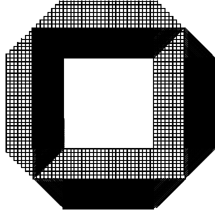


UNIVERSITÄT KARLSRUHE
FAKULTÄT FÜR INFORMATIK
Am Fasanengarten 5, D–76128 Karlsruhe

Latency Hiding in Parallel Systems: A Quantitative Approach

Thomas M. Warschko
Christian G. Herter
Walter F. Tichy

Interner Bericht Nr. 10/94 – März 1994

Latency Hiding in Parallel Systems: A Quantitative Approach

Thomas M. Warschko, Christian G. Herter and Walter F. Tichy
University of Karlsruhe, Dept. of Informatics
Am Fasanengarten 5, 76128 Karlsruhe, Germany
email: {warschko|herter|tichy}@ira.uka.de

April 5, 1994

Abstract

In many parallel applications, network latency causes a dramatic loss in processor utilization. This paper examines software pipelining as a technique for network latency hiding. It quantifies the potential improvements with detailed, instruction-level simulations.

The benchmarks used are the Livermore Loop kernels and BLAS Level 1. These were parallelized and run on the instruction-level RISC simulator DLX, extended with both a blocking and a pipelined network. Our results show that prefetch in a pipelined network improves performance by a factor of 2 to 9, provided the network has sufficient bandwidth to accept at least 10 requests per processor.

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. Latencies lower than $10\mu s$ are rare among commercial parallel architectures[HS94]. Given a 100 MHz clock, a $10\mu s$ latency corresponds to 1000 clock cycles. Thus, instead of a single communication operation one could perform 1000 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, then the loss in efficiency can be reduced by overlapping communication and computation. The basic concept of hiding latency can be used with a great variety of policies [GHG+91, CB92, RL92, CKP91, GGV90].

Little is known about the effects of latency hiding applied to communication networks in massively parallel computers with distributed memory. This paper reports on simulation experiments that quantify the effects of latency hiding on real programs, namely parallel versions of the Livermore Loops, BLAS Level 1 and a few others.

Basic latency hiding techniques are discussed in Section 2, while Section 3 introduces the pipelined network communication analyzed in this paper. Section 4 contains the description of our simulation framework and the benchmark set used. Simulation results are presented in Section 5.

2 Latency Hiding

In general there are two ways for implementing latency hiding: (a) *thread switching* and (b) *code reorganization*, the latter together with a non-blocking prefetch mechanism. These two basic methods apply to both hiding primary memory latency as well as hiding communication latency. Data access over a network is simply another level in the memory hierarchy with an extremely large access time. There is a large body of related work discussing various techniques, such as prefetching cache lines, non-blocking loads, scheduling techniques, and speculative executions on uniprocessors or small-scale multiprocessors [CB92, RL92, MLG92, CKP91, GGV90].

The basic idea of latency hiding by *thread switching* is to de-activate a thread stalled by communication and to switch to another thread. When the communication operation has finished, the system can continue processing the original thread. The cause for a stall can be an explicit communication operation in a distributed memory architecture or a cache miss (i.e., and implicit communication) in a cache-based shared memory architecture. For a detailed discussion of thread switching see [GHG+91].

The concept of *code reorganization* for hiding communication latency is based on splitting each commu-

nication operation into a request, i.e. a non-blocking prefetch, followed by an access operation. The prefetch operation is moved backwards in the code to the extent allowed by data dependencies and consistency requirements. The execution of the code between the prefetch and corresponding access operation overlaps the communication operation. It is also possible to overlap several communication operations by issuing multiple requests in succession.

Latency hiding by code reorganization combines well with virtualization loops. Virtualization loops are used frequently in data parallel programs as an efficient substitute for thread switching. Given n processes and p processors (where $n > p$), we define the virtualization ratio $v = \lceil n/p \rceil$. This ratio is also called parallel *slackness* [Val90a]. In data parallel programs, v processes are efficiently simulated by a compiler-generated virtualization loop per processor [PHL93]. For hiding communication latency, a compiler can use knowledge about virtualization and data layout to generate prefetch code for non-local data references.

In the following, we concentrate on latency hiding by code reorganization in virtualization loops. We expect similar results to hold for fast thread switching.

3 Software Pipelining

The technique of combining virtualization loops with code reorganization enables a compiler to hide a significant portion of communication latency for remote read operations[1]. This section shows how a compiler would use a non-blocking prefetch operation and which hardware support is needed. The difference to traditional pipelining techniques on vector computers is that we assume no vector registers.

3.1 Software Controlled Data Pipelining

In the following example, a permutation of vector B is copied to vector A. All elements `A[i]` are assumed local to processor i. For sake of clarity we assume problem size n to be a multiple of the number of physical processors p.

[1] The latency introduced by remote write operations cannot be overlapped in general. If processor i issues a write operation and processor j issues a read operation to the same data element, some kind of synchronization between these two accesses is needed.

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

On a shared address space architecture one would code the example as a parallel virtualization loop:

```
v := n/p;
FORALL j : [0 .. p-1] DO
  FOR k = j*v TO (j+1)*v - 1 DO
    a1 := calc_address(B[q[k]]);
    A[k] := remote_read(a1);
  END;
END
```

We assume that the majority of data accesses caused by the permutation vector q are nonlocal. Furthermore, to simplify the code, we also assume that the `remote_read` function recognizes local addresses and performs local reads for them. In many parallel architectures, the `remote_read` function is atomic, which causes the processor to stall until the communication network delivers the requested data. This means that the execution of the whole loop is slowed down by approximately $v = n/p$ communication stalls.

With appropriate prefetch instructions issued as early as possible the code becomes:

```
v := n/p;
FORALL j : [0 .. p-1] DO
  FOR k = j*v TO (j+1)*v - 1 DO
    a1 := calc_address(B[q[k]]);
    prefetch(a1);
  END;
  FOR k = j*v TO (j+1)*v - 1 DO
    a1 := calc_address(B[q[k]]);
    A[k] := access(a1);
  END;
END
```

The first `FOR` loop fetches all necessary elements into a prefetch buffer. The buffer is used by the `access` function to read the corresponding values in the second `FOR` loop. Each prefetch operation v_i of the first `FOR` loop corresponds to an access operation v_j of the second `FOR` loop. The result is a data pipeline between the two `FOR` loops. If the prefetch pattern equals the access pattern, we obtain a perfect pipeline (fifo). Given sufficient bandwidth of the network and enough buffer space, the time T that may be overlapped with communication can be calculated from the elapsed time between the first `prefetch` and the first `access` operation:

$$T = (v-1) * (t_{\texttt{calc_address}} + t_{\texttt{init_prefetch}} + t_{\texttt{loop}}) + t_{\texttt{calc_address}} + t_{\texttt{loop}} \quad (1)$$

The maximum overlap T_{max} is achieved when the data access patterns of both loops are the same.

3.2 Network Models

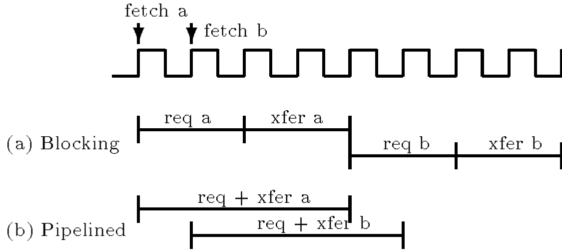


Figure 1: Timings of data access for network models

Before we can calculate the performance improvement that can be gained through prefetching we have to take a closer look at two possible network models (see Figure 1).

- **Blocking:** As soon as a processor issues a communication request, it stalls until the data is delivered. There is no per processor overlap of computation and communication or among multiple communication requests issued by the same processor. Thus, the waiting time of each request is the network latency L.

- **Pipelined(m,k):** A request can be issued every k cycles with the limitation that each node may place at most m simultaneous requests into the network. Limiting the number of requests is necessary in most networks to avoid overload and severe slowdown. In a practical implementation, a prefetch loop would place all requests into a prefetch buffer, from which a communication controller releases them into the network so that no more than m per node are underway. This model provides not only overlap of communication and computation, but also m-way overlap of communication operations. The theoretically possible performance improvement is

$$P_P = \frac{t_{computation} + t_{communication}}{\max(t_{computation}, \frac{t_{communication}}{m})}$$

For communication intensive applications (i.e. $t_{communication} \gg t_{computation}$) where all communication latency can be hidden, P_P evaluates to m. If $t_{communication}/m = t_{computation}$, the formula reaches its maximum $P_{P_{max}} = m + 1$.

Note that the pipelined model includes what is usually called the asynchronous message passing model. By choosing $m = 1$ and $k = L$, we model the situation that each processor can overlap computation with a single communication request. However, multiple requests by the same processor cannot be overlapped. The maximum theoretically possible speedup in this case, according to the above formula, is 2.

3.3 Hardware Considerations

Hardware support for the prefetch mechanism sketched above depends on the memory architecture. In case of a cache-based shared memory, the prefetch buffer can be implemented within the cache by introducing presence bits. A prefetch operation clears the presence bit of the corresponding cache entry. The bit is set when the prefetch completes. The access operation stalls only as long as the presence bit is cleared. This method implements the necessary interlock mechanism.

In a distributed memory architecture, the prefetch buffer can be implemented as a tagged register file or a tagged cache within the network interface of each processor. The interlock mechanism can be the same as above.

The size of the prefetch buffer may be insufficient for the amount of prefetch an application demands. In this case, somewhat more complicated virtualization loops for both prefetch and execution are necessary.

The performance of a *pipelined(m,k)* network is mainly a property of bandwidth and throughput. m simultaneous requests per node requires robustness against network contention. Also, the network should keep the m requests in order (i.e. communication operation j should be completed before communication operation $j + 1$) for the calculated timings of equation 1 to hold. Proper ordering is not necessary for correct execution, but achieves optimal overlap.

4 Simulation Framework and Benchmarks

The architectural model of our simulator is a MIMD machine with a common address space but not necessarily shared memory (XPRAM [Val90b]).

We have extended the instruction-level RISC simulator DLX [HP90] for our experiments. The simulator itself consists of a CPU simulator and a memory system which is capable of handling data access to remote data elements. It takes compiler generated assembler

sources in extended DLX assembly language and reports CPU and memory related statistics.

We augmented the instruction-level CPU simulator for the DLX instruction set with a prefetch instruction. The interlock mechanism for prefetching was implemented as described above by tagging. The simulator uses time stamps to emulate the presence bits. Each time a datum is referenced in memory by a load instruction, the simulator checks if the data reference is valid. This is done by comparing the actual system time (in cycles) with the time stamp associated with each memory word. The system stall time is then calculated by the difference of the time stamp and the actual system time. A prefetch instruction simply adds the network latency to the actual system time and stores the result in the time stamp of the associated memory address.

Both a blocking and a pipelined network model are implemented within our simulator. Blocking means that the CPU is stalled on a remote data access and waits until the network returns the referenced value. In the pipelined case, the possible stalling delay when accessing data is calculated by the difference between its time stamp and the actual system time. Parameters for the *pipelined(m,k)* model are (a) the expected network latency L and (b) the minimum time between two consecutive communication operations k. The missing parameter m (number of simultaneous requests in the network) is set to $m = L/k$. Thus, the request times for the k operations is equally spaced over the latency L.

We use scientific problems, namely a selection of the Livermore Loop kernels, BLAS Level 1, and some others, as the basis for our experiments. The Livermore Loop kernels and BLAS are a popular benchmark in the area of vector supercomputers. We rewrote most of the Livermore Loops and the BLAS routines in a data-parallel fashion[2]. Based on this work, it was easy to extract the necessary node level programs written in C. The `dlxcc`[3] compiler generates assembly code for our simulator. The prefetching code was inserted by hand into the C source code according to the example given in section 3.1. The prefetch instruction itself was inserted using the `asm` facility of `dlxcc`.
To sketch the impact of latency hiding on our benchmark set, Table 1 shows a characterization of each benchmark in terms of used communication patterns.

[2] A description of how to parallelize the Livermore Loops can be found in [Feo88]. Livermore Kernels 17 and 20 were omitted, because they can not be parallelized and Livermore Loop 16 is too difficult to simulate.

[3] `dlxcc` is derived from the `gcc` compiler to produce DLX assembly output.

benchmark		communication pattern
ll1	hydro fragment	array offset
ll2	ICCG excerpt	reduction
ll3	inner product	reduction
ll4	banded linear equation	reduction
ll5	tri-diagonal elimination	reduction
ll6	general linear recurrence equations	broadcast
ll7	equation of state fragment	array offset
ll8	ADI integration	array offset
ll9	integrate predictors	none
ll10	difference predictors	none
ll11	first sum	reduction
ll12	first difference	array offset
ll13	2-D particle in cell	indirect addressing
ll14	1-D particle in cell	indirect addressing, reduction
ll15	casual Fortran	array offset
ll18	2-D explicit hydro-dynamics fragment	array offset, reduction
ll19	general linear recurrence equations	reduction
ll21	matrix*matrix product	broadcast, reduction
ll22	Planckian distribution	none
ll23	2-D implicit hydro-dynamics fragment	array offset, reduction
ll24	first minimum	reduction
	jacobi	2-D grid
	red-black SOR	2-D grid
bl1.1	srotg	none
bl1.2	srot	array offset
bl1.3	sswap	array offset
bl1.4	sscal	array offset
bl1.5	scopy	array offset
bl1.6	saxpy	array offset
bl1.7	sdot	array offset, reduction
bl1.8	snrm2	reduction
bl1.9	sasum	reduction
bl1.10	isamax	reduction

Table 1: Classification of benchmark suite

- **none**: No communication needed.
- **array offset**: Communication between different array elements (i.e. A[i] and A[i+offset]). Here it is possible to prefetch all remote elements at once.
- **2-D grid**: Typical north-west-south-east communication pattern. With an optimal data layout, only the border of the local $j \times j$ rectangular block consists of remote data elements, which can be prefetched.

- **reduction**: Typical reduction tree, used to compute a vector-sum, min(max) of a vector, pre/postfix sums, etc. Prefetching in this kind of application is difficult, because the results calculated in level j of the reduction tree are needed in level $j+1$ as input operands. The opportunity for latency hiding was improved by increasing the fan-in in each stage.

- **indirect addressing**: Most of the data is accessed in a $A[q(i)]$ fashion, where $q(i)$ is an a priori unknown permutation. Thus, it is impossible to exploit any knowledge about data layout for statically optimizing communication patterns. Nevertheless, aggressive prefetching of all data elements was used.

5 Results

We chose the following parameters for our experiments. To get close to the reality of massively parallel machines we set the number p of physical processors to 1024. However, most of our benchmarks are not influenced by the number of physical processors, because the communication patterns – array offset, 2-D grid and indirect addressing (see Table 1) – do not depend on machine size. Only those benchmarks with reduction operations are influenced by machine size, because the height of the reduction tree and therefore the number of communication operations evaluates to $h = \log_f(p)$, where f is the fan-in. Communication latency L was set to 1000 CPU-cycles, according to the estimate in Section 1. For the *pipelined(m,k)* network model we choose $m = 10$ and $k = 100$, so that $m * k = 10 * 100 = 1000 = L$. Additionally, results are presented for latency $L = 100$ ($m = 10, k = 10$) and $L = 10,000$ ($m = 10, k = 1000$). For the BLAS Level 1 routines, we use different offsets for the X and Y arrays (i.e. $incx = 2$ and $incy = 4$). Otherwise there would be no communication at all. We define *utilization* of the parallel computer for a given program as the ratio of measured runtime versus runtime on an ideal PRAM without any communication latency.

The Jacobi iteration (benchmark 22) calculates for each point in a 2-dimensional grid the arithmetic mean value of its four neighboring elements. An optimal data layout is achieved by dividing the original $N \times N$ grid into rectangular blocks of size $j \times j$ ($N^2 = j^2 * p$). This decomposition reduces the amount of communication operations per processing element to $4 * j$, which matches the number of border elements of the local $j \times j$ rectangular block.

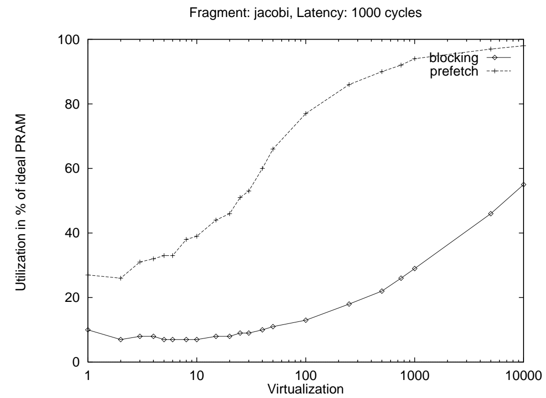


Figure 2: Utilization of Jacobi iteration

According to this data layout, Figure 2 shows the utilization of the Jacobi iteration while varying parallel slackness (virtualization). Over a wide range, the utilization under blocking communication does not exceed 20%. A utilization of 50% is not reached until a virtualization of 5000. Prefetching reaches 50% utilization with a virtualization of only 25. With a parallel slackness of only 750, utilization with prefetch is above 90%.

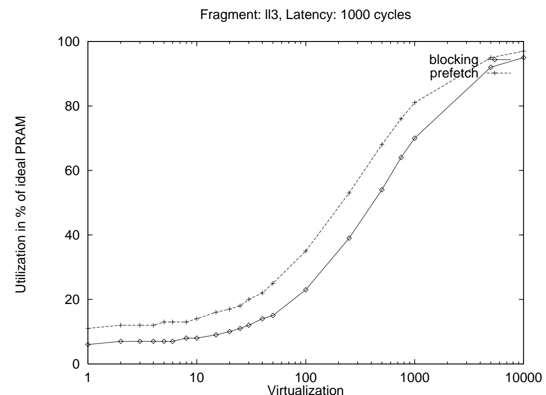


Figure 3: Utilization of Livermore Loop 3

Results for the inner product (ll3) are shown in Figure 3. As before, the latency hiding mechanism shows advantages over normal (blocking) communication. A slackness of 250 in contrast to 500 in the blocking case is needed to get to 50% utilization. To reach 90% utilization one has to increase virtualization to 2000 and 5000, respectively.

Figure 4 shows the utilization for the tri-diagonal elimination (ll5). Neither 50% nor 90% utilization is reached because a reduction operation for each of the local elements is needed. Thus, a parallel slackness of k needs $k \times \log p$ communication operations. Neverthe-

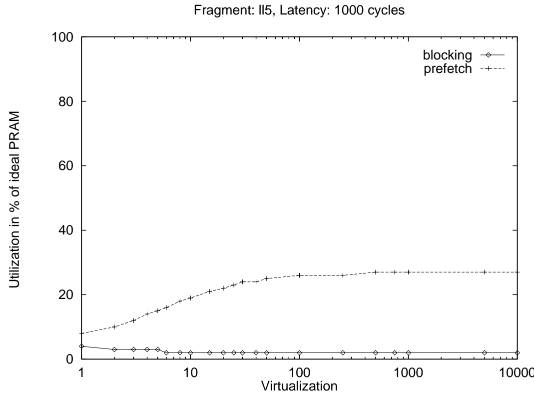


Figure 4: Utilization of Livermore Loop 5

less, the latency hiding mechanism with 27% utilization shows an enormous advantage over normal (blocking) communication with only 4% utilization.

Another possibility to visualize the advantages of latency hiding techniques is to calculate the performance improvement. This is done by dividing the execution times of pipelined and blocking execution ($pi = t_{pipelined}/t_{blocking}$). Note that with the number of simultaneous outstanding requests set to $m = 10$ the maximum possible performance improvement – as calculated in section 3.2 – is 11. The results for the Jacobi iteration, Livermore Loop 3 and Livermore Loop 5 are graphed in Figures 5 to 7.

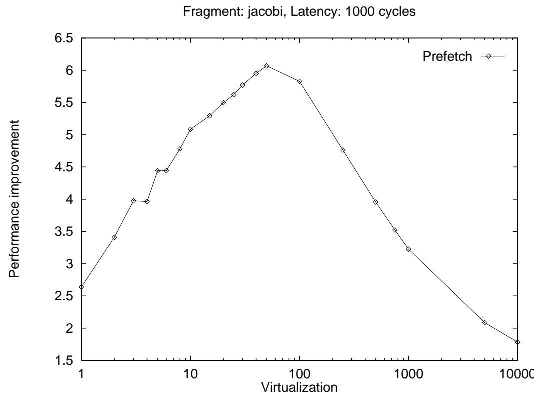


Figure 5: Performance improvement of Jacobi

The Jacobi iteration has a performance improvement larger than a factor of 3 over a wide range of virtualization ratios. Peak values of 5 to 6 are reached between virtualization ratios of 10 to 200. As shown in Figures 2 and 3, the utilization of blocking communication tends to 100% with increasing virtualization. This effect explains why performance improvement decreases, especially when looking at the per-

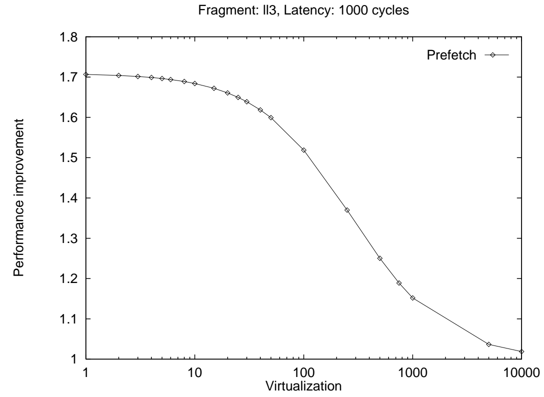


Figure 6: Performance improvement of ll3

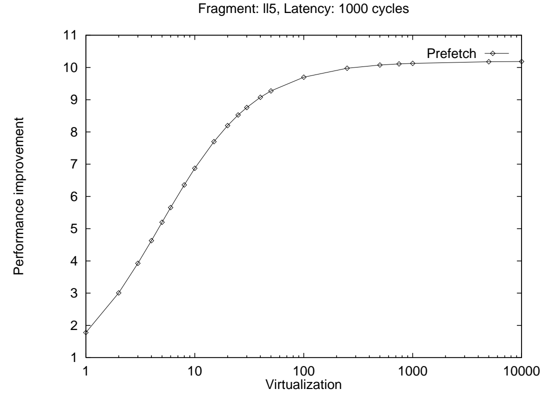


Figure 7: Performance improvement of ll5

formance improvement of Livermore Loop 3. The performance improvement of the tri-diagonal elimination (ll5) shows quite a different behavior. With increasing virtualization the performance improvement tends to the theoretical maximum of 11. In this case, the network capability of overlapping $m = 10$ simultaneous requests is fully used.

The three completely different graphs can be characterized by the communication complexity of the appropriate benchmark:

- $O(communication) = const.$: The communication complexity is fixed regardless of the virtualization degree. With increasing virtualization, the $computation/communication$ ratio also increases. This leads to a better utilization, but also to a decreasing performance improvement. Typical examples of this behavior are single broadcast / reduction operations (i.e., ll3).
- $O(communication) < O(computation)$: Communication complexity is rising slower than the computation complexity. With increasing virtualiza-

tion, the *computation/communication* ratio still increases, but less than in the previous case. Prefetching in the pipelined network model leads to a better utilization at a smaller virtualization degree compared to a blocking network. This results in an increasing performance improvement up to a certain threshold. Afterwards – with better utilization – performance improvement decreases. Typical examples of this behavior are array offset and grid communication patterns (i.e., jacobi).

- $O(communication) \geq O(computation)$: Communication complexity is higher than computation complexity. Thus, with increasing virtualization the *computation/communication* ratio is fixed or decreases! Is is hard to reach an acceptable level of utilization, but the performance improvement of a pipelined over a blocking network is significant, because the network capability of overlapping m simultaneous requests can be fully used. Typical examples of this behavior are indirect addressing, cyclic data-layout (BLAS) and multiple broadcast / reduction communication patterns (i.e., ll5).

One question of interest is: What is the necessary degree of virtualization to reach 50% or 90% utilization? Table 2 gives the answer for all of our benchmarks, including the examples discussed above.

Through latency hiding, nearly all benchmarks reach 50% as well as 90% utilization with much lower virtualization degree. In general, the problem size can be at least 2 to 10 times smaller to reach the same utilization. That makes it possible to solve even small problems with acceptable utilization. Codefragments ll9, ll10, ll22, srotg and sscal do not need any communication at all, so we see maximum utilization regardless of problem size. Livermore Loops 13 and 14 use only indirect addressing to access non-local data. This makes it impossible to use any knowledge about data layout to optimize communication patterns, so an aggressive prefetching method has to be used (i.e., prefetch all elements which might be remote). With blocking communication, a high virtualization degree is needed to reach 50% utilization (`ll13`). Benchmark `ll14` is even worse: Only 9% utilization with virtualization 10000 is reached. With latency hiding, a much smaller degree of virtualization is sufficient to get 50%. As consequence of the aggressive prefetching, the maximum utilization of `ll13` is about 68% (`ll14`: 55%). The missing 32% (45%) are due to overhead of the latency hiding mechanism. The difference in utilization between blocking and pipelined communication leads

benchmark	50% utilization blocking/prefetch		90% utilization blocking/prefetch	
ll 1	250	20	5000	40
ll 2	600	150	7500	2000
ll 3	500	250	5000	2000
ll 4	500	250	5000	2000
ll 5	(4%)†	(27%)†	(4%)†	(27%)†
ll 6	1500	600	6000	3000
ll 7	50	6	500	12
ll 8	20	3	200	30
ll 9	1	1	1	1
ll 10	1	1	1	1
ll 11	250	150	2000	1500
ll 12	60	30	750	75
ll 13	4000	400	>10000	(68%)†
ll 14	(9%)†	1000	(9%)†	(55%)†
ll 15	15	5	150	30
ll 18	100	50	1000	500
ll 19	200	75	2500	750
ll 21	(9%)†	(17%)†	(9%)†	(17%)†
ll 22	1	1	1	1
ll 23	(4%)†	(29%)†	(4%)†	(29%)†
ll 24	2500	1000	>10000	10000
jacobi	5000	25	>10000	500
red-black-sor	3000	20	>10000	750
bl1.1	1	1	1	1
bl1.2	(8%)†	10	(8%)†	(54%)†
bl1.3	(12%)†	6	(12%)†	(57%)†
bl1.4	1	1	1	1
bl1.5	(6%)†	50	(6%)†	(53%)†
bl1.6	(6%)†	25	(6%)†	(56%)†
bl1.7	(5%)†	750	(5%)†	(53%)†
bl1.8	150	80	2500	900
bl1.8	150	90	2500	1000
bl1.10	150	90	2500	1000
† values indicate maximum utilization achieved				

Table 2: Necessary virtualization (1000 cycles latency)

to a performance improvement of factor 9.4 and 7.2, respectively, in these two cases. Another behavior is shown in codefragments ll5, ll23, srot, sswap, scopy and saxpy. With increasing virtualization the number of communication operations also increases and no acceptable utilization is reached. The utilization varies between 4% and 12% for blocking communication and between 27% and 57% for pipelined communication. Still, the codefragments show an excellent improvement factor of 9 and above with latency hiding.

Comparing the simulation results (Table 2) and the classification of our benchmark set (Table 1), it becomes evident that benchmarks with similar communications patterns show also roughly the same behavior

in our simulator. Interesting is the combination of two or more communication patterns in one benchmark. An example is `ll18`, where an array offset is combined with reduction operations. According to the results in Table 2 the reduction operation seems to dominate the array offset and in fact this holds. With an increasing virtualization degree all latency caused by the array offset can be hidden, so the remaining latency is due to reduction operations.

Another question is how these results are influenced by communication latency. Table 4 (see appendix) shows the results of our benchmark set with latency set to 100 CPU-cycles ($m = 10, k = 10$). Table 5 does the same with latency 10,000 ($m = 10, k = 1000$). The basic conclusions drawn from the results above still hold. Of course, the necessary virtualization ratio must increase with communication latency to achieve a given utilization. With a close look at Table 5 (latency 10,000) it becomes evident that it is nearly impossible to reach an acceptable utilization with blocking communication on high latency networks. A communication delay of 10000 CPU-cycles is not as unlikely as one might think: A CPU operating at 200 MHz coupled with a $50\mu s$ network latency can execute 10000 instructions during one communication.

Table 3 summarizes our results by showing the maximum performance improvement achieved. All benchmarks, besides those with reduction operations, show performance improvements ranging from 2 to 9. Increasing network latency makes the advantage of prefetching even more pronounced. It is also interesting to check how close to the theoretically possible performance improvement – as calculated in section 3.2 – one can get. According to the parameters chosen, the maximum possible performance improvement is 11, because the number of simultaneous requests in the network was set to $m = 10$ in all simulations.

6 Conclusion

This article quantifies the performance losses caused by the communication network in massively parallel, distributed memory computers. High-latency networks without overlap of communication requests may never reach acceptable utilization or require high virtualization ratios. This fact may be a major cause for the frequently observed, poor utilization of parallel supercomputers.

Software controlled prefetching of data can improve utilization by a factor of 2 to 9, provided the communication network can accept a load of 10 simultaneous requests per processor. A simple analysis of the communication pattern, the computation as well as the communication complexity of a codefragment is often enought to predict the performance improvement of a pipelined over a blocking network.

We are currently extending our benchmark suite with the BLAS Level 2 routines and other examples. We are also studying techniques for incorporating latency hiding into optimizing compilers. Analyzing the tradeoffs possible with ultra-fast thread switching is another research topic.

The general goal of this work is to make the performance of parallel systems predictable for the applications programmer. This involves improvements in communication networks, compiler technology, and performance prediction models.

References

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *The 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[Feo88] John T. Feo. *An analysis of the computational and parallel complexity of the Livermore Loops*, volume 2, pages 163–185. Elsevier Science Publishers B. V. (North-Holland), 1988.

[GGV90] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, pages 354–368, September 1990.

[GHG+91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pages 254–265, Toronto CDN, June 1991. ACM Press, New York, NY, USA.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, INC., San Mateo, California, 1990.

[HS94] Hartmut Häfner and Willi Schönauer. Kommunikation auf verschiedenen Parallelrechnern,

benchmark		Latency 100		Latency 1000		Latency 10000	
		perf. imp.	virt.	perf. imp.	virt.	perf. imp.	virt.
ll1	hydro fragment	2.05	8	5.51	30	5.57	250
ll2	ICCG excerpt	2.24	1	2.56	25	2.60	250
ll3	inner product	1.59	1	1.70	1	1.72	1
ll4	banded linear equation	1.63	1	1.76	1	1.77	1
ll5	tri-diagonal elimination	3.35	500	10.13	1000	9.97	1000
ll6	general linear recurrence equations	1.34	1	2.26	1	2.47	1
ll7	equation of state fragment	2.19	1	3.96	10	4.49	100
ll8	ADI integration	1.92	1	3.52	2	3.97	4
ll9	integrate predictors	1.00	1	1.00	1	1.00	1
ll10	difference predictors	1.00	1	1.00	1	1.00	1
ll11	first sum	1.22	1	1.65	1	1.71	1
ll12	first difference	1.34	6	1.62	50	1.87	750
ll13	2-D particle in cell	4.09	50	9.39	50	9.46	100
ll14	1-D particle in cell	2.20	100	7.18	500	9.27	5000
ll15	casual Fortran	1.48	1	1.97	1	2.11	1
ll18	2-D explicit hydrodynamics fragment	1.38	1	1.95	1	2.03	1
ll19	general linear recurrence equations	1.47	2	1.87	15	1.90	250
ll21	matrix*matrix product	1.45	20	1.90	50	2.02	500
ll22	Planckian distribution	1.00	1	1.00	1	1.00	1
ll23	2-D implicit hydrodynamics fragment	3.06	500	9.90	750	9.95	1000
ll24	first minimum	1.51	1	1.69	1	1.71	1
	jacobi	1.73	6	6.06	50	8.49	750
	red-black SOR	1.63	6	4.76	15	7.36	100
bl1.1	srotg	1.00	1	1.00	1	1.00	1
bl1.2	srot	1.08	500	6.40	250	10.06	1000
bl1.3	sswap	0.93	1	4.55	1000	10.03	1000
bl1.4	sscal	1.00	1	1.00	1	1.00	1
bl1.5	scopy	1.47	1000	9.93	1000	10.03	5000
bl1.6	saxpy	1.44	1000	9.38	1000	10.04	5000
bl1.7	sdot	1.45	1000	9.62	5000	9.94	5000
bl1.8	snrm2	1.48	1	1.68	1	1.71	1
bl1.9	sasum	1.49	1	1.68	1	1.72	1
bl1.10	isamax	1.49	1	1.68	1	1.72	1

Table 3: Maximum performance improvement achieved

Auswirkung auf die Programmierung. In *Proc. of the Third ODIN Symposium*, pages 195–206. Universität Karlsruhe, Rechenzentrum, March 1994.

[MLG92] Todd C. Mowry, Monica S. Lamm, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *The 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[PHL93] Michael Philippsen, Ernst A. Heinz, and Paul Lukowicz. Compiling machine-independent parallel programs. *ACM SIGPLAN Notices*, 28(8):99–108, August 1993.

[RL92] Anne Rogers and Kai Li. Software support for speculative loads. In *The 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1992.

[Val90a] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, pages 103–111, August 1990.

[Val90b] L. G. Valiant. General purpose parallel architectures. In *Handbook of theoretical Computer Science, Vol A: Algorithms and Complexity*, Amsterdam, 1990. J. van Leeuwen, Ed., North Holland.

A Results for Latency 100 and Latency 10000

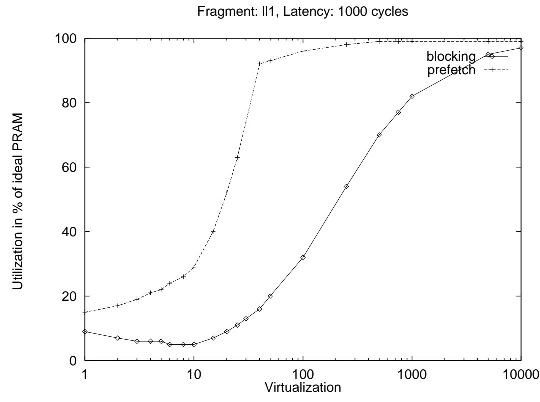
benchmark	50% utilization blocking/prefetch		90% utilization blocking/prefetch	
ll 1	20	1	250	30
ll 2	30	1	500	75
ll 3	15	1	500	200
ll 4	20	1	500	200
ll 5	(21%)	(72%)	(21%)	(72%)
ll 6	200	150	1000	750
ll 7	3	1	40	4
ll 8	2	1	15	2
ll 9	1	1	1	1
ll 10	1	1	1	1
ll 11	15	10	250	200
ll 12	1	1	50	6
ll 13	400	1	4000	(68%)
ll 14	5000	3	> 10000	(83%)
ll 15	1	1	15	2
ll 18	8	3	100	50
ll 19	7	1	200	50
ll 21	10000	1	> 10000	(67%)
ll 22	1	1	1	1
ll 23	(23%)	(71%)	(23%)	(71%)
ll 24	100	1	2500	750
jacobi	25	1	7000	250
red-black-sor	1	1	2500	50
bl1.1	1	1	1	1
bl1.2	(45%)†	1	(45%)†	(50%)†
bl1.3	1	1	(56%)†	(52%)†
bl1.4	1	1	1	1
bl1.5	(35%)†	1	(35%)†	(52%)†
bl1.6	(38%)†	1	(38%)†	(55%)†
bl1.7	(37%)†	1	(37%)†	(54%)†
bl1.8	2	1	200	45
bl1.9	3	1	200	45
bl1.10	2	1	200	45
† values indicate maximum utilization achieved				

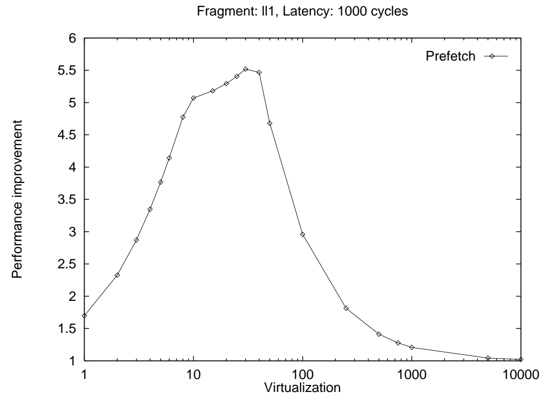
Table 4: Necessary virtualization (100 cycles latency)

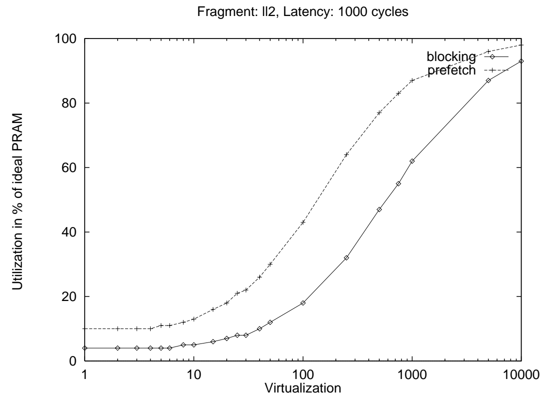
benchmark	50% utilization blocking/prefetch		90% utilization blocking/prefetch	
ll 1	2500	200	> 10000	400
ll 2	9000	3000	> 10000	> 10000
ll 3	5000	4000	> 10000	> 10000
ll 4	7500	4000	> 10000	> 10000
ll 5	(0%)	(2%)	(0%)	(2%)
ll 6	7000	4000	> 10000	> 10000
ll 7	500	75	5000	150
ll 8	200	40	2500	400
ll 9	1	1	1	1
ll 10	1	1	1	1
ll 11	3000	2000	> 10000	> 10000
ll 12	700	300	7000	600
ll 13	(23%)	4000	(23%)	(68%)
ll 14	(1%)	(9%)	(1%)	(9%)
ll 15	150	40	2500	400
ll 18	1000	500	10000	5000
ll 19	3000	800	> 10000	10000
ll 21	(1%)	(2%)	(1%)	(2%)
ll 22	1	1	1	1
ll 23	(0%)	(3%)	(0%)	(3%)
ll 24	> 10000	10000	> 10000	> 10000
jacobi	> 10000	2500	> 10000	> 10000
red-black-sor	> 10000	2000	> 10000	> 10000
bl1.1	1	1	1	1
bl1.2	(0%)†	(8%)†	(0%)†	(8%)†
bl1.3	(1%)†	(12%)†	(1%)†	(12%)†
bl1.4	1	1	1	1
bl1.5	(0%)†	(5%)†	(0%)†	(5%)†
bl1.6	(0%)†	(6%)†	(0%)†	(6%)†
bl1.7	(0%)†	(5%)†	(0%)†	(5%)†
bl1.8	2500	1000	> 10000	10000
bl1.9	2500	1500	> 10000	10000
bl1.10	2500	1000	> 10000	10000
† values indicate maximum utilization achieved				

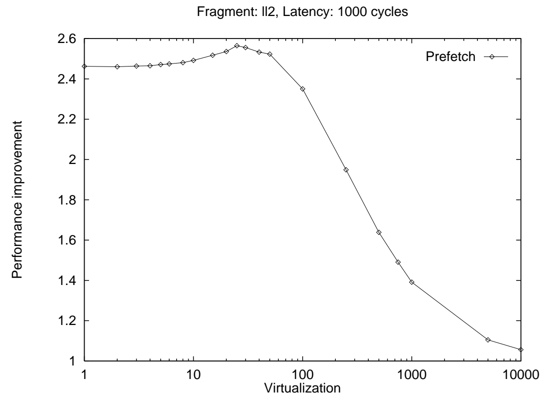
Table 5: Necessary virtualization (10000 cycles latency)

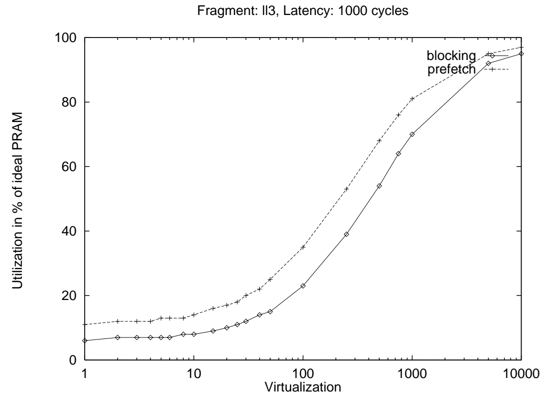
B Figures

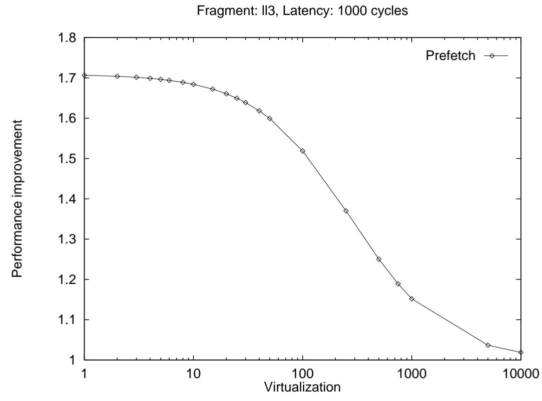


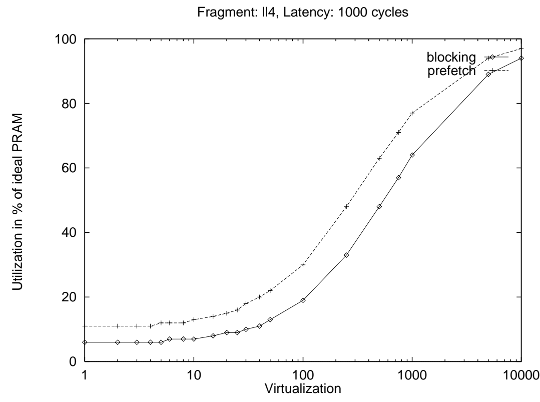


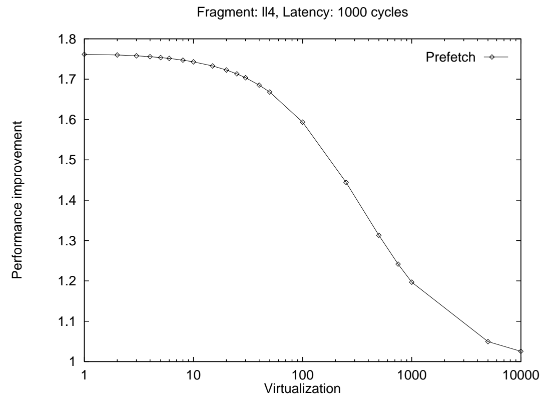


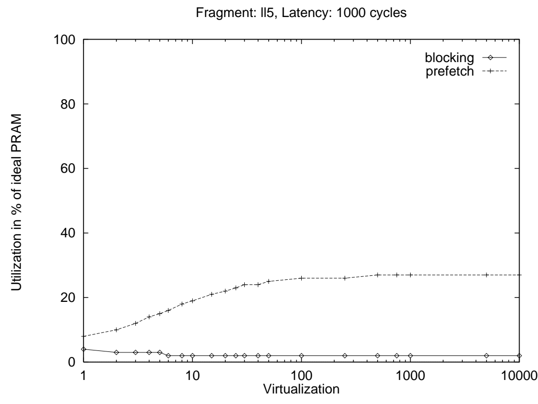




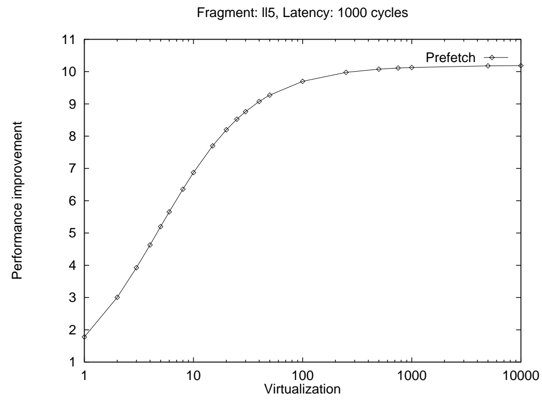




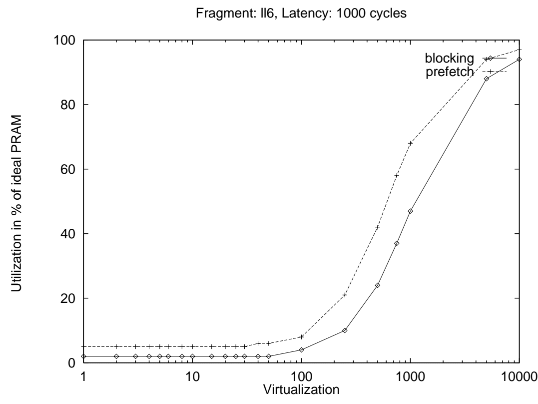




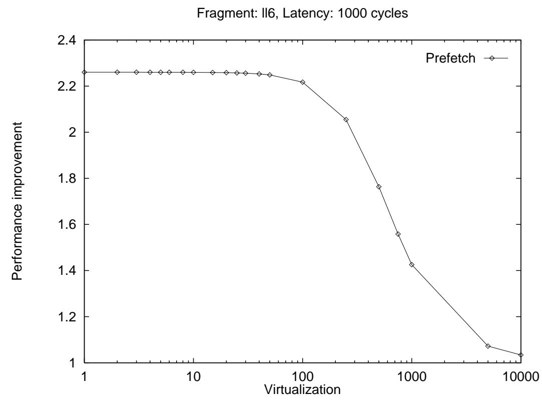
Fragment: ll5, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization



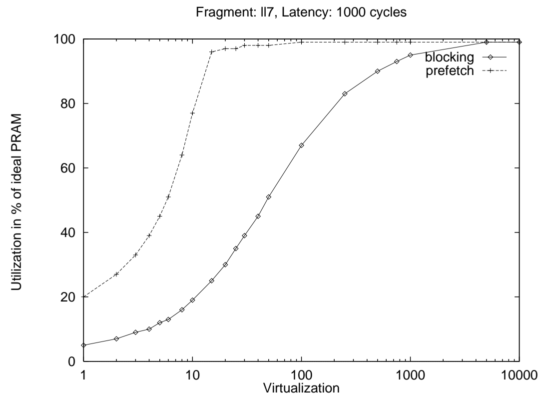
Fragment: ll5, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization



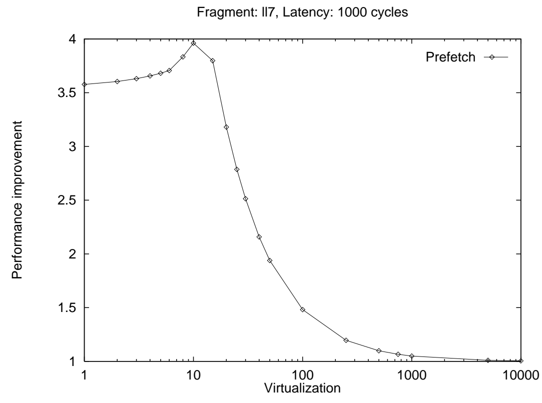
Fragment: ll6, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization



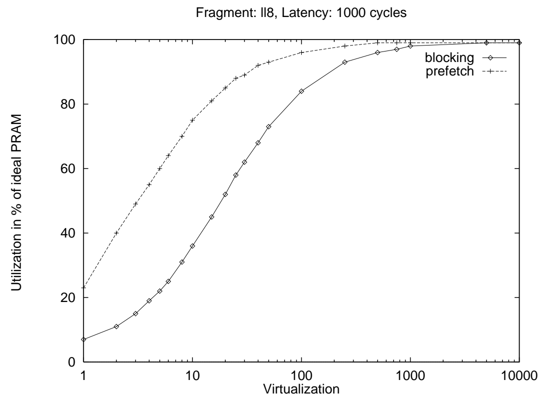
Fragment: ll6, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization



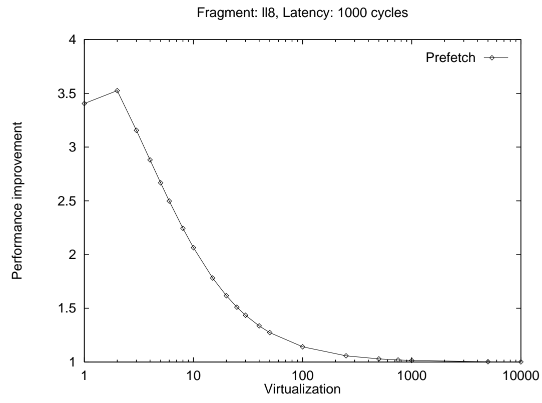
Fragment: ll7, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization



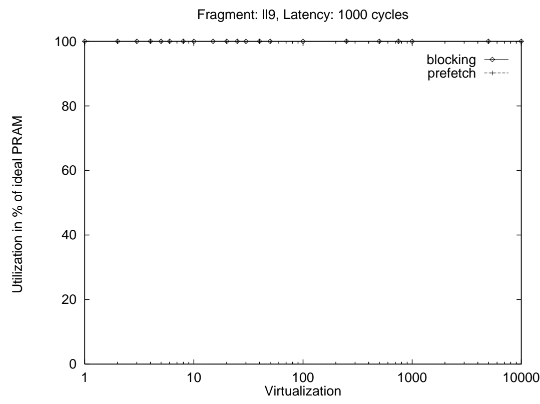
Fragment: ll7, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization

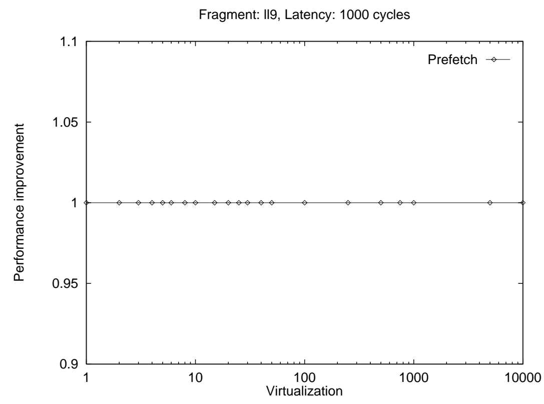


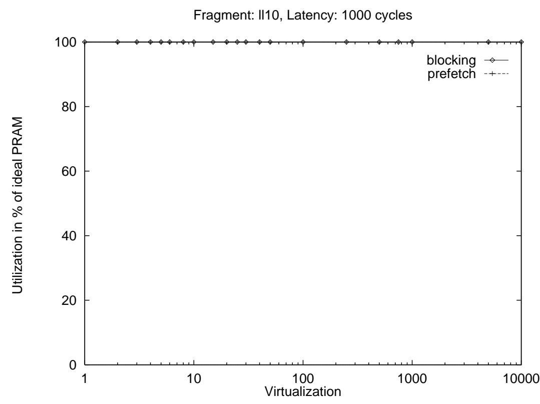
Fragment: ll8, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization

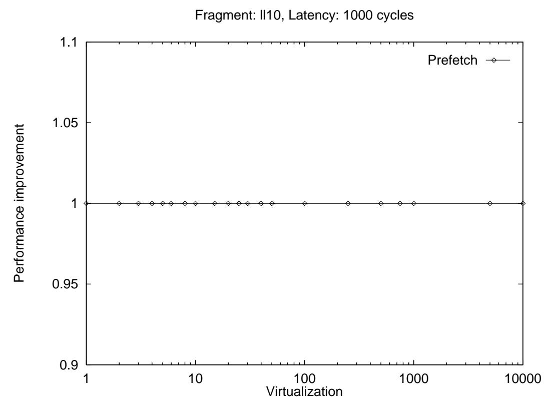


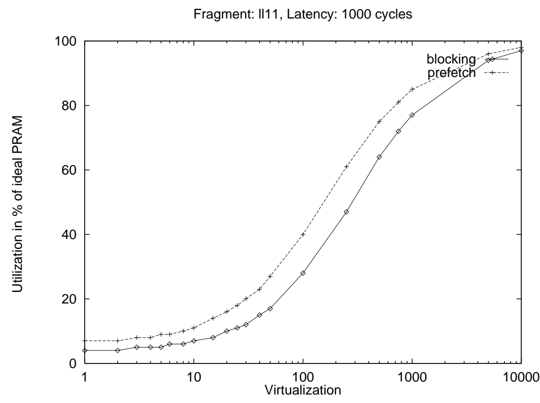
Fragment: ll8, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization

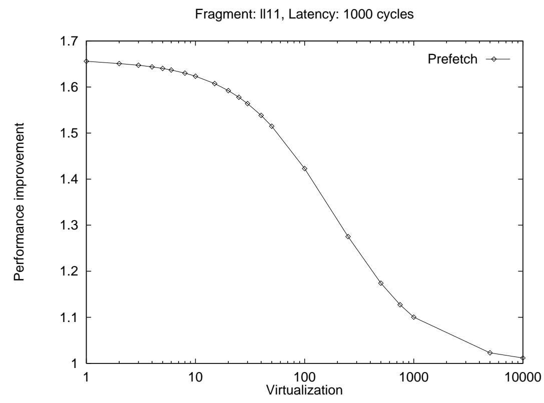

Fragment: ll9, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

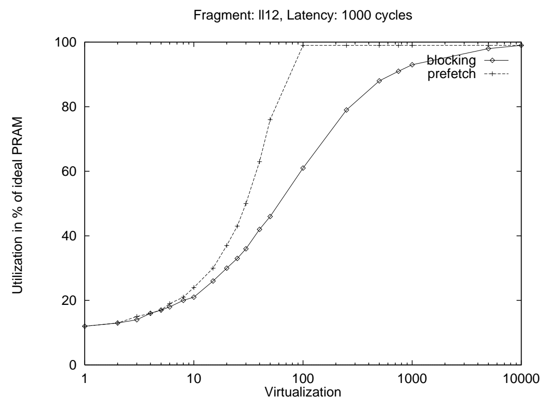

Fragment: ll9, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

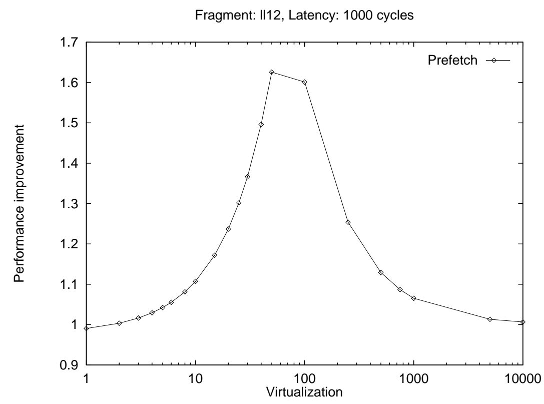

Fragment: ll10, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

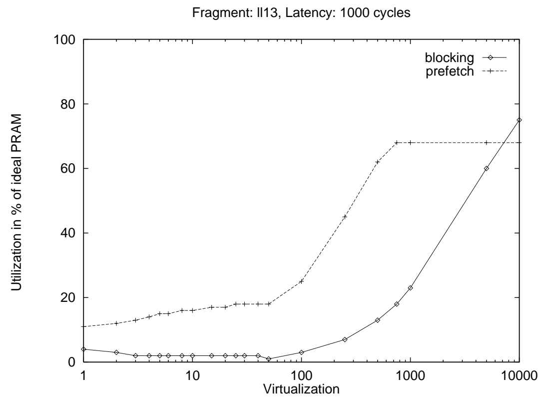

Fragment: ll10, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

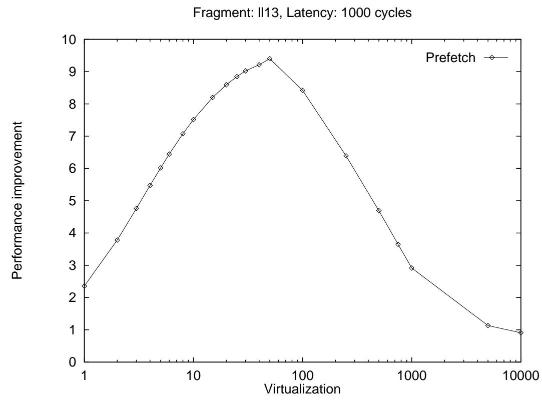

Fragment: ll11, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

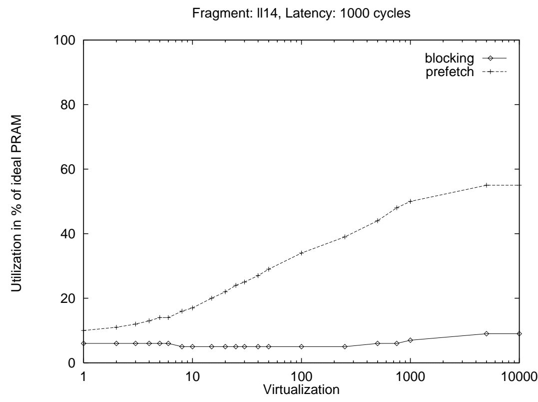

Fragment: ll11, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

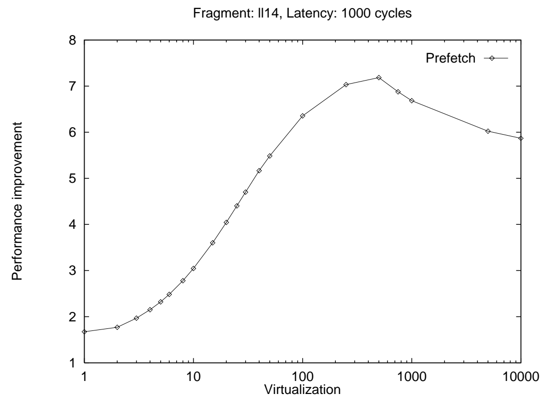

Fragment: ll12, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

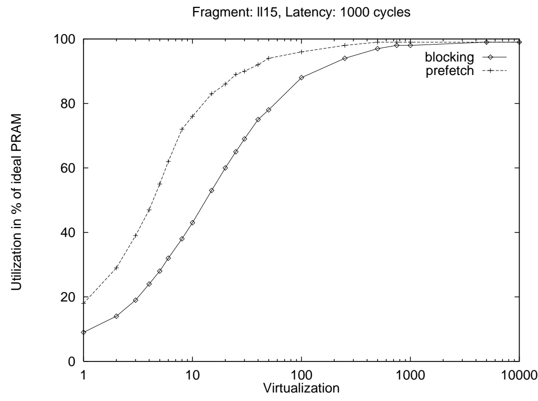

Fragment: ll12, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

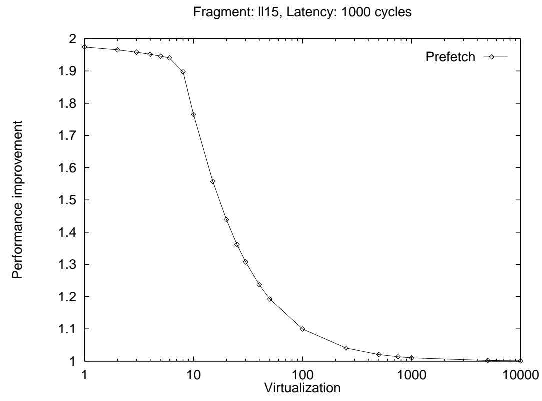

Fragment: ll13, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization
0
20
40
60
80
100
1
10
100
1000
10000

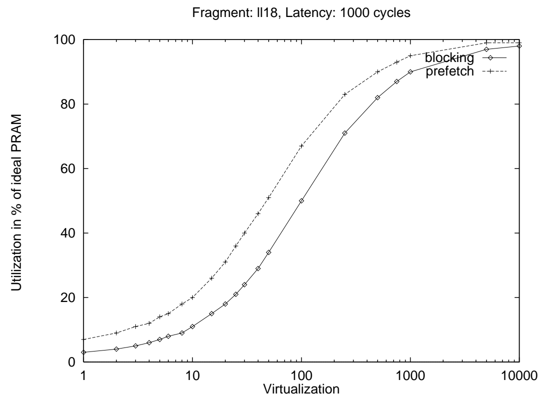

Fragment: ll13, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization
0
1
2
3
4
5
6
7
8
9
10
1
10
100
1000
10000

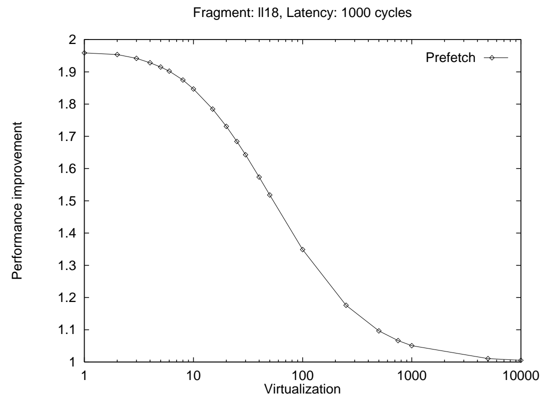

Fragment: ll14, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization
0
20
40
60
80
100
1
10
100
1000
10000

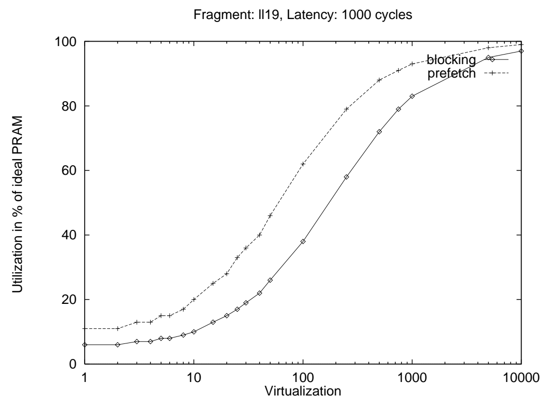

Fragment: ll14, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization
1
2
3
4
5
6
7
8
1
10
100
1000
10000

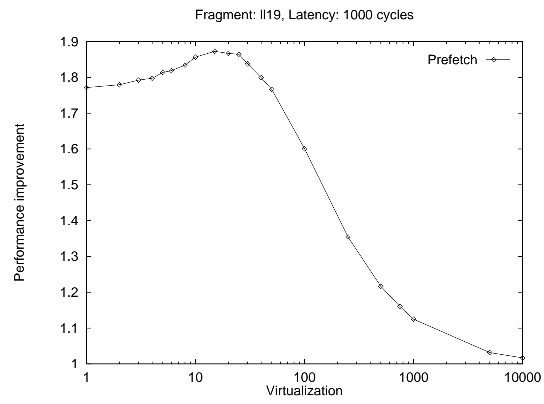

Fragment: ll15, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization
0
20
40
60
80
100
1
10
100
1000
10000

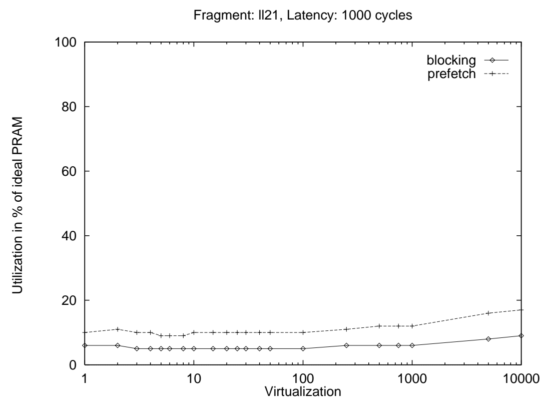

Fragment: ll15, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2
1
10
100
1000
10000

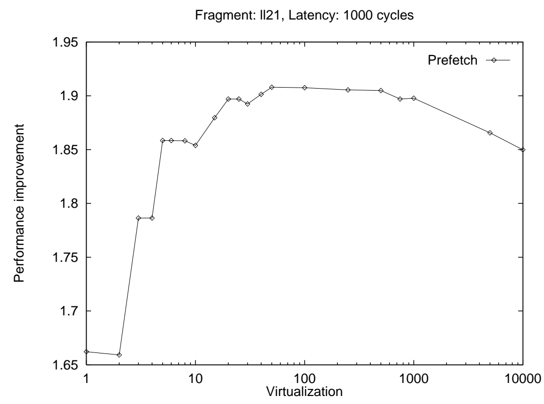

Fragment: ll18, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization
0
20
40
60
80
100
1
10
100
1000
10000

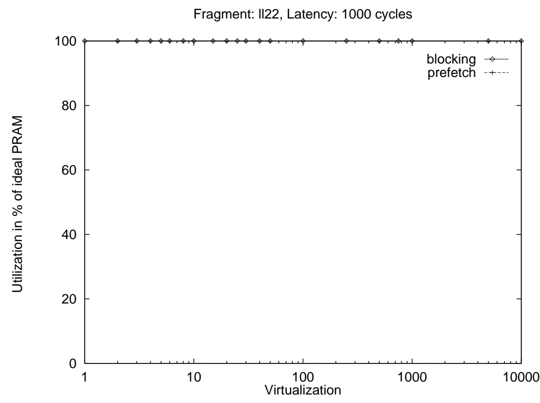

Fragment: ll18, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2
1
10
100
1000
10000

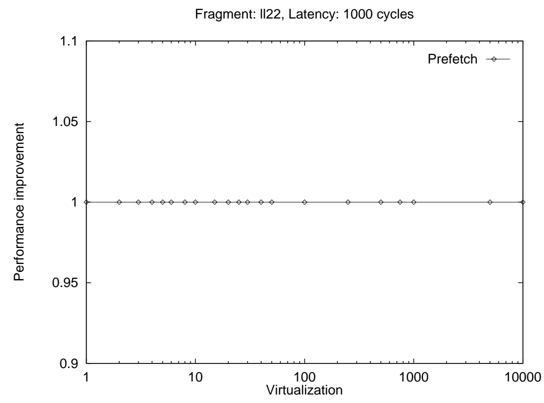

Fragment: ll19, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

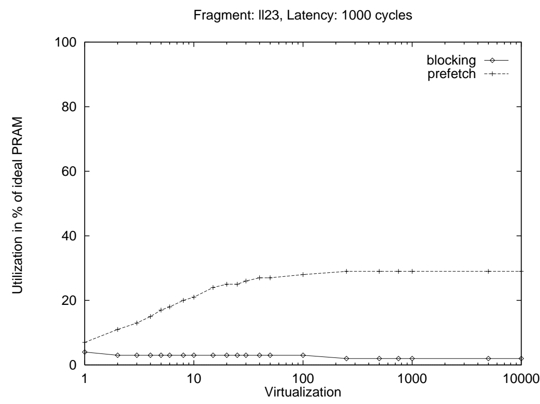

Fragment: ll19, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

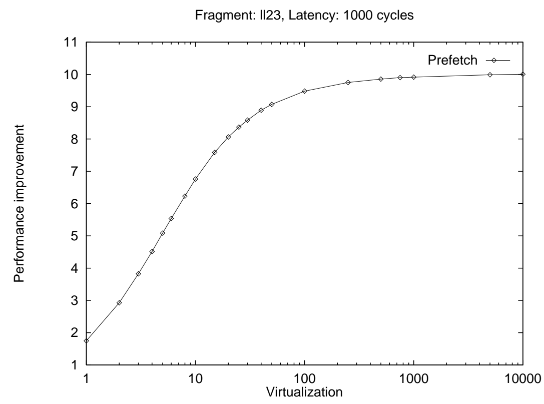

Fragment: ll21, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

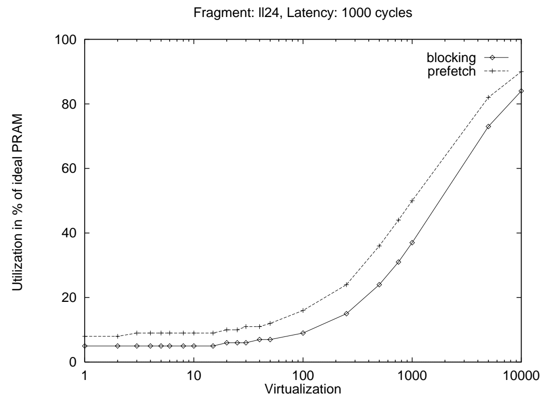

Fragment: ll21, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

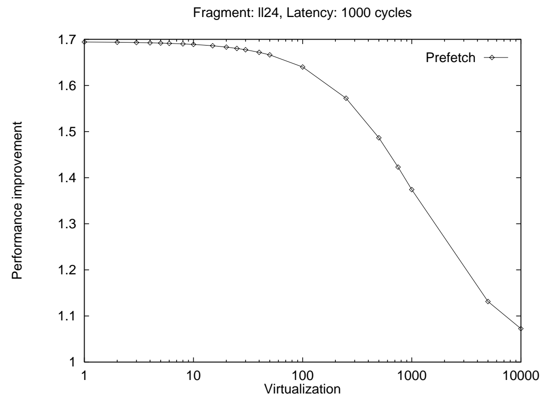

Fragment: ll22, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

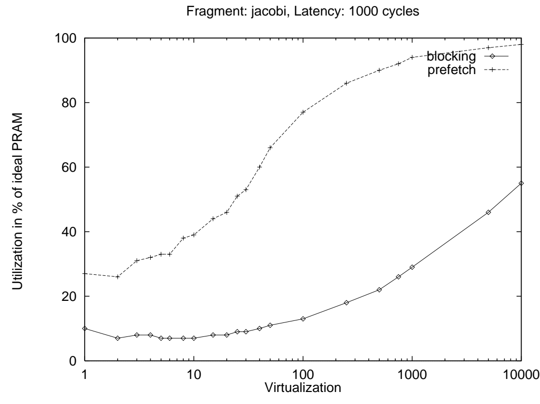

Fragment: ll22, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

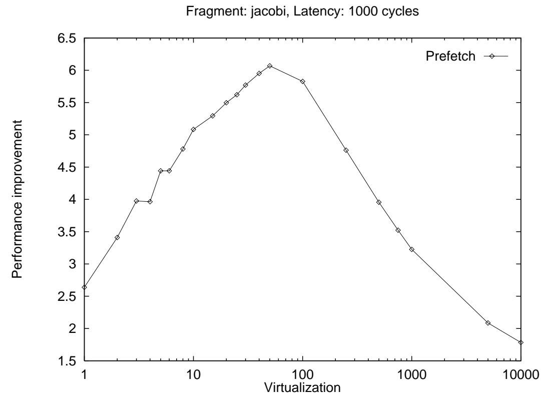

Fragment: ll23, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

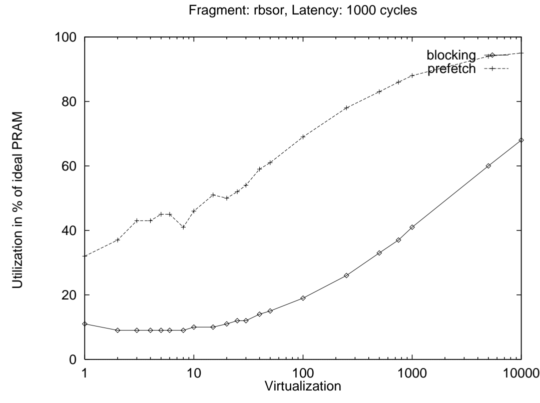

Fragment: ll23, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

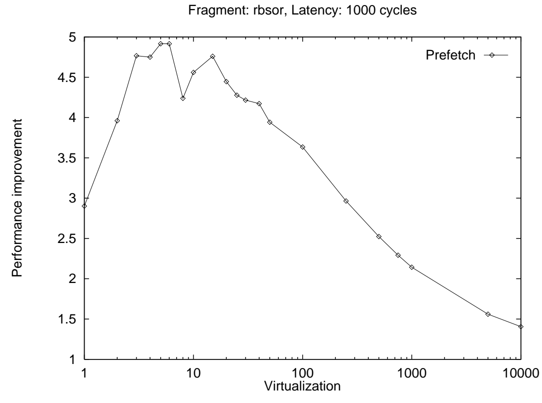

Fragment: ll24, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch
0
20
40
60
80
100
1
10
100
1000
10000

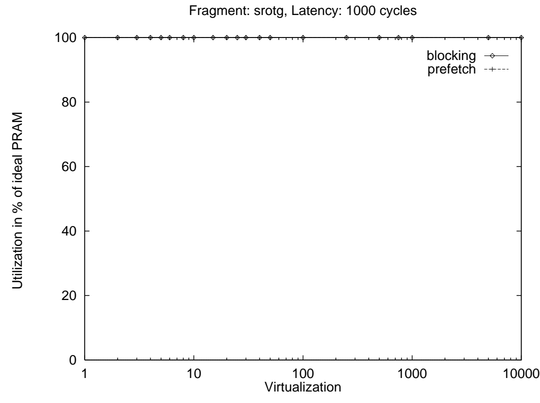

Fragment: ll24, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1
10
100
1000
10000

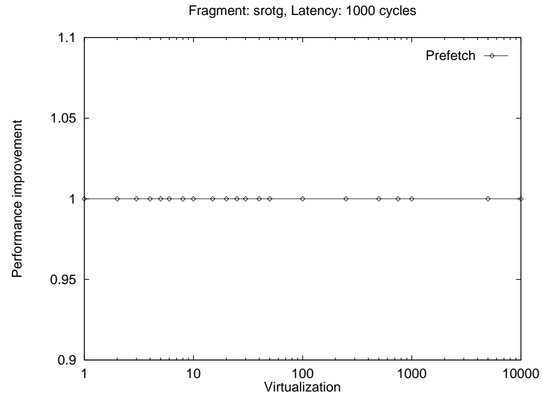

Fragment: jacobi, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch
0
20
40
60
80
100
1
10
100
1000
10000

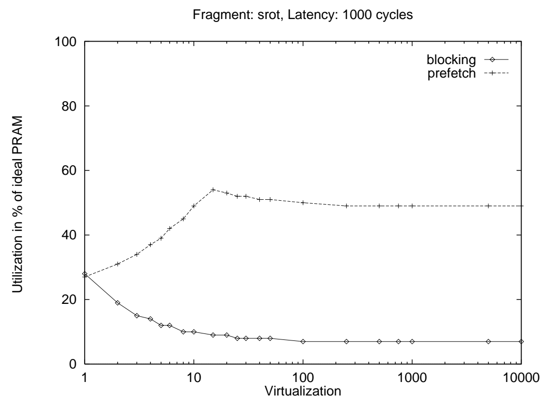

Fragment: jacobi, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
6.5
1
10
100
1000
10000

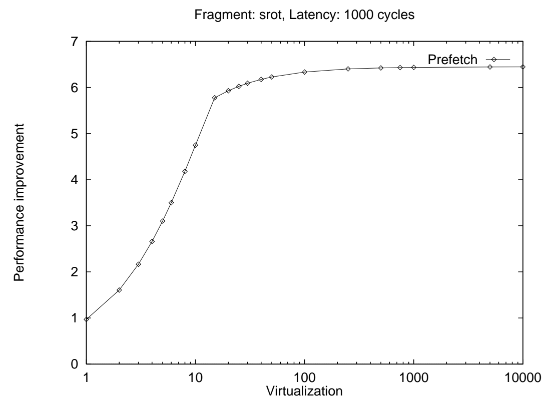

Fragment: rbsor, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch
0
20
40
60
80
100
1
10
100
1000
10000

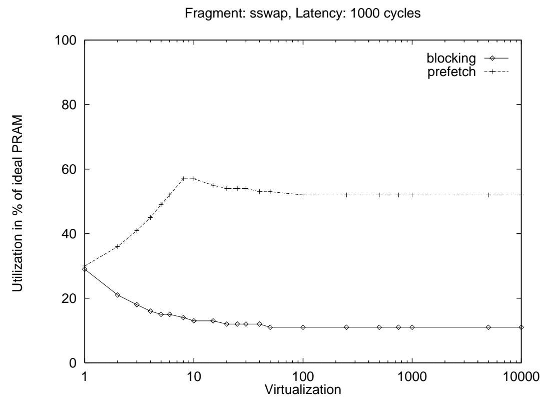

Fragment: rbsor, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch
1
1.5
2
2.5
3
3.5
4
4.5
5
1
10
100
1000
10000

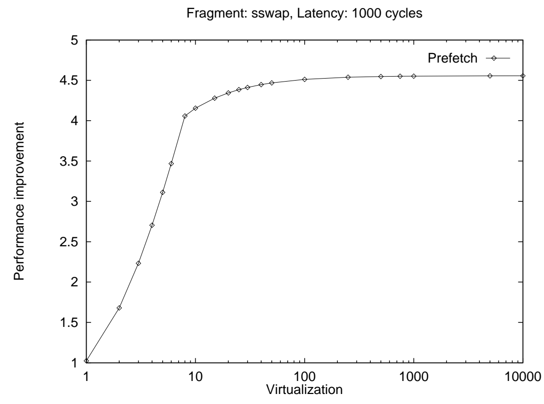

Fragment: srotg, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch
0
20
40
60
80
100
1
10
100
1000
10000

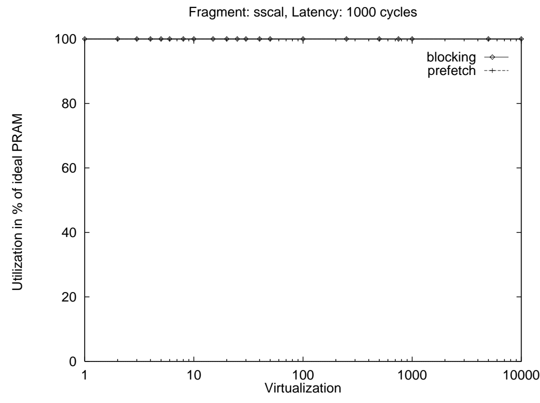

Fragment: srotg, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch
0.9
0.95
1
1.05
1.1
1
10
100
1000
10000

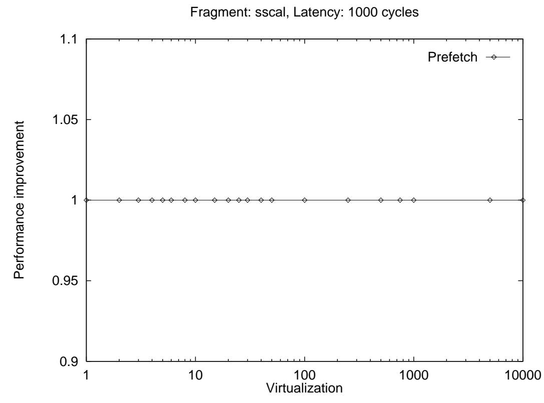

Fragment: srot, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization
0
20
40
60
80
100
1
10
100
1000
10000

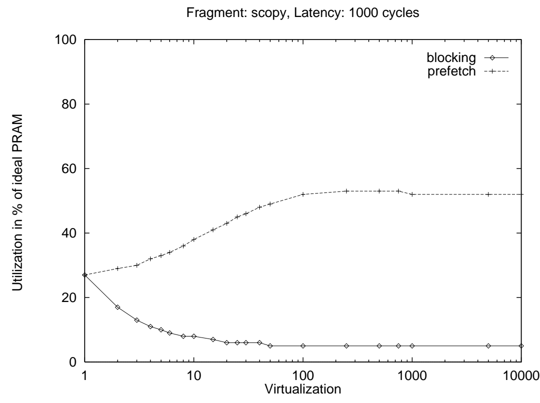

Fragment: srot, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization
0
1
2
3
4
5
6
7
1
10
100
1000
10000

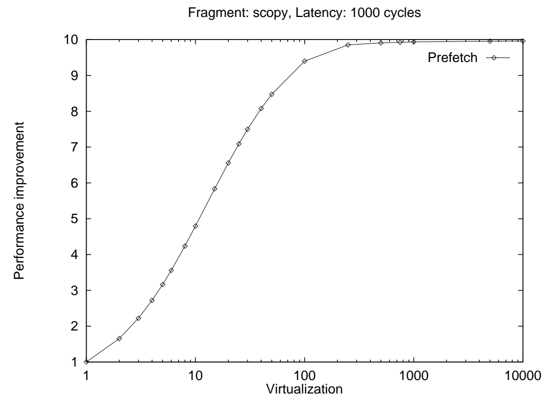

Fragment: sswap, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization
0
20
40
60
80
100
1
10
100
1000
10000

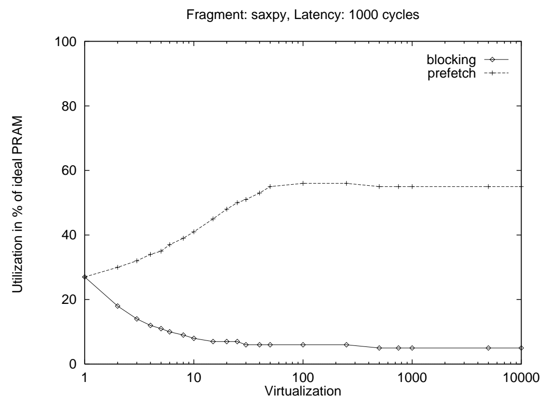
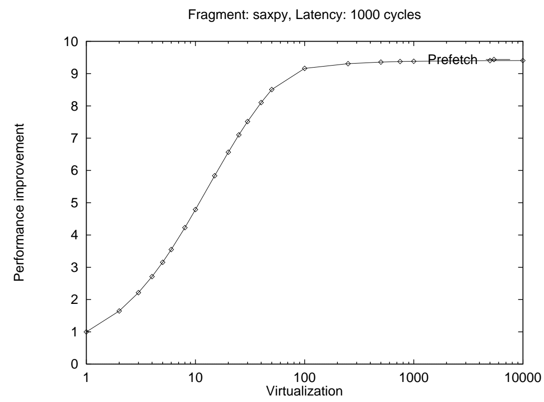
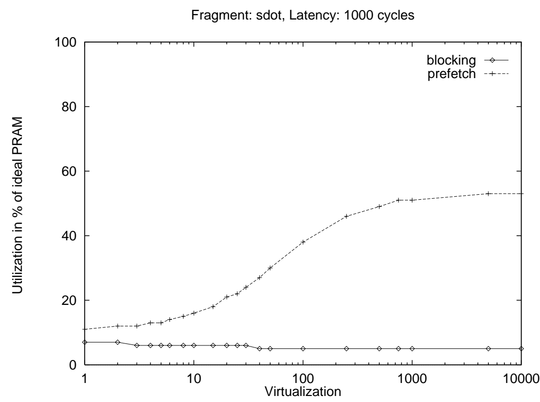
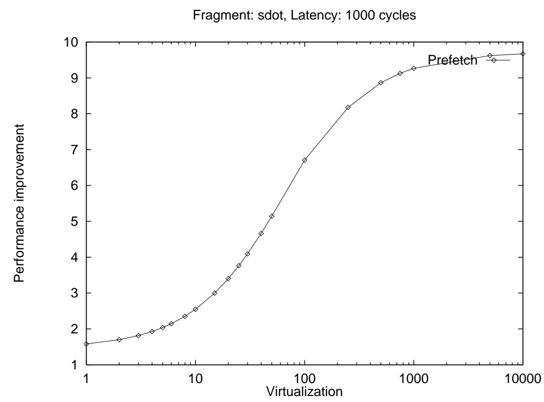
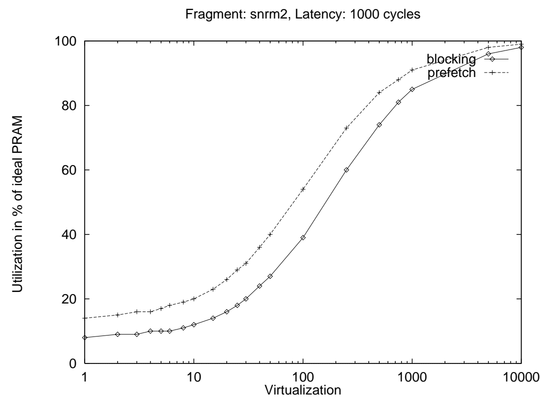
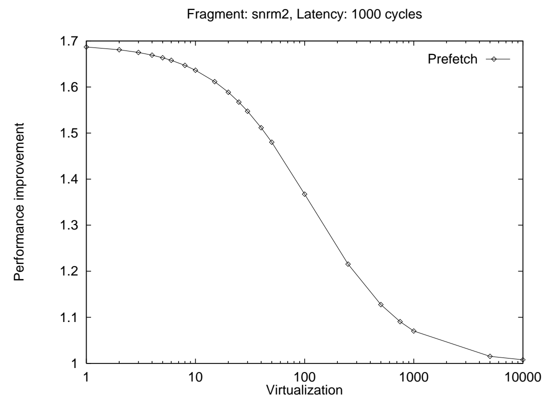
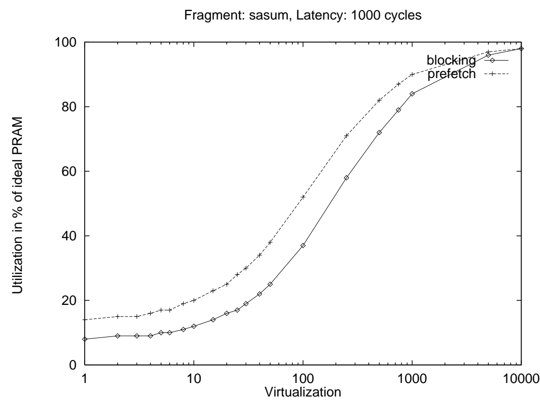
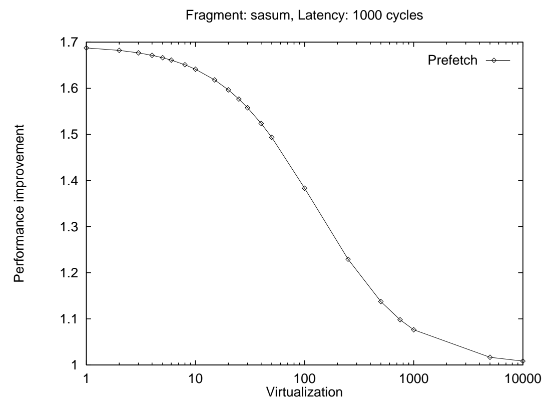

Fragment: sswap, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization
1
1.5
2
2.5
3
3.5
4
4.5
5
1
10
100
1000
10000

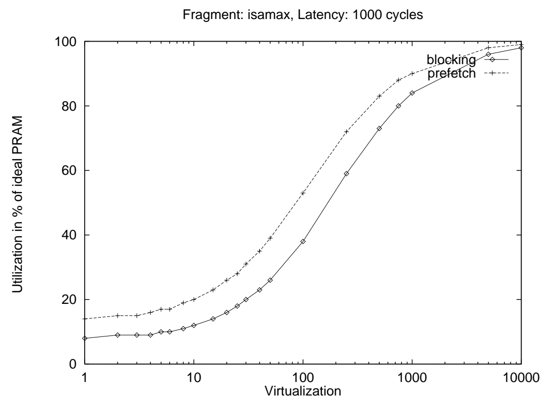

Fragment: sscal, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization
0
20
40
60
80
100
1
10
100
1000
10000

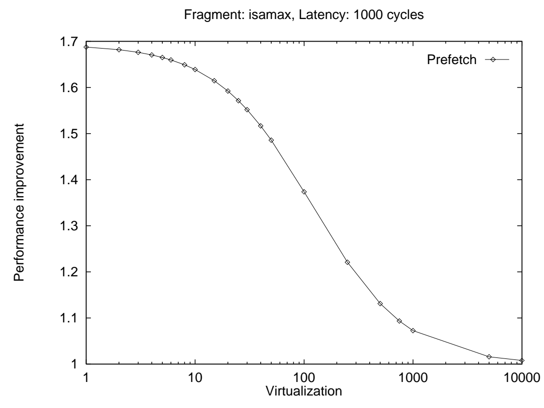

Fragment: sscal, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization
0.9
0.95
1
1.05
1.1
1
10
100
1000
10000


Fragment: scopy, Latency: 1000 cycles
blocking
prefetch
Utilization in % of ideal PRAM
Virtualization
0
20
40
60
80
100
1
10
100
1000
10000


Fragment: scopy, Latency: 1000 cycles
Prefetch
Performance improvement
Virtualization
1
2
3
4
5
6
7
8
9
10
1
10
100
1000
10000


Fragment: saxpy, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

Fragment: saxpy, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

Fragment: sdot, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

Fragment: sdot, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

Fragment: snrm2, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

Fragment: snrm2, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch

Fragment: sasum, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch

Fragment: sasum, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch


Fragment: isamax, Latency: 1000 cycles
Utilization in % of ideal PRAM
Virtualization
blocking
prefetch
0
20
40
60
80
100
1
10
100
1000
10000


Fragment: isamax, Latency: 1000 cycles
Performance improvement
Virtualization
Prefetch
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1
10
100
1000
10000