Tech. Rep. 432, Computer Science Dept., Univ. of Rochester, July 1992

A Trace-Based Comparison of Shared Memory Multiprocessor Architectures *

William J. Bolosky and Michael L. Scott
Computer Science Department
University of Rochester
Rochester, NY 14627-0226
{bolosky,scott}@cs.rochester.edu

Abstract

There are three major classes of MIMD multiprocessors: cache-coherent machines, NUMA (non-uniform memory reference) machines without cache coherence, and distributed-memory multicomputers. All three classes can be used to run shared-memory applications, though the third requires software support in order to do so, and the second requires software support in order to do so well. We use trace-driven simulation to compare the performance of these classes, in an attempt to determine the effect of various architectural features and parameters on overall program performance. For those systems whose hardware or software supports both coherent caching (migration, replication) and remote reference, we use optimal off-line analysis to make the correct decision in all cases. This technique allows us to evaluate architectural alternatives without worrying that the results may be biased by a poor data placement policy. We find that the size of the unit of coherence (page or cache line) is the dominant factor in performance; that NUMA systems can have performance comparable to that of cache coherent machines; and that even relatively expensive, software-implemented remote reference is beneficial in distributed shared memory machines.

*This work was supported in part by a DARPA/NASA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland, by an IBM summer student internship, by a Joint Agreement for Loan of Equipment (Number 14520052) between IBM and the University of Rochester, and by the National Science Foundation under Institutional Infrastructure grant number CDA-8822724.

1 Introduction

There currently exist many multiprocessor designs, many of which fall into a small number of broad architectural classes. Deciding which of these classes will provide the best price/performance in large configurations for various applications is a major open problem. It is also a difficult problem to address, for several reasons. Studies of individual designs often make different assumptions about the underlying hardware technology, the type or quality of systems software, or the target applications. These factors easily obscure the impact of true architectural differences. Even in studies that hold these factors constant across a range of architectures, data placement policies (e.g. for cache coherence) may make better decisions on one machine than on another, thereby biasing the results.

The aim of this study is to evaluate the relative merits of the three principal architectural alternatives for shared-memory parallel processing: cache-coherent multiprocessors, non-uniform memory access (NUMA) multiprocessors, and multicomputers running distributed shared-memory software. To avoid the problems just cited, we employ *optimal off-line analysis* in trace-driven simulation. Specifically, the performance used for comparison of each machine model is that which would be achieved if every data placement policy decision were made correctly. Depending on the architecture, a policy may choose at each step of a trace to access a datum in (one of) its current location(s), or to copy it to a closer location, possibly invalidating existing copies. In addition, it may choose to modify the policy to be followed in the event of future references, e.g. by enabling or disabling caching, providing that the particular architecture being simulated allows this.

All of the machine models are based upon a set of components of equal speed. All have similar processors and caches, an inter-processor memory connection medium of equal bandwidth and latency, and for those machines that have it, remote memory reference hardware of the same speed. The differences lie in the way in which the components are used: do they provide hardware support for remote memory access; do they implement coherency in hardware; what size are the cache lines or pages?

We present a simple model of memory cost for the execution of a program on a machine. This cost includes the latency of all references to data and the overhead of all data replication or migration mechanisms implemented in software. It does not include contention. By

using optimal analysis, we ensure that memory cost depends only on the behavior of the program and on the options available to the (optimal) data placement policy. While it is not possible for a real system to make optimal choices in all cases, there is evidence that a properly designed policy can do quite well [6]. Moreover, knowing the performance of an optimal policy can inform a decision as to whether it is worth searching for good, realizable policies for a given architecture. Further discussion of off-line optimal analysis, together with a formal presentation of the technique, can be found in a companion paper [5].

The goal of the analysis is to shed light on a number of questions: What is the relative performance of cache-coherent, NUMA, and distributed memory machines when running shared-memory applications? How useful is the ability to access remote locations without performing migration or replication? Does remote reference need to be provided in hardware, or is an implementation based on catching page faults sufficient? How important is it to have small coherent cache lines or software managed pages, rather than large ones? How much does application design affect the answers to these questions?

Definitive answers, of course, will not come from a single study. Our results apply only to the extent that technological parameters approximate those described in section 3.2, that machines are programmed in a shared-memory style, and that latency dominates contention as a contributor to memory system cost.

That said, we find that while only the most well-behaved programs can reasonably be executed on distributed memory machines, most of the programs that perform well on cache-coherent machines perform only marginally worse on NUMA machines. The block size of data placement (i.e. the size of the cache line or page) appears to be the dominant factor in all three classes of machines; smaller blocks perform better than larger ones (within reason) in most applications, despite the fact that communication start-up latencies make moving the same amount of data much more expensive for smaller blocks. Remote references do little to improve the performance of machines with small, hardware-coherent cache lines, but they can improve the performance of distributed-memory machines dramatically, even when implemented in software.

The following section describes the applications and tracing methodology. Section 3 describes the machine models. Section 4 describes the experiments and the results thereof. Section 5 summarizes these results and presents conclusions.

2 Applications and Traces

2.1 Trace Collection

The traces used in this paper were collected on an IBM ACE Multiprocessor Workstation [13] running the Mach operating system [1]. The ACE is an eight processor machine in which one processor is normally used only for processing Unix system calls and the other seven run application programs.

Traces were collected by single-stepping each processor and decoding the instructions to be executed, to determine if they accessed data. Instruction fetches were not recorded. The single-step code resides in the kernel's trap handler, where it maintains a buffer of memory reference data. When that buffer fills, the tracer stops the threads of the traced application and runs a user-level process that empties the buffer into a file. To avoid interference by other processes, the applications were run in single-user mode, with no other system or user processes running. Since there is only a single, system-wide buffer of references, the traces are statically interleaved: the ordering of references between processors is determined at trace collection time, and is virtually the same as that made by the processors as they executed the traced program.

Modifying the kernel's trap handler results in better performance (and therefore longer traces) than would have been possible with the Mach exception facility or the Unix **ptrace** call. Execution slowdown is typically a factor of slightly over 200. Experiments by Koldinger et al. suggest that the impact of this *trace dilation* is relatively minor [15]. A potentially more serious problem than trace dilation is the possibility that changes in the architectural parameters under study will invalidate the trace that drives a simulation; i.e., the ordering of references from different processors, and in fact the actual set of locations accessed in a non-deterministic program, could depend on the machine characteristics being changed. In an attempt to assess the importance of this issue, we performed a series of sensitivity analyses to estimate error bounds. We deliberately induced changes in instruction interleaving and injected delays for kernel execution (e.g. of software page coherence policies). In all cases the effects described in section 4 were well above the noise.

Programs on large-scale machines will likely use distributed data structures [16] or special hardware instructions [14] to eliminate contention due to busy-wait synchronization.

Our test applications are more naive. To eliminate potential simulation anomalies, we deleted from the traces all references due to busy-wait synchronization.

2.2 The Application Suite

The application suite includes a total of fourteen applications, written under three different programming systems by authors at several institutions. The suite is described in more detail elsewhere [6]. Briefly, the programming styles are EPEX [18], a parallel extension to FORTRAN in which the programmer specifies DO loops to execute in parallel; Presto[4], a parallel programming system based on C++ in which there are potentially many more threads than processors; and C-Threads [10], a multi-threaded extension to C in which there is generally one thread per processor. `mp3d`, `cholesky` and `water` are applications from the SPLASH benchmark suite[17]. The other C-Threads applications were written for either the butterfly or ACE machines by researchers at various institutions. EPEX and Presto applications are indicated by having their name prefixed with "`e-`" and "`p-`, respectively." Applications without a prefix are written in C-Threads.

Table 1 shows the sizes of our traces in millions of references. Presto and EPEX have regions of memory that are addressable by only one thread. References to these explicitly private regions are listed in the column named "Private Refs," and are not represented under "References."

2.3 A Model Of Program Execution Cost

Our concept of the cost of execution of a program roughly corresponds to the total amount of time across all processors spent waiting for data memory. This includes both hardware latency and compute time devoted to software-implemented data placement policies. It does not include the time spent doing other things, such as performing arithmetic operations on data in registers, or handling instruction cache misses. Likewise, it does not correspond to wall-clock time; since it is the sum of time spent over all processors, it is more akin to total work performed.

This description considers only the references made to a single block. Real applications use more than one block, but the behavior of separate blocks is independent in the model,

Application	References	Private Refs
e-simp	27.8	109
e-hyd	49.8	445
e-nasa	20.9	326
gauss	270	0
chip	412	0
bsort	23.6	0
kmerge	10.9	0
plytrace	15.4	0
sorbyc	105	0
sorbyr	104	0
matmult	4.64	0
mp3d	19.5	0
cholesky	38.6	0
water	81.8	0
p-gauss	23.7	4.91
p-qsort	21.3	3.19
p-matmult	6.74	.238
p-life	64.8	8.0

Table 1: Trace Sizes and Breakdowns (in millions of data references)

and considering only one simplifies the presentation without loss of generality. In practice, we treat traces as independent streams of references to the various blocks, and add together the cost of all blocks at the end of the run.

A machine is characterized by a set of processors and some parameters that represent the speed of various operations. The parameters are r, R and the block size. r is the amount of time that it takes a processor to read a single word from another processor's cache; it is infinite in machines that do not permit such references. R denotes the length of time that it takes for one processor to copy an entire block from another processor's cache. R and r are measured in units of time: one time unit represents the duration of a processor's reference to its local cache.

At each point in the trace, the block must be at some non-empty set of locations. Any processor that lacks a local copy at the time of a reference must make a copy or pay the cost of a remote reference (r). To maintain coherence, a block is required to be in exactly one location at the time of a write (the model does not handle write-update schemes). When

reads happen the block may be replicated to some or all of the processors. The cost per replication is R.

Deciding when to replicate or migrate, and when to use remote reference, is the fundamental trade-off made by a placement policy. Our notion of optimality characterizes the best performance that can be achieved in any system that requires that all copies of a data block be consistent at all times and that does not migrate threads or update multiple copies. It is not in general possible to find an optimal placement at run time. Our algorithm for computing an optimal policy employs dynamic programming, and executes in $O(x + py)$ time, where x is the number of reads, y the number of writes and p the number of processors.[1] The essential insight in the algorithm is that after each write a block must be in exactly one place. To first approximation, we can compute the cheapest way to get it to each possible place given the cheapest ways to get it to each possible place at the time of the previous write. If there are reads between a pair of writes, then a block may be replicated during the string of reads; whether this replication occurs depends on the starting and ending locations of the block and the number of reads made by each processor during the interval.

3 Machine Models

Since the aim of this work is to compare the performance of a suite of programs on several classes of multiprocessor architectures, it is important to choose machines that are representative of these classes, but which are in other ways balanced. This is necessary to avoid tainting the results with effects unrelated to the cache and memory systems used. Since such a large group of well-balanced machines does not exist in real implementations (or even concrete designs), the models used here extrapolate performance from current machines.

3.1 The Machines Models

We consider five basic machine types. Four of these types are obtained from the four pairs of answers to two questions. The first question is: does the machine support single-word

[1] Paul Dietz has discovered that a slightly more complicated algorithm will run in time linear in the length of the trace.

	Hardware Coherence	Software Coherence
Remote Access Hardware	CC+	NUMA
No Remote Access Hardware	CC	DSM

Table 2: Machine Types Considered

references to remote memory (caches)? The second is: does hardware implement all of the block movement decisions and operations, or does the operating system kernel need to be invoked in some cases? Table 2 shows the four pairs of answers to the questions, and shows the name used here for each machine.

"CC" stands for coherently cached, "NUMA" for non-uniform memory access, and "DSM" for distributed shared memory. The "+" in CC+ indicates that this model contains a feature (remote references) not normally present in cached machines.[2] The NUMA machine includes hardware to remotely reference memory in another processor's cache, but to initiate a page move operation it requires that a page fault occur and that the kernel intervene to initiate the operation to fetch the page from remote memory. CC+ is similar to NUMA, except that it has hardware support for making the remote move or replicate request, and so does so much faster, and because of this hardware support is likely to be able to use smaller blocks. CC is similar to CC+, except that remote references are prohibited. DSM is similar to NUMA, except that remote references are prohibited. In all our machine models, we assume that caches are of infinite size, and that initial cache-load effects are insignificant.

While DSM does not support remote memory access in hardware, it can implement them in software. The resulting system is named DSM+, and is the fifth machine considered in this paper. A page that is to be accessed remotely is left invalid in the page table, so that every reference generates a fault. The kernel then sends a message to read or write the remote location. The cost of a remote reference will be larger than in the NUMA machine, due to this kernel overhead.

The systems without remote reference capability (CC, DSM) make no placement deci-

[2] Many cache-coherent machines allow caching to be disabled for particular address ranges, forcing processors to access data in main memory. Machines with this capability will have performance between that of CC and CC+.

sions; they always migrate data on a write and replicate on a read. The trace analyzer therefore needs to employ its dynamic programming algorithm only for the CC+, NUMA, and DSM+ systems. The NUMA system model dominates DSM+ in performance: it provides the same capabilities with a lower remote reference cost. Also, the CC+ and NUMA/DSM+ models dominate CC and DSM, respectively, in performance; at the very worst they can ignore their remote reference capability and act like their simpler cousins. On the other hand, if we assume that the cache-coherent machines can use a smaller block size than the other competitors, then neither the software nor hardware systems dominate the other. Reducing the block size will improve performance by reducing false sharing [9, 11, 12] and, more importantly, by reducing the number of references that must be made in order to justify movement of a truly-shared block. At the same time, while the hardware-supported alternatives initiate operations more quickly, the total cost of the large number of operations that would be required to move a page-sized chunk of memory one cache line at a time is much more than the cost to move the whole page for the software-controlled machines.

3.2 Computing Cost Numbers for the Machines

To use the cost model (see Section 2.3) the important characteristics of a machine are: how fast can it move a cache line or page from one processor to another; can it reference a single word remotely, and if so how fast; how big are its cache lines or pages? That is, what are R, r and the block size?

All units of time are expressed in terms of a local cache hit. The baseline processor resembles a MIPS R3000 running at 40 MHz, with one wait state for a local cache hit. Thus, a cost of 1 time unit can roughly be considered to be 50ns: one 25ns processor cycle for the load or store instruction and one waiting. The inter-processor interconnection network in all of the machines is identical in performance: it has a bandwidth of 40Mbytes/sec, and thus requires 2 time units to transmit each four bytes of data, once start-up overhead has been paid. The latency of the network is 50 cost units to get a message from one node to another—100 for a round trip message. The one-way latency is named λ.

The R3000 takes about 130 processor cycles to take a trap and return to the user context [2]. If a cache cycle is two processor cycles, then it costs 65 units to take and return from a trap. A well-coded kernel should be able to determine what action to take within another

ten units, so the time charged for a trap to software to initiate a remote operation is 75 units. Presumably, additional processing (e.g. to modify page tables) can proceed in parallel with the network transfer. On the other hand, a cache controller is likely to be able to decide what action to take on a cache miss in much less time; it costs 2 units for the hardware to decide what to do. These costs are named the "software overhead," o_s, and "hardware overhead," o_h.

The operation of moving a remote cache line or memory page to the local processor consists of several distinct phases: taking the initial miss or fault that begins the operation, determining the location of the memory to be fetched, requesting that the memory be sent, waiting for the memory to arrive (while concurrently updating any TLB, page table or cache directory information that may need it), and returning to the process context. The trap, decision and return cost are discussed in the previous paragraph: they are 75 for software implementation and 2 for hardware. All of the machine models assume that cache lines or pages have both *home* and *owner* locations, and that a distributed directory contains, on each home node, the location of the owner. A remote read from a home node directory traverses the network once in each direction, at a cost of 100 units ($= 2\lambda$). This is followed by a request/response to the owner of the memory, which costs another 100 units of latency, plus the amount of time that the interconnection is busy, at a cost of 2 units per 4 byte word. The sum of these factors gives us $R = \text{page_size}/2 + 4\lambda + o_s$, the remote page/line copy cost for the software-coherent models. In the hardware-coherent models, the home location forwards the line move request to the owner, thus obviating the need for one of the four messages used in the software implementations, and giving $R = \text{page_size}/2 + 3\lambda + o_h$.

The second necessary cost value is r, the time to fetch a single word from a (known) remote memory. Hardware supports this operation in CC+ and NUMA. After an initial o_h hardware set-up cost, retrieving the data requires a request/reply latency time of 2λ. In DSM+, every remote reference must go through the kernel trap mechanism, which sends a message to the (known) holder of the memory, who traps, finds the data and responds. Thus, there are *two* software trap costs (one on each side) plus a request/reply latency. The total cost of a remote reference is $2\lambda + o_h = 102$ in CC+ and NUMA, and $2\lambda + 2o_s = 250$ in DSM+.

The experiments presented in section 4 vary several parameters, most notably the block

Machine	r	R
NUMA	$2\lambda + o_h$	$4\lambda + \text{page_size}/2 + o_s$
CC	—	$3\lambda + \text{page_size}/2 + o_h$
CC+	$2\lambda + o_h$	$3\lambda + \text{page_size}/2 + o_h$
DSM	—	$4\lambda + \text{page_size}/2 + o_s$
DSM+	$2\lambda + 2o_s$	$4\lambda + \text{page_size}/2 + o_s$

Table 3: Formulas for computing model parameters

Machine	Block Size	r	R	R for 512 byte block
NUMA	4K	102	2323	531
CC	64	—	184	408
CC+	64	102	184	408
DSM	4K	—	2323	531
DSM+	4K	250	2323	531

Table 4: Relevant parameters for base and 512 byte block machine models

size. Initially, the software coherent machines have a page size of 4 Kbytes and CC and CC+ have a line size of 64 bytes. Table 3 presents the formulas used to compute the model parameters for experiments in which these parameters vary. Table 4 shows the values of the model parameters for our base systems, and also for systems with a block size of 512 bytes. Note that only R is shown for the 512 byte block machines, because r does not depend on the block size.

4 Experimental Results

At this point, the obvious question to ask is: "How well do the applications run on the base machine models?" The results are presented in Figure 1. (This and similar figures present the performance of each of the architectures for each of the applications in term of mean cost per data reference (MCPR). Each application has a set of five bars; these bars are the CC+, CC, NUMA, DSM+ and DSM performance from top to bottom respectively. When an architecture has a MCPR greater than 35, its bar is left open and its cost is recorded in text to the right of the bar: DSM produces an MCPR of 129 on p-life, for example.)

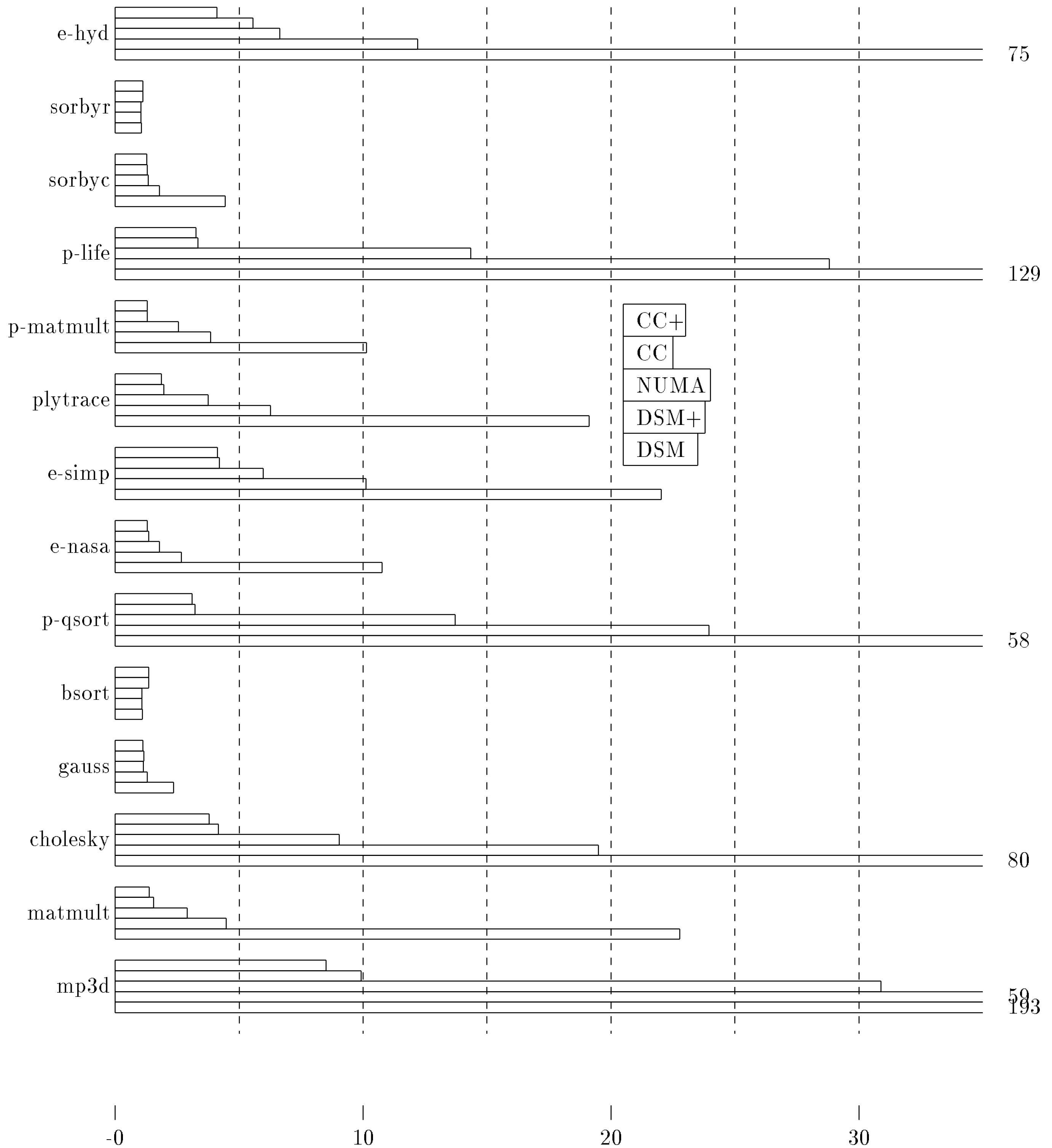

Figure 1: Results for 4K pages and 64 byte lines

In almost all cases the coherently cached machines outperformed the others. This struck us as counterintuitive. Several of the applications display moderately coarse-grain sharing, and the inter-processor latency is so large that the cost of moving a large data structure a (64 byte) line at a time was more than five times the cost of moving it a (4 kilobyte) page at a time, even given the additional software overhead charged to the page-sized systems.

The fact that the CC architectures nearly dominated the others, even considering the extra overhead for moving large regions of memory pointed to the importance of block size. Smaller blocks help performance in two different ways: they allow finer-grained sharing to be profitably exploited (fewer references to a block are required to justify moving it), and they reduce the number of coherence operations caused by false sharing—the co-location in a block of two or more unrelated data objects with different usage patterns. Conversely, reducing the block size increases overhead incurred in moving a fixed amount of data, particularly in the systems we are modeling, in which the remote access latency is high. The fact that CC+ and CC so outperformed the others indicates that the exploitation of finer-grain sharing and reduction of false sharing almost always overcome the increased overhead.

Stated differently, reducing block size, even at the cost of increased overhead, can have significant benefits in performance. These benefits usually overshadow the other architectural decisions, such as whether to support remote references, or whether to manage locality in fast caching hardware or slower software. The overall desirability of small block sizes is largely insensitive to the overhead charged for handling faults in software; even increasing this overhead by an order of magnitude doesn't substantially change the block size effect.

Figure 2 is similar to Figure 1, except that it displays data for 512 byte blocks for all five machines. Differences between the two graphs are striking. Most outstanding is that the DSM architecture achieved significant performance gains: it improved more than an order of magnitude in the case of `p-life`, for example. When faced with fine-grained sharing, DSM has no choice but to migrate the entire block from processor to processor at each "pinging" access. Reducing the block size helps it in two ways: there is less false sharing because there is less opportunity for co-location, and the expense of moving the blocks for the sharing that remains is less.

The equalization of block size at 512 bytes also changes the relative costs of NUMA and

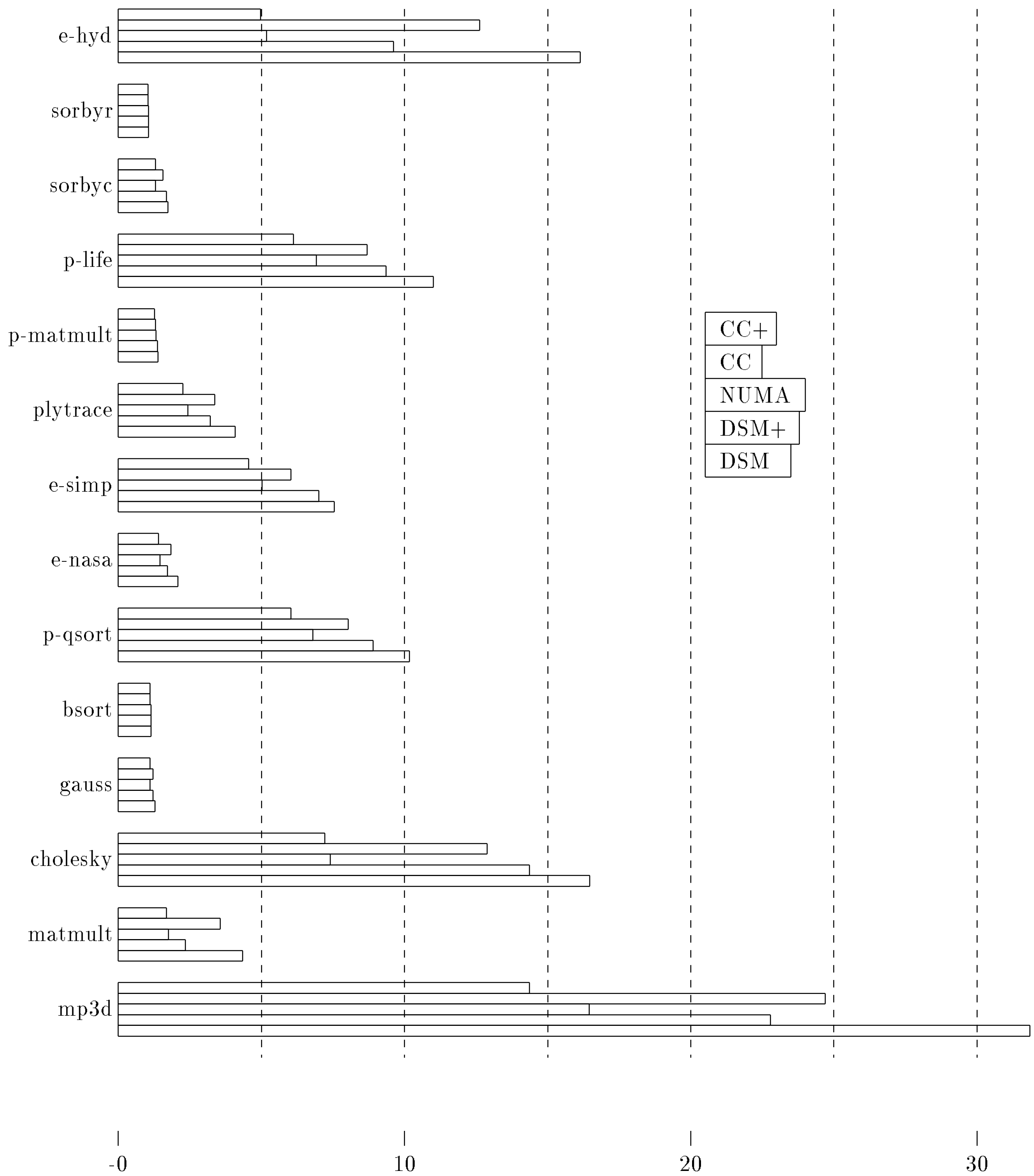

Figure 2: Results for 512 byte pages and 512 byte lines

CC over the base machine models in most applications. When block sizes are equal, the difference between NUMA and CC is that NUMA can make remote references, while CC has a block move that is about 30% faster than NUMA. As a result, NUMA is generally better than CC in Figure 2. The only remaining difference between NUMA and CC+ is the latter's faster block transfer. Generally, CC+ does not do that much better than NUMA, which indicates that for 512 byte blocks NUMA's additional overhead is of minor consequence.

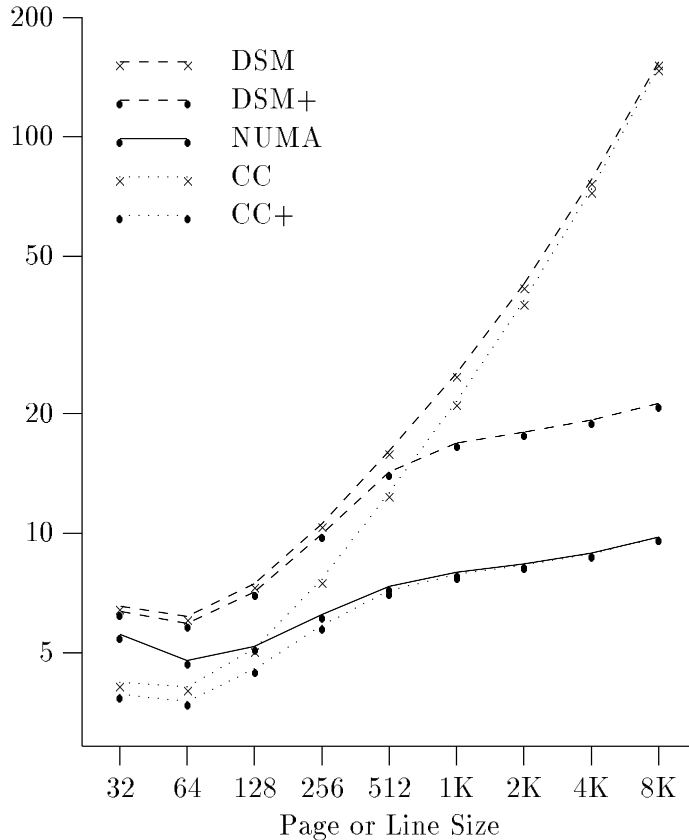

Figure 3: `cholesky` performance versus block size

Figures 3, 4 and 5 show the performance of three of our applications as the block size is varied. These are log-log graphs. `Cholesky` and `qsort` are typical of the rest of the application set in that performance improves significantly with reduction in block size, across the board (excepting the smallest cache line size considered). The value of remote reference (as shown by the distance between DSM and DSM+ and between CC and CC+) increases as the block size increases. This is because the cost of remote reference does not vary with the block size, while the cost of the page moves it replaces does.

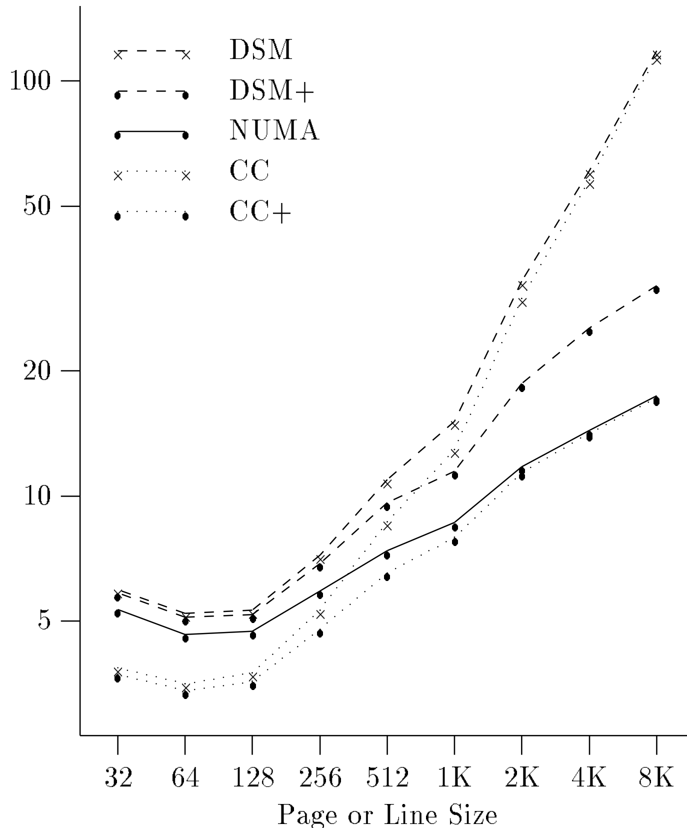

Figure 4: `qsort` performance versus block size

`Sorbyr` is interesting in that it is extremely well-behaved. Most of its memory is used by only one processor after initialization, and the portion that is shared is migratory over relatively long time periods. The size of the portions that migrate is 4K bytes, and thus the curves show a minimum at a block size of 4K. For all machines and all block sizes, `sorbyr` does much better than most of the other applications. Moreover, the shapes of its curves are unique. At small block sizes performance degrades, because the natural size to migrate is 4K, and the overhead of using many high latency operations rather than a single high latency operation to make the migrations dominates the application's overall performance. We expected to see this effect in more applications, but did not. For completeness, graphs of MCPR versus block size for the rest of our applications appear at the end of this paper.

One of the parameter selections that is subject to debate is that of o_s, the software overhead. The value chosen represents an extremely small number of instructions executed by the kernel (about 20, excluding those necessary in an "empty" trap) to decide whether to replicate a page. It may be that this is insufficient time; even if it is possible, it is doubtless

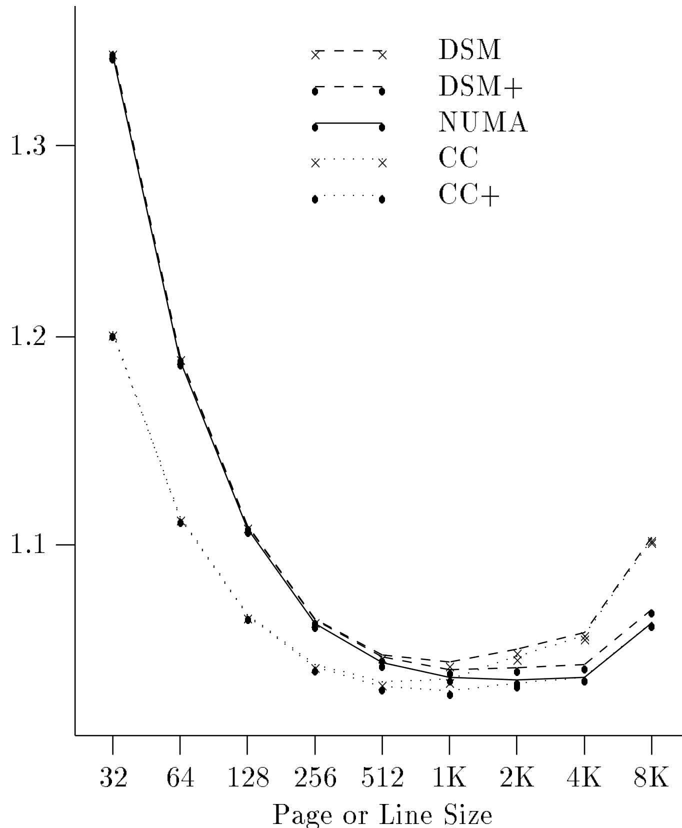

Figure 5: `sorbyr` performance versus block size

more convenient for the kernel writer if more time is allowed. To see how sensitive the experiments are to this parameter, the following experiments observe the result of varying it.

Across the application suite, variations in o_s resulted in two qualitatively different patterns of variation in performance. One pattern is typified by `sorbyr`, shown in Figure 6. The value of o_s in the base machine model is 75. As it increases NUMA, DSM, and DSM+ all show marked, and almost uniform, increases in MCPR. It is important to note the scale of the y axis, however. The only applications to demonstrate this pattern of performance variation were `sorbyr`, `p-matmult`, and `bsort`, all of which have good MCPR values—below 2.2—at all tested values of o_s. In other words, the appearance of the graphs can be deceiving; the quantitative performance degradation is small. CC and CC+ are constant with respect to o_s, because none of their operations require software intervention, and hence none of their model parameters depend on it. They are shown for reference only.

The more common pattern of variation of MCPR with o_s is typified by `e-hyd`, shown

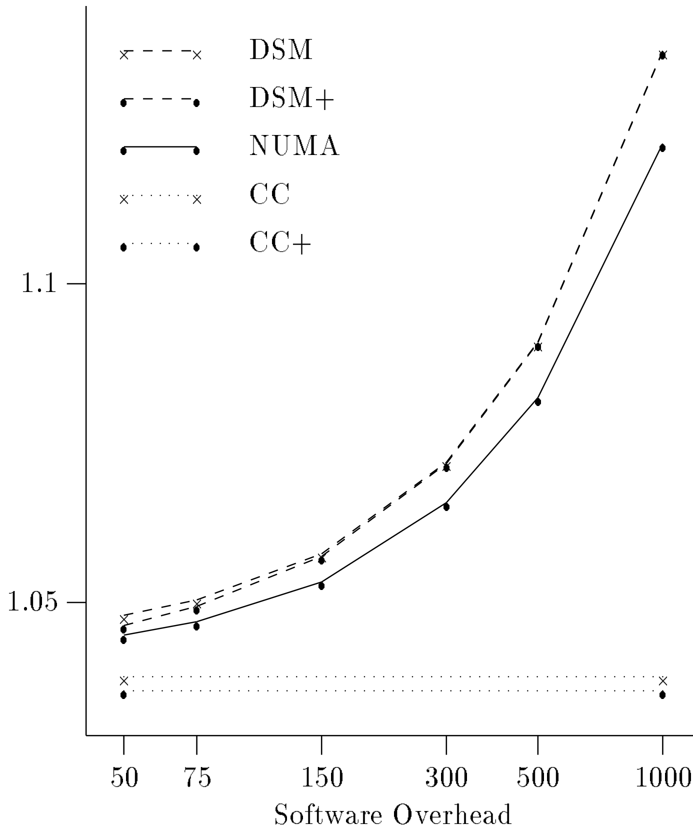

Figure 6: `sorbyr` performance versus o_s for 512 byte blocks

in Figure 7. As o_s increases the performance of NUMA slowly degrades, but not enough to be of much concern. DSM and DSM+ are more severely affected. Since software remote references in DSM+ require *two* traps, the value of remote reference in DSM+ (as evidenced by its benefit over DSM) is quickly reduced, and the two upper curves converge. With no option but to migrate blocks, and with many more migrations occurring at a cost that increases with o_s, performance rapidly degrades. We conclude that fast trap handling is crucial in DSM systems, at least when using a page size as small as 512 bytes. It is less crucial in NUMA systems, though certainly desirable. However, the magnitude of the effect is not such that the assumption of fast trap handling significantly affected the results of the experiments that varied the block size.

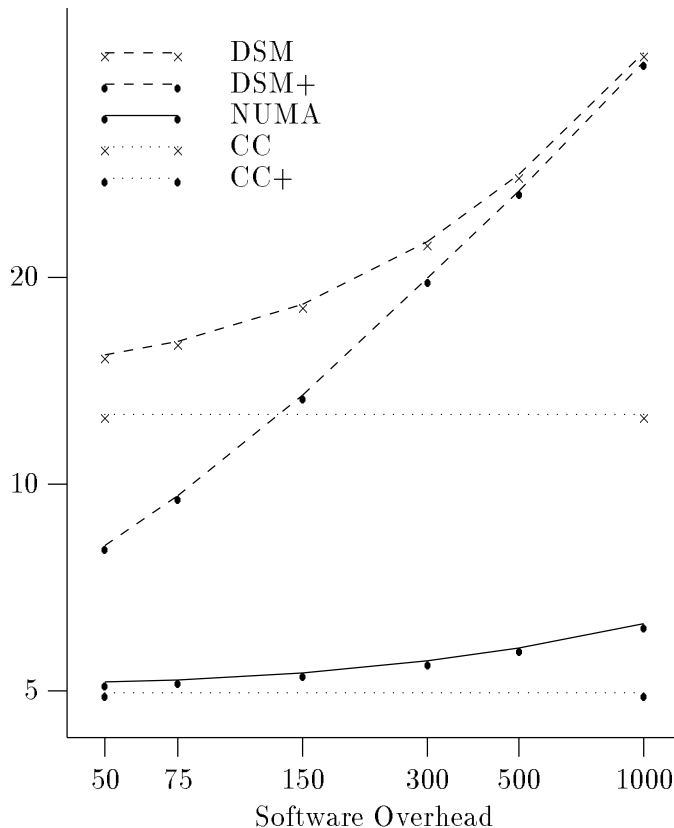

Figure 7: `e-hyd` performance versus o_s for 512 byte blocks

5 Conclusions

This paper presented a model of the cost of data access and locality management in shared-memory multiprocessors, and used it to evaluate the relative performance of alternative multiprocessor architectures executing shared-memory applications. The analysis technique employs trace-driven simulation in conjunction with an optimal off-line data placement policy to avoid any policy bias in the results. It compared five different machine models: cache-coherent multiprocessors with and without remote references; NUMA multiprocessors (with remote references); and shared virtual memory simulations running on top of distributed-memory multiprocessors, with and without software-implemented remote references. All five models share a common technology base, with identical processor, cache, memory, and interconnection speeds.

The results must be considered in the context of assumptions about the hardware technology, application suite, and tracing methodology. Major technology shifts could lead to

different results—a large decrease in network latency, for example, would benefit the cache-coherent machines more than the NUMA or DSM systems, because their smaller block sizes cause them to incur this latency more frequently. Different applications will produce different results as well. Our applications were all designed for NUMA machines or bus-based cache-coherent multiprocessors. They have been modified to represent the use of smart synchronization algorithms, but even so they do not in general display the very coarse-grain sharing most suitable to distributed-memory multicomputers. They are typical of the programs designers would *like* to be able to run on DSM systems. The Munin group [3, 7] at Rice University has explored the use of program annotations to adapt shared memory programs to run well on DSM systems, e.g. by indicating intervals in which sequential consistency is not required. We did not attempt to evaluate these techniques, but our results suggest that something like them is required.

The cost model represents memory access latency and placement overhead only; it does not capture contention. This assumption appears to be fair in well-designed machines and applications. It may penalize the NUMA and DSM models to some degree, since large amounts of contention would be likely to impact the cache-coherent machines most heavily. As in most trace-based studies, caches are of infinite size, and cold-start effects are ignored. Since we use optimal behavior for machines which make policy choices (i.e., the machines with remote reference), their actual performance will be somewhat less than that reported here; on the other hand, the machines without remote reference make no policy choices, and so their numbers show no similar bias. Finally, we ignore the fact that changes in architectural parameters should result in different traces. Experiments indicate [5, 15] that the results are relatively insensitive to the kinds of trace changes that ought to be induced.

With these caveats in mind, our principal conclusions are as follows:

- Block size is the dominant factor in shared-memory program performance. Slopes at the low ends of the graphs suggest that even machines that incur large per-block data movement overheads (i.e. the NUMA and DSM systems) will still benefit from reduction of the block size. Block sizes in the 64 to 128 byte range seem to be sufficiently small that further reductions have little benefit. For particularly well-behaved applications (e.g. `sorbyr`), reductions below the 128-byte level can noticeably *reduce* performance.

- Machines with coherent caching hardware tend to do somewhat better than those without. However, NUMA-style machines, particularly when run with small block sizes, tend to perform comparably, and in some cases outperform the traditional cached machines that do not have remote reference. NUMA machines remain a viable architectural alternative, particularly if they can be built significantly more cheaply than cache-coherent machines.

- The value of remote reference depends on the size of the block being used. For the block sizes typically employed in paged machines, it can yield significant performance improvements. Well coded remote reference software is a promising option for DSM machines, and merits experimental implementation. Conversely, remote reference is unlikely to benefit cache-coherent machines enough to warrant the expense of building it, at least when the line size is small.

- Straight-forward distributed shared memory systems with large page sizes do not appear competitive as a base for generic shared-memory programs. Their performance was acceptable only for the most well-behaved applications, with very coarse-grain sharing. To run a more general set of applications, DSM systems will probably need to rely on Munin-style program annotations, smaller page sizes (to minimize false sharing), or both.

These conclusions suggest several avenues for future work in the field. Dubnicki and LeBlanc have proposed [11] that cache-coherent machines vary their block size adaptively, in response to access patterns; our results suggest that this is a promising idea. Hybrid architectures such as Paradigm [8], which employ bus-based hardware coherence in local clusters and software coherence among clusters, also appear to be promising; hardware coherence is easy (and cheap) for bus-based machines, and the performance of our NUMA model is acceptable for many applications. We plan to investigate the point of diminishing returns for reduction of block size for the various architectures. We have noted (in section 3.1) that reductions in block size can worsen performance by increasing the cost of moving large amounts of data, and can improve performance both by decreasing false sharing and by increasing the probability that a block will be used enough to merit replication before it is invalidated via true sharing. We are pursuing experiments to isolate these factors and measure them independently.

The burden of locality management currently rests with the programmer. Its importance is obvious in the wide performance differences among our applications, but the task can be onerous, particularly to novice programmers. There is a strong need to reduce this burden. Fairly simple diagnostic tools could identify data structures for which coherency imposes a substantial cost at run time, thereby facilitating manual program tuning. More ambitiously, compilers or run-time packages might use application-specific knowledge to partition and place data appropriately, thereby increasing locality.

Acknowledgments

Bob Fitzgerald was the principal force behind the ACE Mach port, and has provided valuable feedback on our ideas. Rob Fowler and Alan Cox helped with application ports and tracing, and also provided good feedback.

Most of our applications were provided by others: in addition to the PLATINUM C-Threads applications from Rob Fowler and Alan Cox at Rochester, the Presto applications came from the Munin group at Rice University; the SPLASH applications from the DASH group at Stanford University; the EPEX applications from Dan Bernstein, Kimming So, and Frederica Darema-Rogers at IBM; and `plytrace` from Armando Garcia. Our thanks to Armando and to Colin Harrison and IBM for providing the ACE machines on which the traces were made.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. Summer 1986 USENIX*, July 1986.

[2] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proc. 4th Intl. Conf. on Arch. Sup. for Prog. Lang. and Operating Sys.*, pages 108–120, April 1991.

[3] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *ACM SIGPLAN Symp. on Princ. and Practive of Par. Prog. (PPoPP), SIGPLAN Notices 25(3)*, pages 168–176, March 1990.

[4] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[5] W. J. Bolosky and M. L. Scott. Evaluation of Multiprocessor Memory Systems Using Off-Line Optimal Behavior. *The Journal of Parallel and Distributed Computing*, August 1992. Also URCS Tech. Rpt. 403.

[6] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proc. 4th Intl. Conf. on Arch. Sup. for Prog. Lang. and Operating Sys.*, pages 212–221, 1991.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. 13th Symp. on Operating Systems Principles*, pages 152–164, 1991.

[8] D. R. Cheriton, H. A. Goosen, and P. D. Boyle. Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture. *IEEE Computer*, pages 33–46, February 1991.

[9] D. R. Cheriton, H. A. Goosen, and P. Machanick. Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience. In *Proc. of the Intl. Symp. on Shared Memory Multiprocessing*, pages 109–118, April 1991.

[10] E. Cooper and R. Draves. C Threads. Technical report, Carnegie-Mellon University, Computer Science Department, March 1987.

[11] C. Dubnicki and T. J. Leblanc. Adjustable Block Size Coherent Caches. In *Proc. 19th Intl. Symp. on Comp. Arch.*, 1992.

[12] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proc. 1991 Intl. Conf. on Parallel Processing*, pages 377–381, 1991. Volume I.

[13] A. Garcia, D. Foster, and R. Freitas. The Advanced Computing Environment Multiprocessor Workstation. Research Report RC-14419, IBM T.J. Watson Research Center, March 1989.

[14] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *3rd Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, pages 64–75, April 1989.

[15] E. J. Koldinger, S. J. Eggers, and H. M. Levy. On the Validity of Trace-Driven Simulations for Multiprocessors. In *Proc. 18th Intl. Symp. on Comp. Arch.*, pages 244–253, 1991.

[16] J. M. Mellor-Crummey and M. L. Scott. Synchronization without Contention. In *Proc. 4th Intl. Conf. on Arch. Sup. for Prog. Lang. and Operating Sys.*, pages 269–278, 1991.

[17] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, 1991.

[18] J. Stone and A. Norton. *The VM/EPEX FORTRAN Preprocessor Reference.* IBM, 1985. Research Report RC11408.

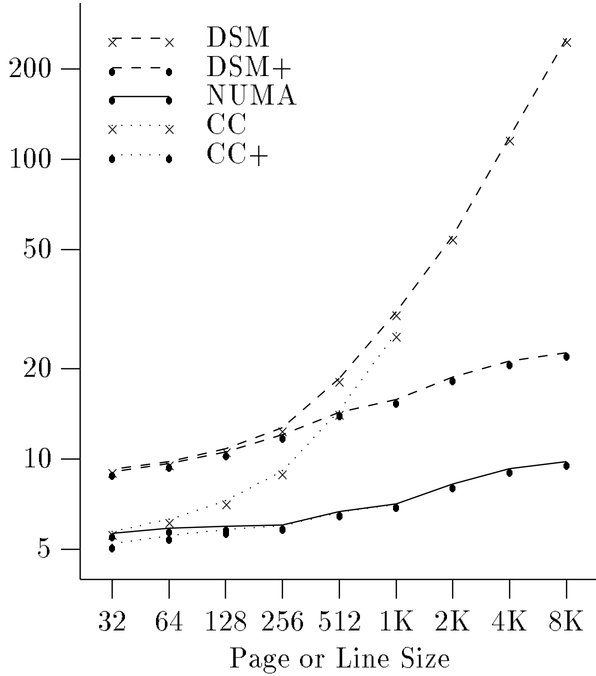

Figure 8: e-fft

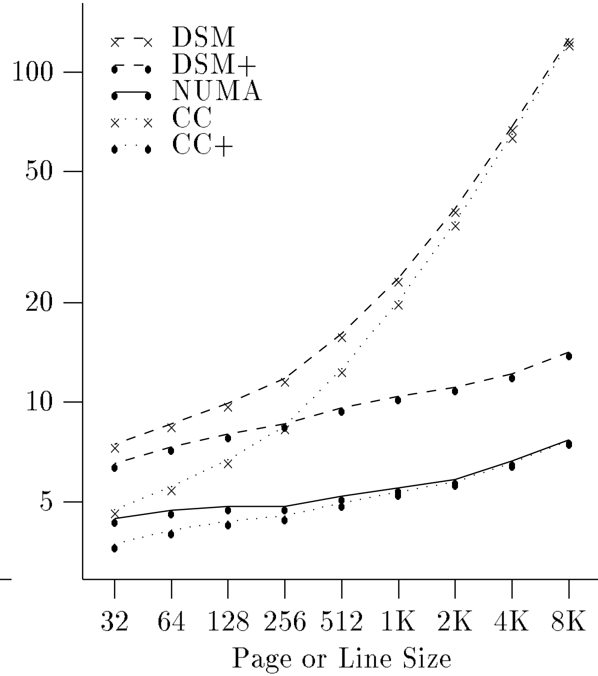

Figure 10: e-hyd

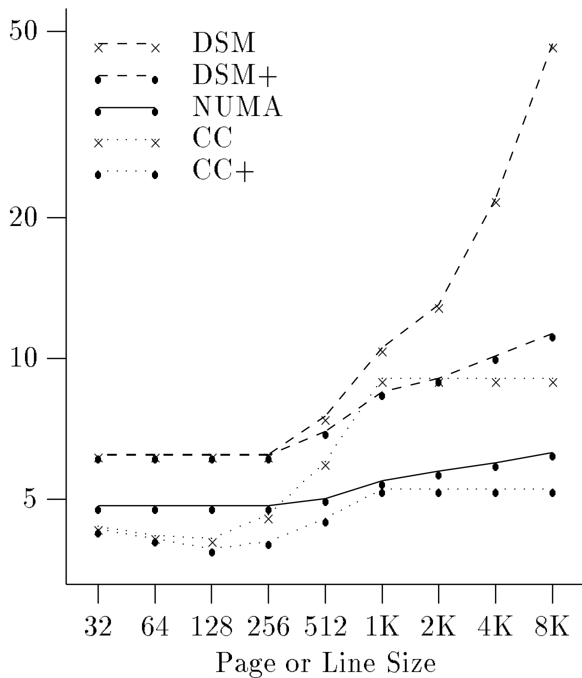

Figure 9: e-simp

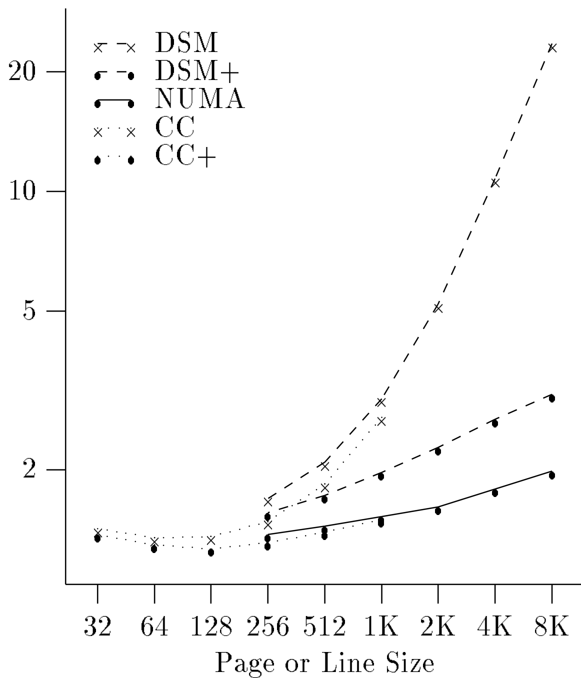

Figure 11: e-nasa

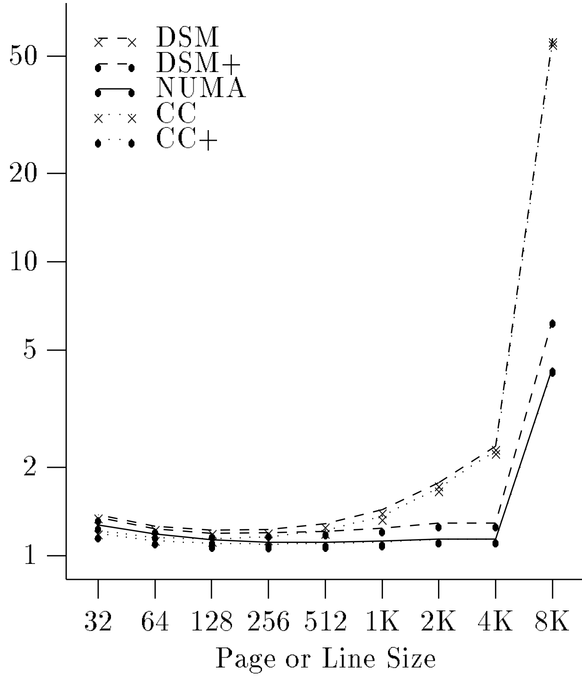

Figure 12: gauss

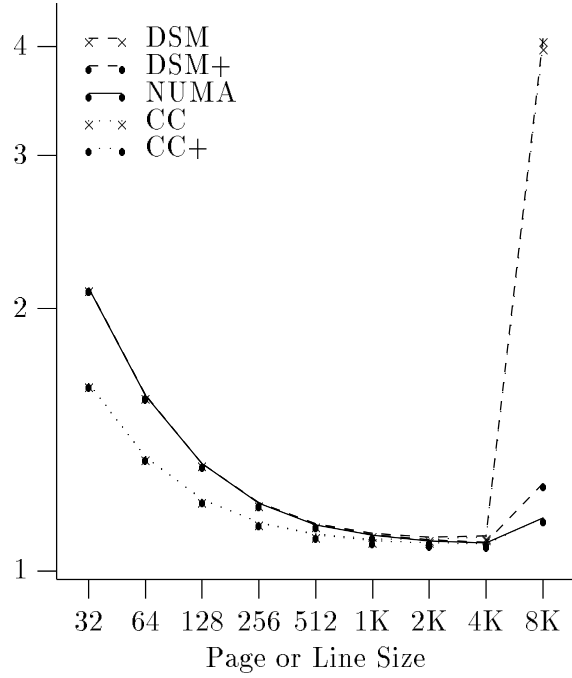

Figure 14: bsort

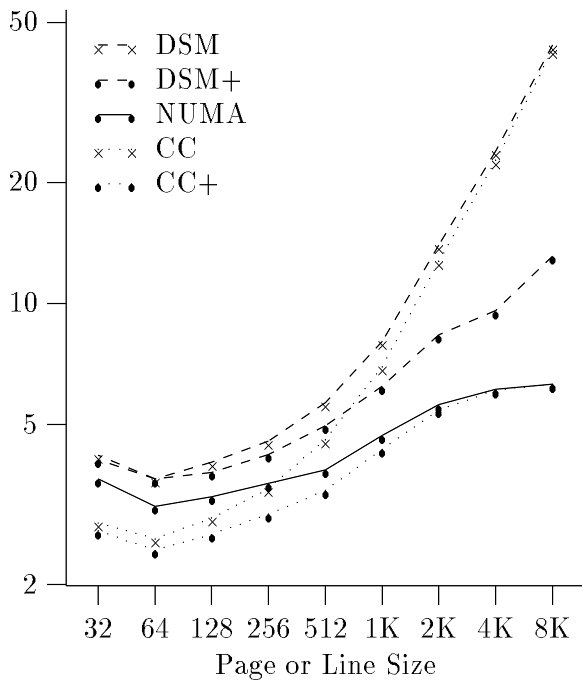

Figure 13: chip

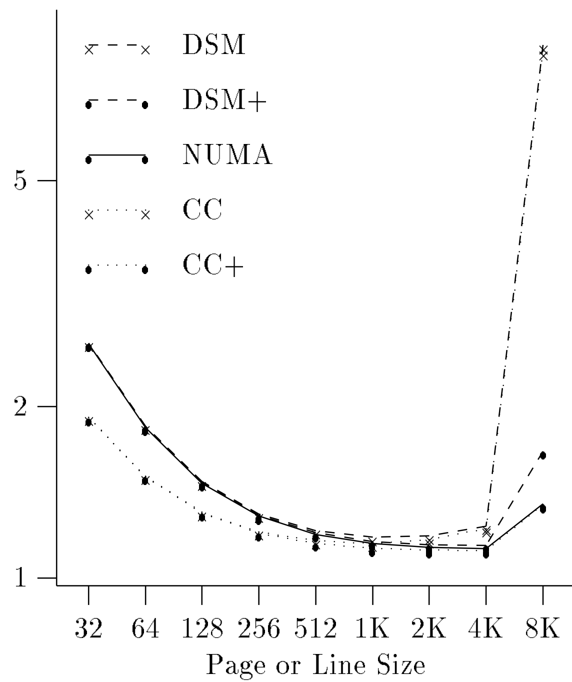

Figure 15: kmerge

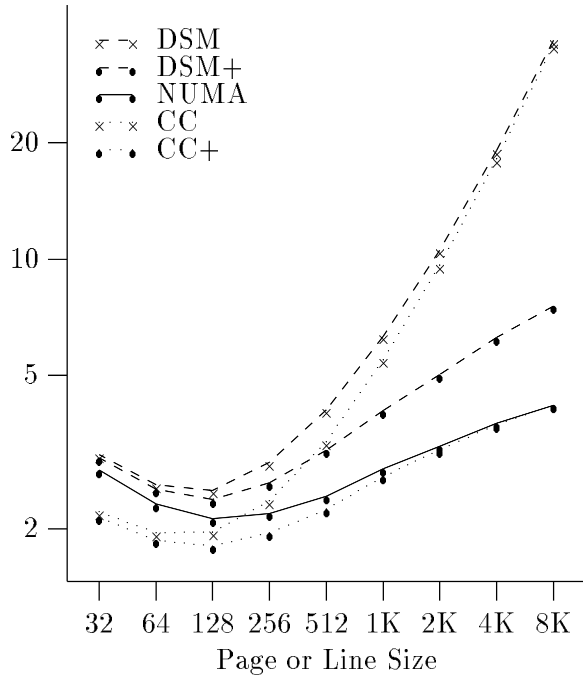

Figure 16: plytrace

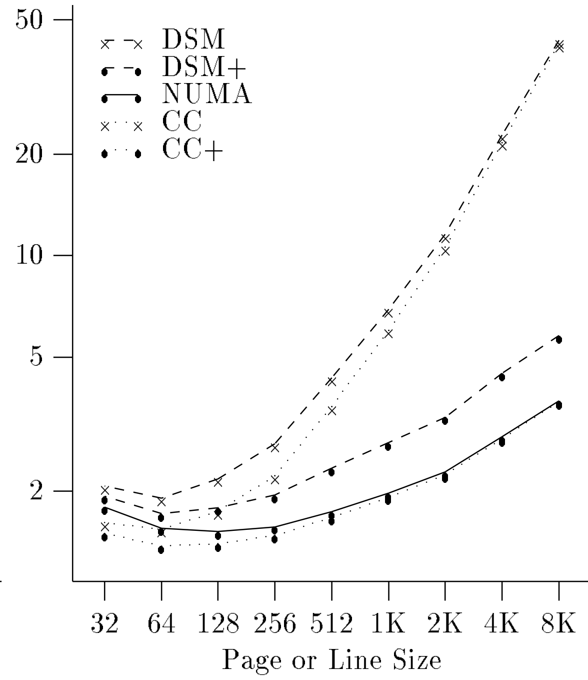

Figure 18: matmult

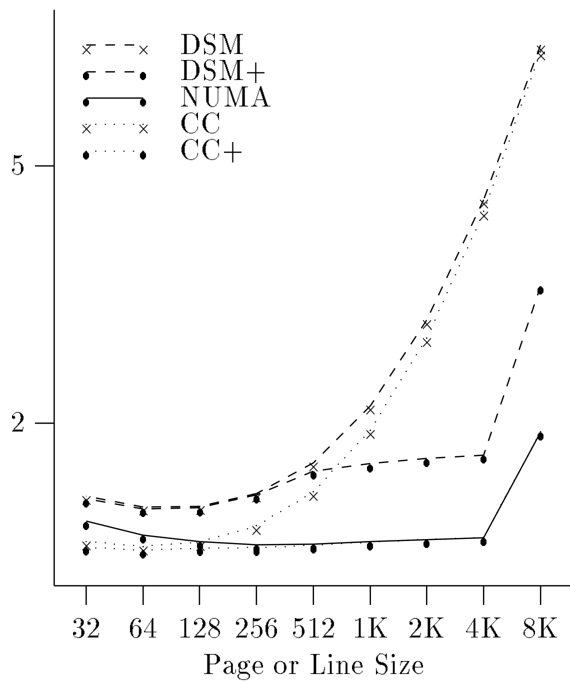

Figure 17: sorbyc

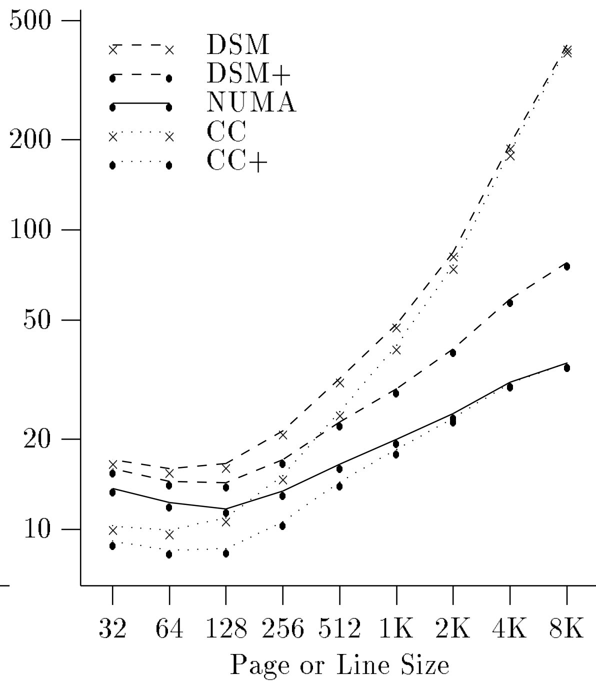

Figure 19: mp3d

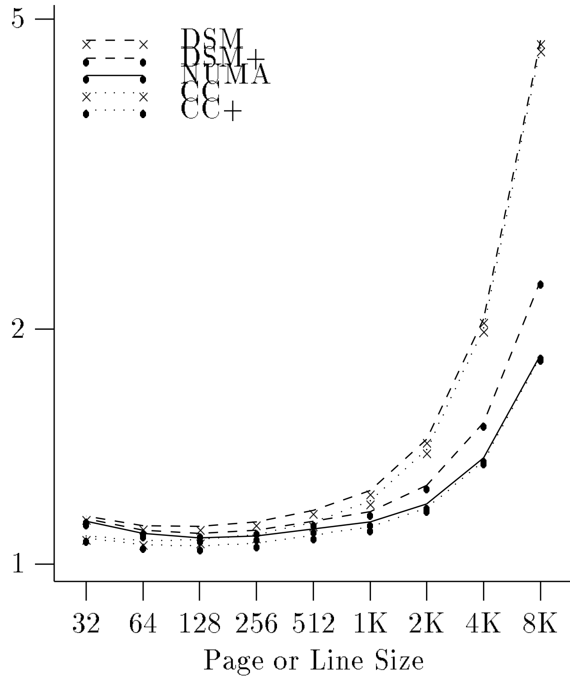

Figure 20: water

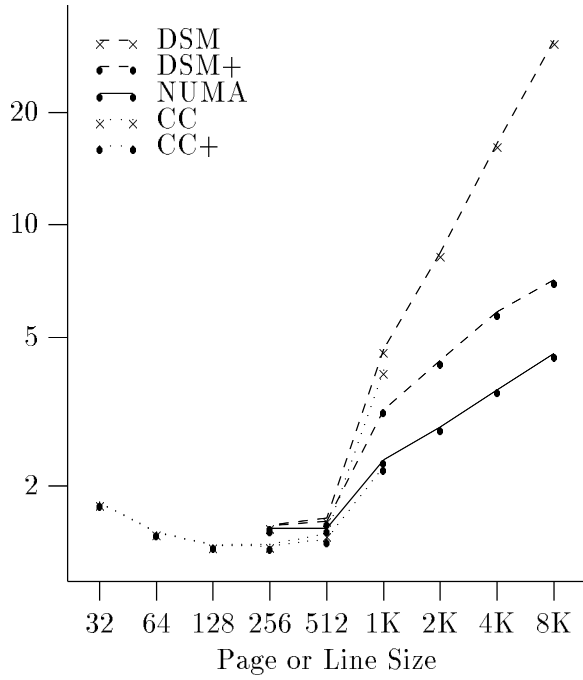

Figure 22: p-matmult

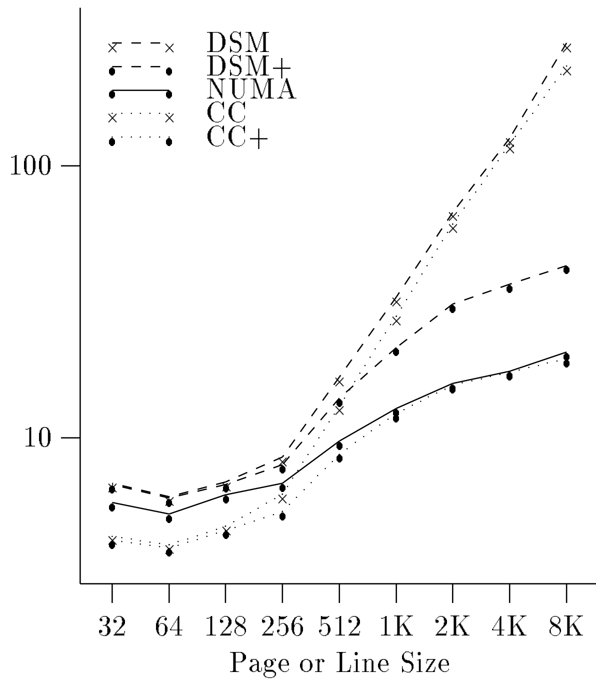

Figure 21: p-gauss

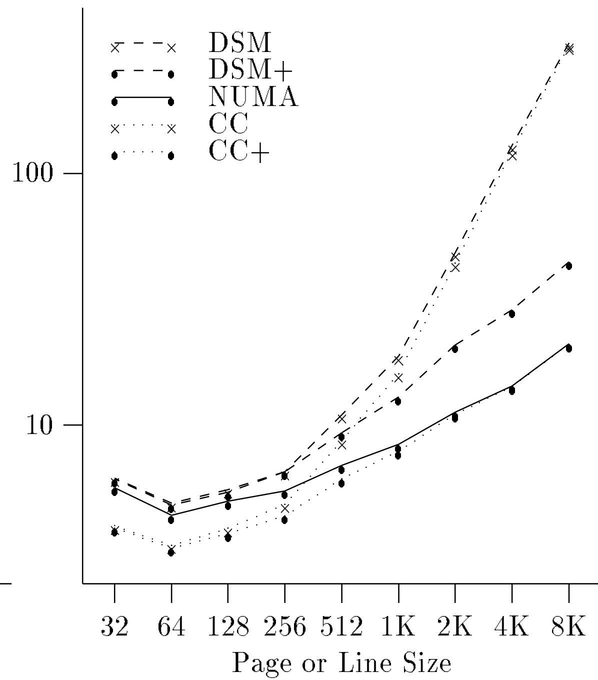

Figure 23: p-life