Communication}} < T_{\text{Computation}}$	$T_{\text{Communication}} \sim T_{\text{Computation}}$
Reduction	LL3 (32)	LL5 (32)
Indexed Arrays	LL1 (32)	Data-transfer (2), Rotate (32)
Indirect Indexed Arrays	-	LL13 (64)
2D-grid	Jacobi (64)	-

Table 3: Parallel algorithm classes.

The columns present the different amount of communication while the rows focus on the communication pattern. The number in brackets show the quantity of PEs which took part in the calculation. We chose the algorithms from table 3 because they are simulated and discussed in [10] which presents a detailed discussion of them, also. These algorithms are representatives for different algorithm classes and show the principle SCAP behavior of their class.
LL1, LL3, LL5 and LL13 are parallelized versions of the corresponding Livermore loops. JACOBI implements a 2d-grid nearest neighborhood calculation. ROTATE is a cyclic shift of an integer array and DATA-TRANSFER copies memory blocks from one to an adjacent node.
Where possible, we parallelized the algorithms in five different ways:

Blocking: There is only one communication request at a time. As one E-register acts on one data element there is only one E-register in use.

SCAP: Up to 128 communication requests are used in parallel. If there are more than 128 non-local memory accesses SCAP changes to a three step loop execution.

Vector (-SCAP): 8 E-registers can be combined to a vector. Each time message aggregation is possible vector communication is preferred. There are up to 64 vectors (=256 E-registers) used in parallel to obtain a maximum sustained data transfer.

Shared memory library: (Shmem) All communication is done with the Cray standard shared memory library functions.

HPF: As a comparison to an existing data parallel compiler the executables of the Portland Group HPF-compiler version 2.2 [9] are considered. This seemed interesting to us as SCAP is a constructive transformation of parallel algorithms and it is going to be integrated into a parallelizing compiler.

For a detailed description of the Livermore Loops and their parallelization see [1, 10]. The first four of the above versions are coded by hand in C and compiled with the *-O3* command line option. The options for the HPF compilation are *-Msmp* and *-O2*. Most of the HPF-versions are instrumented with the HPF-directive `!HPF$ independent, on home(...)` which results in a parallelization of the corresponding do- or forall-loop. We iterated each test one million times and measured runtime with the Unix `clock` function.

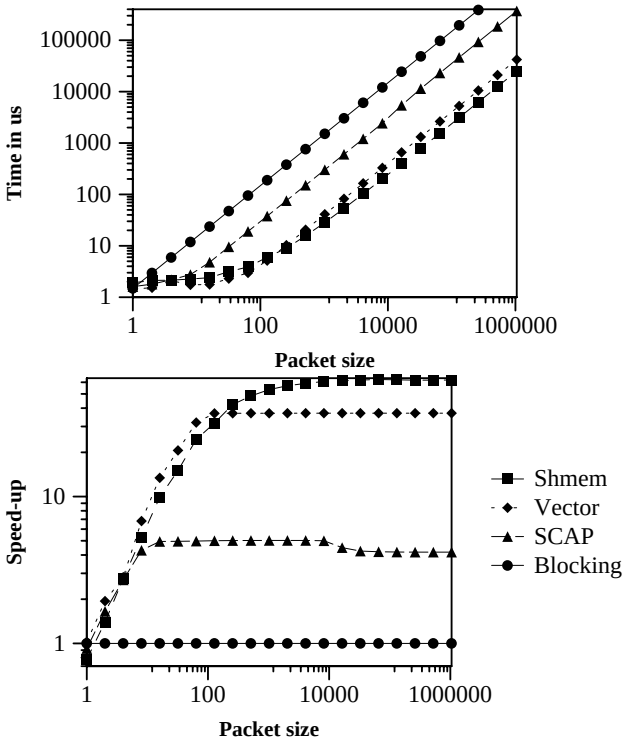

Figure 5: Benchmark: DATA-TRANSFER. Packet sizes are in units of integers (8 Byte).

5 Results

The runtimes of the different versions of each benchmark are compared to blocking execution. We also show the ratio of the approximated and measured runtimes of SCAP and blocking execution. Approximation was done with respect to our model. The discussion of each benchmark includes three plots. The first one shows the runtimes, the second one presents the relative performance compared to blocking execution, and the last one shows the ratio of approximation and measurement. In the speed-up plot, a number less than one indicates a slow-down.

5.1 Data-Transfer

The different versions of DATA-TRANSFER behave as expected (see figure 5). SCAP performs five times better than Blocking. Vector and Shmem get a relative speed-up of 37 and 62 compared to Blocking. SCAP improves performance just with overlapping communication requests. The improvement of Shmem against Vector seems due to the heavily optimized shared memory library of Cray (which we were not able to reproduce).

The model approximates runtime of Blocking in a range of 0.5% (see figure 6). Approximation of SCAP lies in a 10% range of the actual runtimes. The kink at packet size 128 shows the change from a two loop to a three loop execution of SCAP. The good approximation of Blocking is due the small amount of computation and easy communication structure in this benchmark.

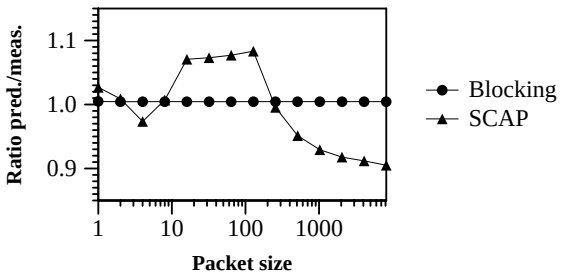

Figure 6: Approximation of DATA-TRANSFER.

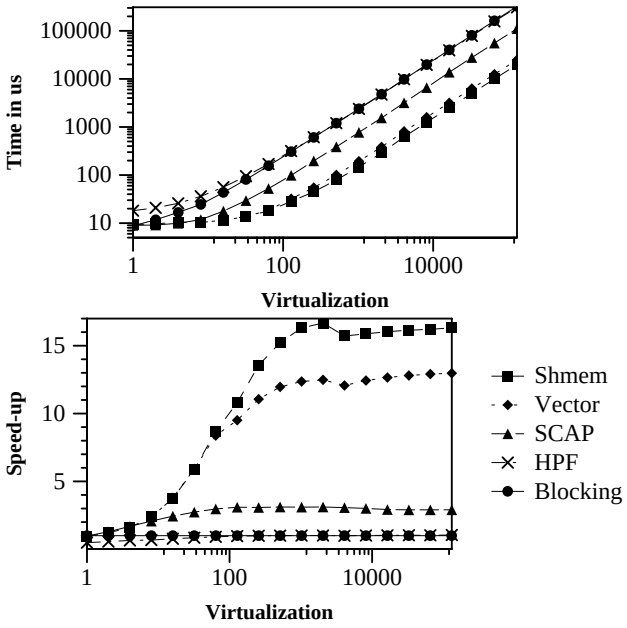

Figure 7: Benchmark: Rotate

5.2 Rotate

Figure 7 shows the different versions of ROTATE. The maximum achieved performance speed-ups of Shmem, Vector, SCAP and HPF in relation to Blocking are 16.6, 12.9, 3.1 and 1.0 respectively. As in DATA-TRANSFER communication increases with virtualization because the arrays are distributed cyclicly. Therefore, the high speed-up numbers are expected. The reason for the kink in the speed-up of Shmem and Vector at virtualization 4096 is not known, so far. It seems to be hardware related because two different implementations are affected.

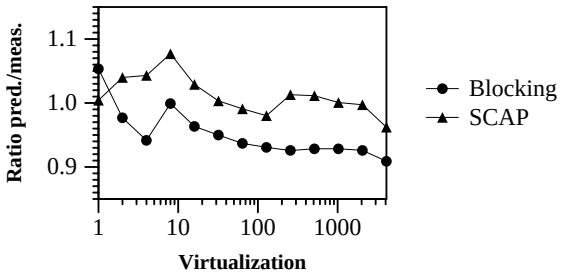

Figure 8: Approximation of ROTATE.

Both approximations (see figure 8) lie within a 10% range of the measured runtimes. This is not surprising as the communication structure of ROTATE is simple (only communication with one PE) and the proportion of computation is still very small.

5.3 Jacobi

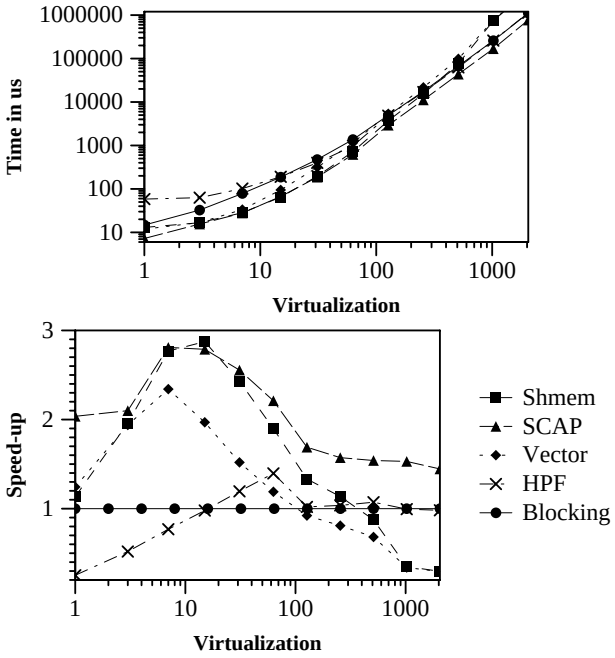

Figure 9: Benchmark: Jacobi

Figure 9 presents runtime and program speed-up of JACOBI. For large problem sizes all versions have nearly the same runtimes because computation increases with the square of virtualization whereas communication increases linearly. Shmem, Vector, SCAP and HPF perform better than Blocking (factor of 2.8, 2.3, 2.8, and 1.4 respectively) only for small virtualizations. Later, computation gets the dominant factor and advantages of better communication primitives decreases. In contrast to DATA-TRANSFER and ROTATE, JACOBI has a 2d-grid neighborhood communication which behaves rather different than left-right communication. The slow-down of Shmem and Vector is due to communication because they differ from SCAP in the way communication is done.

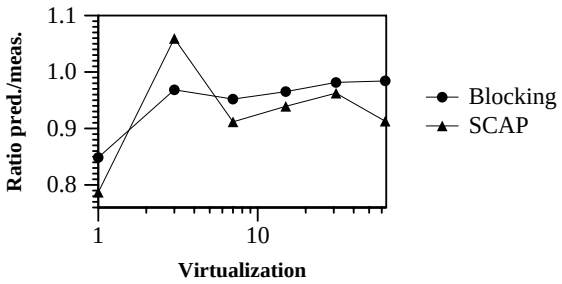

Figure 10: Approximation of JACOBI.

Compared to the benchmarks before approximation is not as close as the ones before (see figure 10). It performs worse than 20% for virtualization of 1. Later, estimation of Blocking lies within 8% and SCAP swings from 6% to -9% of the measured time.

5.4 LL1

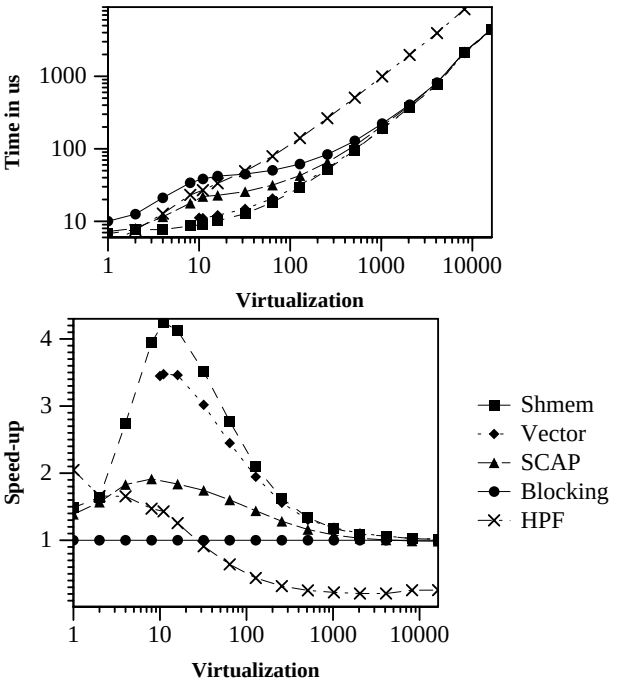

Figure 11: Benchmark: LL1

With runtime of LL1 (see figure 11), the at most 11 non-local memory references can be seen. They are reached at virtualization of 11. At this point, Shmem, Vector and SCAP behave 4.3, 3.4, and 1.9 times better than Blocking. Later, they show the same behavior as the latter one whereas HPF decreases to a slow-down of 5. The relative performances are as expected because SCAP gets its best relative speed-up at virtualizations with highest proportion of communication.
Apart from small virtualizations ($\leq$ 4), approximations of SCAP and Blocking are near the measurements ($\leq$ 4% for both versions) due to the constant amount of communication and its easy structure (see figure 12).

5.5 LL3

The difference from LL3 to LL1 is the constant amount of communication for *all* virtualizations as it

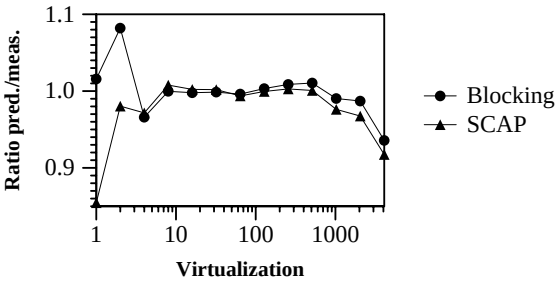

Figure 12: Approximation of LL1.

is a reduction over the PEs which took part at the computation.

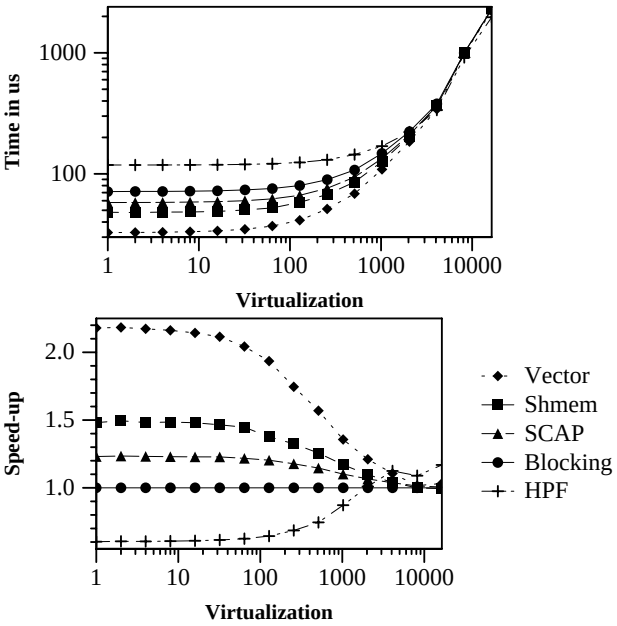

Figure 13: Benchmark: LL3

Hence, the different runtimes and relatives performances (see figure 13) for small virtualizations are not surprising. Vector, Shmem, and SCAP behave 2.2, 1.5, and 1.2 times better than Blocking (HPF is 35% slower). Like LL1, LL3 is dominated later by computation explaining the same performance results. Protrusion of Vector compared to Shmem is due to the higher *fan-in* of 8 (2 for Shmem) of the reduction. Therefore, local results are fetched with *one* vector operation. SCAP acts with the same fan-in as Vector.
Approximation of Blocking is done with a deviation of at most 5.5% (see figure 14). The difference of SCAP is slightly larger ($\leq$ 9%). So far, we can not explain the stepping in the SCAP ratio plot. As the amount of PEs is not varied and communication stays constant

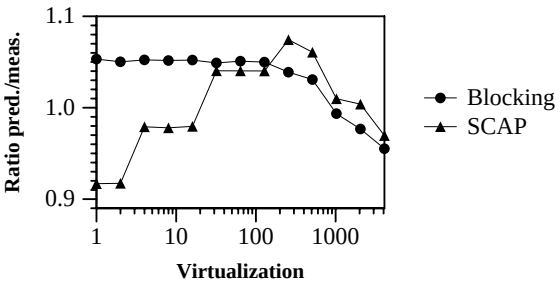

Figure 14: Approximation of LL3.

estimation of SCAP works well.

5.6 LL5

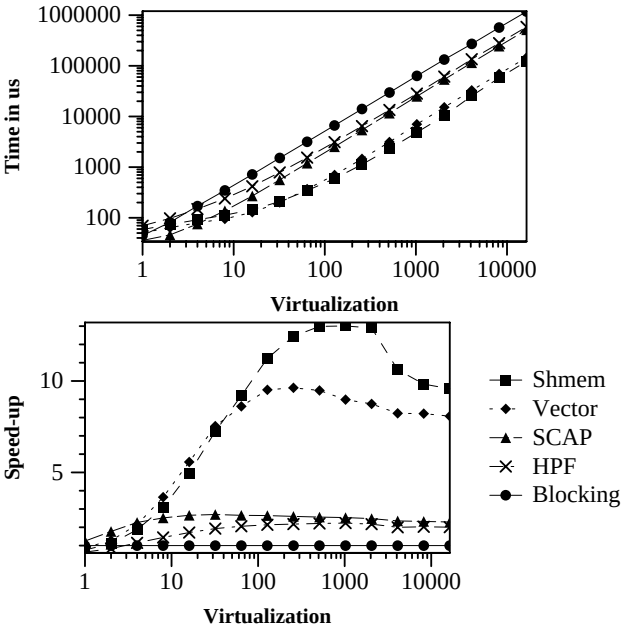

Figure 15: Benchmark: LL5

The behavior of the different versions of LL5 are shown in figure 15. Shmem, Vector, SCAP, and HPF behave 13.0, 9.6, 2.6, and 2.2 times better than Blocking. In contrast to LL3, LL5 involves each local element in the reduction. Hence, communication depends on problem size and therefore, SCAP performs better than Blocking for lager virtualizations.

Runtime approximation of Blocking is not worse than 15% (see figure 16). The results for SCAP are encouraging as they are in a range of 10% relative to the measurements despite the high amount of communication.

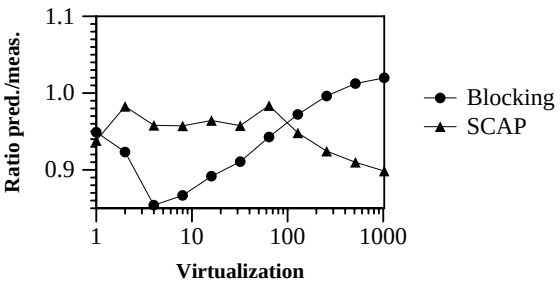

Figure 16: Approximation of LL5.

5.7 LL13

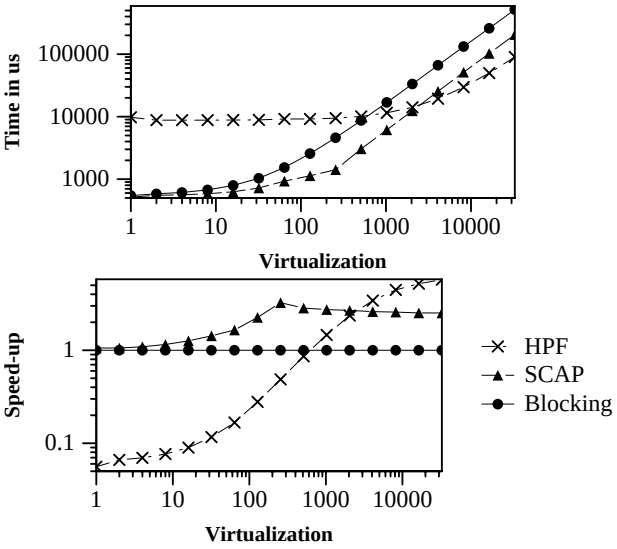

Figure 17: Benchmark: LL13

LL13 is the only benchmark with an irregular communication pattern. Therefore, Vector and Shmem were omitted. HPF and SCAP behave 5.7 and 3.2 times better than Blocking, respectively. Advantage of HPF for large virtualizations is due to *inspector-executor* ([3]) which recognizes local array elements and fetches them effectively. SCAP has no runtime check and gets all elements with E-registers which is time consuming for large virtualizations. For the future, it is interesting to enhance SCAP with a little runtime check for local array elements and to compare this version to HPF. The disadvantage of inspector-executor is the large overhead decreasing runtime for small virtualizations (about 9 times slower than SCAP !).

Blocking is estimated within a range of 10% except for virtualization 1 and 2 (see figure 18). For virtualizations between 4 and 64 SCAP approximation is not worse than 10%. The error for larger ones is explained

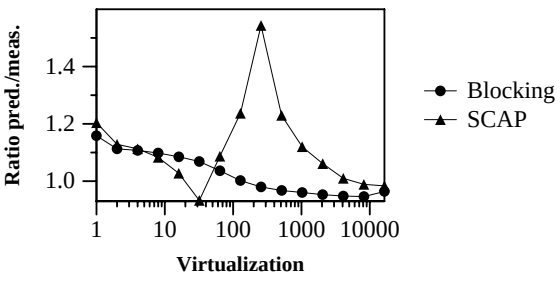

Figure 18: Approximation of LL13.

with the maximum speed-up of SCAP at virtualization of 256 which could not be detected by the model and seems to be hardware related.

6 Conclusions and Future work

This paper introduced SCAP as a constructive transformation rule to decrease communication costs in data parallel applications. This is done by overlapping both communication and computation with parallel communication requests. Our work distinguishes to other prefetching papers as our target architecture implements a global address space and data distribution lies in the response of the programmer and not the system. As we know data distribution we can prefetch effectively and data cannot be invalidated by others. This decreases network utilization as there is no additional net traffic caused by e.g. the virtual address space.

Our model achieved runtime approximations on the T3E in a range of 10% (with some explained exceptions). We presented a transformation rule which is easy to implement and which performs better than HPF in six of our seven benchmarks. As we target data parallel programs, SCAP is the technique to improve performance of HPF. It is true that shared memory programs are very fast compared to platform independent implementations like HPF but SCAP and its simple communication mechanism is an example for both platform independent modeling and good performance.

As a plan for the near future, SCAP is going to be implemented in an HPF or HPF-subset compiler to show the possibilities of an automatic transformation compared to hand-written code and with respect to existing compilers. This work will address a runtime check for local elements to improve runtime for irregular communication patterns, an extension for vector prefetching and an advanced model for runtime approximation.

References

[1] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore loops. *Parallel Computing*, 7(2):163–185, June 1988.

[2] High Performance Fortran Forum. *High Performance Fortran Language Specification 1.1*, November 1994.

[3] Charles Koelbel and Piyush Mehrotra. Supporting shared data structures on distributed memory architectures. In *Proc. of the 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPOPP*, pages 177–186, March 1990.

[4] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[5] Michael Metcalf and John Ker Reid. *Fortran 90 explained*. Oxford science publications. Oxford University Press, Walton Street, Oxford OX2 6DP, UK, reprinted with corrections edition, 1994.

[6] Wilfried Oed. Massiv-paralleles Prozessorsystem CRAY T3E. Technische Dokumentation, Cray Research GmbH, Riesstraße 25, 80992 München, 1996.

[7] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. *ACM SIGPLAN Notices*, 31(9):26–36, September 1996.

[8] Steven L. Scott and Gregory M. Thorson. The Cray T3E network: Adaptive routing in a high performance 3D torus. *HOT Interconnects IV*, August 15-16 1996.

[9] The Portland Group, Inc. *pghpf: User's Guide*. 9150 SW Pioneer Court, Suite H, Wilsonville, Oregon 97070, February 1997.

[10] Thomas M. Warschko. *Effiziente Kommunikation in Parallelrechnerarchitekturen*. PhD thesis, Institut für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, Am Fasanengarten 5, 76128 Karlsruhe, 1997. To appear.

[11] Thomas M. Warschko, Christian G. Herter, and Walter F. Tichy. Latency hiding in parallel systems: A quantitative approach. Interner Bericht

10/94, Universität Karlsruhe, Fakultät für Informatik, März 1994.

A E-register programming

The appendix covers some details needed for programming of the E-registers. The information about hardware centrifuge and address-translation is taken from [7]. All other features are documented in the according C-headers.

A.1 The hardware centrifuge

An E-register command operates on five single E-registers. These are partitioned into an aligned Mask-Offset-(MO-)block which contains four E-registers and another E-register (see figure 19).

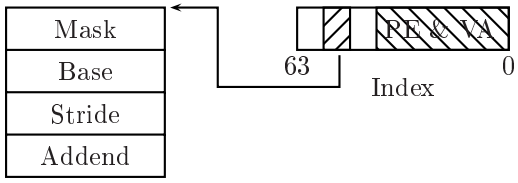

Figure 19: Five E-registers forming a command.

Each E-register is 64 bits long. The *Index* contains at bit 56 the number of the additional MO-block. This number is read and extracted. The remaining bits form a merged PE and virtual-address index. The *Mask* of the aligned-block indicates those bits in *Index* forming the PE number. The masked bits are extracted and form the virtual PE number. Now, the *Base* is added to *Index* resulting in the final virtual-address. According to the virtual PE number the command is issued to the network in case of a non-local memory-access. Typically a *Base* and *Mask* of a MO-block are set up for each shared distributed array and then the index is varied. The *Stride* is necessary for vector-gets and -puts. After each E-register command, it is added to the *Index* to get a new mixed PE number and virtual-address. The *Addend* is used for *fetch-and-add* operations.

A.2 Programming

An E-register command is performed as follows:

1. *Mask* and *Base* are written to a MO-block. The number of the *Mask*-register has to be a multiple of 4.
2. The command itself is separated into the actual command and the *Index* containing the address. The command as its whole is issued as an assignment. The actual command is the left-hand side and the address is the right-hand side, e.g in C:

```
*PUT(E-register-number) = Index
```

As a result, a write to a non-local memory-location is a store to I/O-space, which is expensive but with less processor-overhead than a traditional message-passing system.
E-registers are able to perform the following operations:

Put: To perform a put to a global (local or non-local) memory-location the processor must store the value to a specified E-register first. In the according E-register-command this E-register is named.

Get: It is formed in the same way as a put. The distinction is in reading the desired value. On a load from the specified E-register the processor will be stalled if the E-register is not yet filled by the network. Stalling is done by the above mentioned state-bits as long as the value arrives. Afterwards the processor continuous.

Vector-operations: E-registers can be formed to a vector. Thus, one vector-command serves eight E-registers. This decreases processor overhead and increases network throughput. The stride for gets and puts is given by *Stride* in the aligned-block. For fast access of the vector-values, they should be written first to local memory and then loaded into the processor, because cacheable memory loads can be performed at roughly twice the bandwidth of E-register loads.

Atomic-Memory-Operations: (AMOs) Most interesting of these are the *fetch-and-increment* and *fetch-and-add* AMOs. They add a specified value to the target memory-content and return the original value. *Addend* supplies the value for fetch-and-add.