ASPLOS '91

NUMA Policies and Their Relation to Memory Architecture

William J. Bolosky[1,2] Michael L. Scott[1]
Robert P. Fitzgerald[2]
Robert J. Fowler[1] Alan L. Cox[1]

Abstract

Multiprocessor memory reference traces provide a wealth of information on the behavior of parallel programs. We have used this information to explore the relationship between kernel-based NUMA management policies and multiprocessor memory architecture. Our trace analysis techniques employ an off-line, optimal cost policy as a baseline against which to compare on-line policies, and as a policy-insensitive tool for evaluating architectural design alternatives. We compare the performance of our optimal policy with that of three implementable policies (two of which appear in previous work), on a variety of applications, with varying relative speeds for page moves and local, global, and remote memory references. Our results indicate that a good NUMA policy must be chosen to match its machine, and confirm that such policies can be both simple and effective. They also indicate that programs for NUMA machines must be written with care to obtain the best performance.

1 Introduction

As multiprocessors increase in size, truly uniform-speed shared memory becomes possible only by making memory uniformly slow. In other words, systems with a sufficiently large number of processors necessarily have shared memory that is physically closer to some processors than to others, in the form of caches, distributed main memory, or some combination thereof. To best take advantage of such systems, threads and the data they access must be placed near one another. In a machine with hardware coherent caches, data is moved automatically. In other machines, the kernel or other software must decide to move data explicitly. The acronym NUMA is used to refer to multiprocessors in which the non-uniformity of memory access times is explicitly visible to the programmer, and in which the placement of data is under software control. The task of the software—deciding when to move data, and where—is called the "NUMA problem."[3]

Several groups have explored kernel-based solutions to the NUMA problem on machines with distributed main memory. Solutions focus on replicating and migrating pages, generally in response to page faults. Holliday has explored migration based on periodic examination of reference bits [16], and has suggested [17] that good dynamic placement of code and data offers little additional benefit over good initial placement. In [10] and [13] we argue that simple mechanisms in the kernel work well, and probably achieve most of the benefits available without application-specific knowledge.

A major weakness of past work has been its lack of a formal framework or of solid quantitative comparisons. Black and Sleator have devised a dynamic page placement algorithm with provably optimal worst-case behavior [9], and Black, Gupta and Weber have simulated it on address traces [7], but their approach does not appear to exploit "typical" program behavior, and requires a daunting amount of hardware assistance. LaRowe and Ellis [18] have compared competing policies by implementing many alternatives in their DUnX version of the operating system for the BBN Butterfly. We adopted an alternate approach.

We use multiprocessor memory reference traces to drive simulations of NUMA policies. We implement page placement policies and simulate architectures in our trace analysis program. At the same time, we present a formal model of program execution "cost," and an efficiently-computable off-line policy that is optimal with respect to this metric. In comparison to implementation-based experiments, our approach has two principal advantages:

1. Our optimal algorithm gives us a tight lower bound on the cost savings that could be achieved by any placement policy. We can quantify the differences between policies, and assess the extent to which NUMA management contributes to overall program performance.

2. By varying architectural parameters we can investigate the extent to which policies should be tuned to the ar-

[0]This material is based upon work supported by the National Science Foundation under Grant number CDA-8822724, by ONR/DARPA under research contract no. N00014-82-K0193, by an IBM summer student internship, and by a Joint Research Agreement for Loan of Equipment (Number 14520052) between IBM and the University of Rochester.

[1]Department of Computer Science,
University of Rochester, Rochester, NY 14620
internet: bolosky, scott, fowler or cox @cs.rochester.edu
[2]IBM TJ Watson Research Center,
PO Box 704, Yorktown Heights, NY 10598
internet: fitzgerald@ibm.com

[3]It is possible, of course, to move threads as well as data, but this option is beyond the scope of the work described here.

chitecture, and can assess the utility of novel architectural features. We observe, as one might expect, that to perform well different architectures require different management policies. The optimal algorithm allows us to avoid a bias toward any particular policy in making these assessments.

In addition, the off-line nature of our analysis allows us to examine program executions at a very fine level of detail. We observe that program design, and memory usage patterns in particular, can have a profound effect on performance and on the choice of a good NUMA policy. Obtaining the best performance from a NUMA machine will require that compilation tools be aware of memory sharing patterns in the programs they are producing, that programmers work to produce "good" sharing, or, more likely, some combination of the two.

The design space of policies is large [18], but our experience suggests that simple policies can work well, and that minor modifications are likely to yield only minor variations in performance. Major changes in policy are likely to be justified only by changes in architecture. Rather than search for the perfect policy on any particular machine, we have therefore chosen to investigate the way in which architectural parameters determine the strategy that must be adopted by any reasonable policy.

Section 2 presents our methodology, defining our cost metric, presenting the optimal policy, and explaining how we collected and analyed traces. Section 3 presents the basic results of our analysis. For each application in our test suite, we show the optimal memory and page placement cost, and compare that with the cost achieved by each of the on-line policies. We compare the policies presented in our previous papers, and argue that each is appropriate for the machine for which it was designed. We suggest that a small amount of hardware support could help eliminate inappropriate page moves, and may be desirable on a machine in which such moves are comparatively expensive. Section 4 focuses on the tradeoff between block transfer time and the latency of remote memory references. We study one application in detail to identify the points at which it is able to exploit faster page moves.

2 Methodology

Our trace analysis techniques rely on a formal model of machine behavior (described informally here) that captures the essential elements of the NUMA problem while abstracting away much of the complexity of an implementation. In particular, we have chosen a model of program "cost" that is consistent with post-mortem analysis of traces, and that is amenable to optimization by an efficient off-line algorithm.

2.1 Execution and Cost Model

Our model of program execution cost approximates the total amount of processor time spent on page movement and latency of accesses to data memory. It does not attempt to model elapsed wall-clock time, nor does it model contention, either in the memory or the interprocessor interconnect, or instruction-fetch references.

We consider two types of NUMA architectures: local/remote and local/global/remote. A local/remote machine is one in which all memory is associated with a given processor, and accessing another processor's memory is possible but slower than local. The BBN Butterfly Parallel Processor [4] is a local/remote machine. A local/global/remote architecture is one in which in addition to memory at every node, there is *global* memory: intermediate in speed between local and remote, and not associated with any particular processor. The IBM ACE multiprocessor [15] is a local/global/remote machine. The IBM RP3 [21] can be considered a local/global/remote machine if its interleaved memory system is used for global memory.

A machine is characterized by the size of a page and by four memory-latency parameters: g, r, G, and R. In a local/global/remote architecture, it costs g to access a single word of global memory, G to move a page from global to a local memory (or vice-versa), r to access a single word of remote memory, and R to move a page from one local memory to another. In a local/remote architecture, g and G are not used, while r and R retain their meaning. In this paper we assume a page size of 1K 4-byte words. By definition, the cost of a full-word local memory reference is one, so the unit for the other costs is the local memory reference time.

A *trace* is a record of the data references made by a parallel program, with accesses from different processors interleaved as they occurred during tracing. A *placement* for a trace is a choice of location(s) for each data page at each point in time, with the restriction that exactly one copy of each page exists after each write to that page.[4] A *policy* is a function that maps traces to placements. We assume that individual memory accesses take a predictable amount of time, and that the total cost for a page is the sum of these individual times (1, g, or r), plus the time required for any page move operations that occur (G or R per move). Pages are assumed to be independent, so the locations chosen and costs incurred for one page have no effect on another. The *total cost of a placement* is the sum of the costs for its individual pages.

The cost of invalidations (eliminating replicas of a page) is implicit in this model. The page move costs G and R include a certain amount of overhead for the eventual invalidation of the copy being made, as well as overhead for processing the memory fault (or other event) used to trigger the copy or move operation. Pages are initially placed in global memory if it is available and otherwise in the local memory of processor 0.

Our measure of policy performance is mean cost per reference (MCPR), which is the cost of the placement chosen by the policy divided by the number of data references. If all data references were made by processor 0 on a local/remote machine, and if a policy left all pages on that processor, then the MCPR of that policy on that trace would be 1. For any trace on a local/global/remote machine, the MCPR of a policy that never moved or replicated pages would be g.

2.2 Discussion

As in any trace-based study, our results are interesting to the extent that they successfully model some important aspect of real-world behavior, despite simplifying assumptions. In our case, one major simplifying assumption is that the memory locations accessed by a program, and the order in which those accesses occur, are independent of the NUMA placement policy. This assumption is fundamental to the trace-based methodology. In practice, of course, a change

[4]This definition precludes write-update replication schemes.

in policy will alter program timings, leading to a different trace, which in turn may change the behavior of the policy, and so on. At the very least a change in policy will change the interleaving of references from different processors. One could adjust the interleaving during trace analysis, based on per-processor accumulated costs, but this approach would run the risk of introducing interleavings forbidden by synchronization constraints in the program. It would also at best be a partial solution, since the resolution of race conditions (including "legitimate" races, such as removing jobs from a shared work queue) in a non-deterministic program could lead to a different execution altogether. Forbidden interleavings could be avoided by identifying synchronization operations in a trace, and never moving references across them, but even this approach fails to address race conditions. On-the-fly trace analysis, such as performed in TRAPEDS [24], could result in better quality results, but only at a significant cost for maintaining a global notion of time (ie., possibly synchronizing on every simulated machine cycle).

The execution of a parallel program can be characterized by a set of executed instructions, linked into a partial order by control and data dependencies. This partial order captures one possible outcome of any non-determinism in the program, and a reference trace such as ours captures one total order consistent with the partial order. Establishing the interleaving of data references during trace collection guarantees that analysis is performed on a valid total order. It also results in a simple formal model. Intuitively, we expect the results of analyses based on one valid total order to be close to those of another valid order. Since conflicting writes are likely to represent synchronization operations or deliberate data dependencies, changes in interleaving consistent with program logic are unlikely to alter the behavior of the optimal algorithm.

In an attempt to validate our methodology we performed a pair of experiments to test the sensitivity of our results to changes in the order of references in our traces. In the first experiment we locally reordered references from different processors to simulate small perturbations in their relative rates of progress. In the second experiment we introduced pauses in the execution of individual processors long enough to simulate the overhead of a page move operation. The change in the cost of the optimal policy never exceeded 0.3% in either experiment on any of our traces, and was typically much smaller. The on-line policies described in section 2.5 were somewhat more sensitive, but typically changed by less than 1%—well below the level of changes considered significant for them elsewhere in this paper.

2.3 Trace Collection

The traces used in this paper were collected on an IBM ACE Multiprocessor Workstation [15] 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.

We collected traces by single-stepping each processor and decoding the instructions to be executed, to determine if they accessed data. We did not record instruction fetches. Our single-step code resides in the kernel's trap handler, resulting in better performance (and therefore longer traces) than would have been possible with the Mach exception facility [8] or the Unix **ptrace** call. Execution slowdown is typically a factor of slightly over 200. Other tracing techniques range from about an order of magnitude slower [25][5] to two orders of magnitude faster [19].

Our kernel tracer maintains a buffer of trace 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, we ran our applications in single-user mode, with no other system or user processes running.

2.4 An Efficiently-Computable Optimal Policy

Recall that a policy is a mapping from traces to page placements. An *optimal policy* is one that for any trace produces a placement whose cost is as low or lower than that of any other placement. Policies may be on-line or off-line. An on-line policy makes its placement decisions based only on references which have already happened, while an off-line policy may use future knowledge. It is easy to show that any optimal policy must be off-line; all the other policies described in this paper are on-line.

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. The essential insight in the algorithm is that after each write a page 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 page may be replicated during the string of reads; whether this replication occurs depends on the starting and ending locations of the page and the number of reads made by each processor during the interval. Replication and migration are not entirely independent: if processor A reads a page several times between writes y and $y + 1$, but not often enough for the reads themselves to warrant replication of the page, it may be cheaper in the long run to place the page at location A at write y, even if location B is cheaper at that time.

The placement generated by an optimal policy is interesting because, to the extent that the cost metric is realistic, it demonstrates the best possible use of available hardware. It provides a tighter lower bound against which to compare the performance of implementable policies than the more obvious "MCPR = 1." More important, it provides a policy-insensitive tool for evaluating architectural tradeoffs. We can use it, for example, to determine whether remote access is useful in a local/global/remote machine, or to estimate how aggressive a policy should be in migrating pages on a machine with a fast block transfer, without being biased by a policy that favors one type of architecture or another.

2.5 Implementable (Non-Optimal) Policies

In addition to the optimal policy, we consider three implementable alternatives. Two of them have been used in real systems and are described in prior papers: the ACE policy [10] and the PLATINUM policy [13]. The third policy, Delay, is based on the ACE policy, and exploits simple hypothetical hardware to reduce the number of pages moved or "frozen" incorrectly.

The ACE policy can be characterized as a dynamic technique to discover a good static placement. The expectation

[5]They report "50Mbytes" of trace; we assume that they are using 4 bytes/trace entry.

is that the chosen placement will usually be nearly as good as a user-specified placement, and often better, and will be found with minimal overhead. The ACE policy was designed for a machine that has fast global memory ($g = 2$) and no mechanism to move a page faster than a simple copy loop ($G = 2 * pagesize + 200$ (200 is fault overhead)). It operates as follows: Pages begin in global memory. When possible, they are replicated to each processor reading them. If a page is written by a processor that has no local copy, or if multiple copies exist, then a local copy is made and all others are invalidated. After a fixed, small number of invalidations, the page is permanently frozen in global memory. We permit four invalidations per page in the studies in this paper.

The PLATINUM policy was designed for a machine with no global memory, slow remote memory ($r = 15$), and a comparatively fast block transfer ($R = 3 * pagesize + 200$). Its principal difference from the ACE policy is that it continues to attempt to adapt to changing reference patterns by periodically reconsidering its placement decisions. PLATINUM replicates and moves pages as the ACE algorithm does, using an extension of a directory-based coherent cache protocol with selective invalidation [11]. The extension freezes a page at its current location when it has been invalidated by one processor and then referenced by another within a certain amount of time, t_1. Once every t_2 units of time, a daemon defrosts all previously frozen pages. On the Butterfly, t_1 and t_2 were chosen to be $10ms$ and $1s$ respectively. Since time is not explicitly represented in our simulations, t_1 and t_2 are represented in terms of numbers of references processed. The specific values are obtained from the mean memory reference rate on an application-by-application basis, by dividing the number of references into the (wall clock) run time of the program and multiplying by $10ms$ and $1s$ respectively. The PLATINUM algorithm was designed to use remote rather than global memory, but could use global memory to hold its frozen pages.

Because they are driven by page faults, the ACE and PLATINUM policies must decide whether to move or freeze a page at the time of its first (recent) reference from a new location. Traces allow us to study the pattern of subsequent references, and confirm that the number of references following a page fault sometimes fails to justify the page move or freeze decision. Bad decisions are common in some traces, and can be quite expensive. An incorrect page move is costly on a machine (like the ACE) that lacks a fast block transfer. An incorrect page freeze is likewise costly under the ACE policy, because pages are never defrosted. Motivated by these observations, we postulate a simple hardware mechanism that would allow us to accumulate some reasonable number of (recent) references from a new location before making a placement decision.

The mechanism we suggest is a counter implemented in the TLB that is decremented on each access, and that produces a fault when it reaches zero. When first accessed from a new location, a page would be mapped remotely, and its counter initialized to n. A page placement decision would be made only in the case of a subsequent zero-counter fault. This counter is similar to the one proposed by Black and Sleator [9] for handling read-only pages, but unlike their proposal for handling writable pages, it never needs to be inspected or modified remotely, and requires only a few bits per page table entry. We set $n = 100$ for the simulations described in this paper. Our observations are that a delay of 100 is more than is normally needed, but the marginal cost of a few remote references as compared to the benefit of preventing unnecessary moves seems to justify it.

3 Experiments

Our analysis of traces attempts to answer the following kinds of questions within the formal framework of our model:

- To what extent can one hope to improve the performance of multiprocessor applications with kernel-based NUMA management—is the NUMA problem important?
- How closely do simple, easily implemented policies approach the performance limit of the optimal off-line policy?
- How does the choice of application programming system and style affect the effectiveness of each of the policies?
- To what extent does the effectiveness of policies vary with changes in memory architecture? Can we characterize the "strategy" used by the optimal policy as a function of these parameters?

We begin our discussion of experiments with a brief description of our application suite. We then turn to the questions above in sections 3.2.1 through 3.2.4, deferring full consideration of the final question to section 4.

3.1 The Application Suite

We traced a total of sixteen applications, written under three different programming systems. Each of the three systems encourages a distinctive programming style. Each is characterized by its memory access patterns and granularity and by its style of thread management. 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."

EPEX [23] is an extension to FORTRAN developed for parallel programming at IBM. EPEX applications typically are numeric. The programmer explicitly identifies the private and shared data in source code and as a result the amount of shared data can be relatively small [3]. Parallelism arises from the distribution of DO loops to the set of available processors. The EPEX applications traced were `e-fft`, a fast Fourier transform; `e-simp`, a version of the Simple benchmark [14]; `e-hyd`, a hydrodynamics code; and `e-nasap`, a program for computing air flow. The prefix `e-` indicates an EPEX application.

Mach C-Threads [12] is a multi-threaded extension to C. Our C-Threads programs were either written for the ACE, or for PLATINUM and ported to the ACE. In either case, they were written with a NUMA architecture in mind, and employ a programming style that can be characterized as coarse-grain data parallelism: a single thread of control is assigned statically to each available processor and data is partitioned evenly among them. All data is potentially shared, and the pattern of access is not identified in the program.

The C-Threads programs traced were `bsort`, a simple merge sort program in which half of the processors drop out in each phase; `kmerge`, a merge sort program in which

Application	References	Private Refs
`e-fft`	10.1	81.1
`e-simp`	27.8	109
`e-hyd`	49.8	445
`e-nasap`	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
`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)

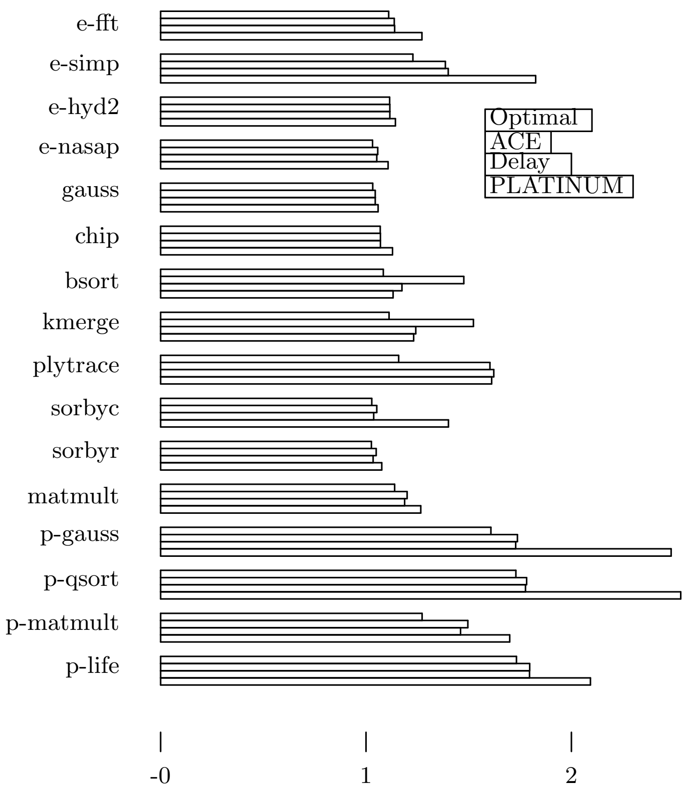

Figure 1: MCPR for ACE Hardware Parameters

groups of processors cooperate in each merge step, thus keeping all processors busy to the end of the computation [2]; `matmult`, a straightforward matrix multiplier; `plytrace`, a scene rendering program; `sorbyr` and `sorbyc`, a pair of red-black successive over-relaxation [20] programs that differ in the order of their inner loops, and thus in their memory access patterns; and `chip`, a simulated annealing program for chip placement.

In many of the C-Threads applications two-dimensional numerical matrices are represented as an array of pointers to single-dimensional arrays representing the rows, as recommended in *Numerical Recipes in C* [22]. In these programs the unit of data-sharing is the row, so data sharing patterns exhibit a fairly coarse grain.

Presto [6] is a parallel programming system based on C++. Because Presto was originally implemented on a Sequent Symmetry, a coherent cache machine, its applications were written without consideration of NUMA memory issues. The Presto programs we traced are characterized by fine-grain data sharing and by a programming style that allocates a large number of threads of control, regardless of the number of physical processors actually available. Presto was ported to the ACE and the applications were run unmodified. The applications traced were: `p-qsort`, a parallel quicksort; `p-gauss`, a Gaussian elimination program; `p-matmult`, a matrix multiplier; and `p-life`, an implementation of Conway's cellular automata. The behavior of these programs was studied in a different context in [5].

3.2 Performance of the Various Policies

The performance of each of our policies on each of our applications, expressed as Mean Cost Per Reference (MCPR), appears in Figures 1 and 2–3, for architectures resembling the ACE and the Butterfly, respectively. Each application has a group of four bars, which represent the performance of Optimal, ACE, Delay and PLATINUM, from top to bottom. To place the sizes of the bars in context, recall that an MCPR of 1 would result if every memory reference were local. For ACE hardware parameters, an MCPR of 2 is trivially achievable by placing all shared data in global memory: any policy that does worse than this is wasting time on page moves or remote references

3.2.1 The importance of the NUMA problem

For the NUMA problem to be of importance, several things must be true: memory access time must be a significant fraction of the execution time of a program; the performance difference between executions with correctly and incorrectly placed pages must be large; there must be some reasonably good solution to the problem. In [10] we estimate the memory times for programs running on an ACE to be in the 25%–60% range. Newer, more aggressive processor architectures will only increase this percentage, as demonstrated by the increasing emphasis on cached memory systems.

One possibility for NUMA management is to statically place all private data and to leave shared data in global memory or in an arbitrary local memory. This strategy will work well for applications such as `e-fft`, which have only private and fine-grained shared data, but it will not work well for others. Many programs require data to migrate, particularly when remote references are costly. Examples include matrix rows lying at the boundaries between processor bands in `sorbyr`, and dynamically-allocated scene information in `plytrace`. This is demonstrated by the number of page moves performed by the optimal policy, presented in Figure 6. It explains why the PLATINUM policy (which is more aggressive about moving pages) generally does better than the ACE or Delay policies on a machine such as the Butterfly, in which a page move can be justified to avoid a

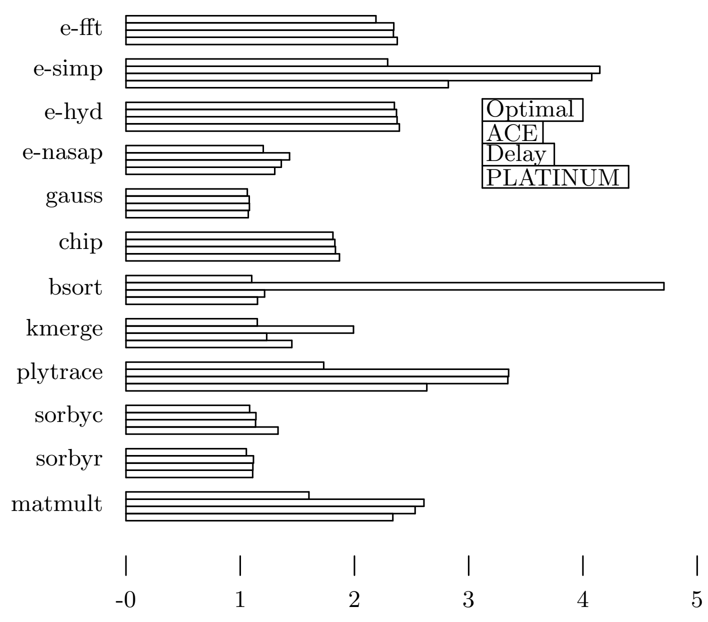

Figure 2: MCPR for Butterfly Hardware Parameters

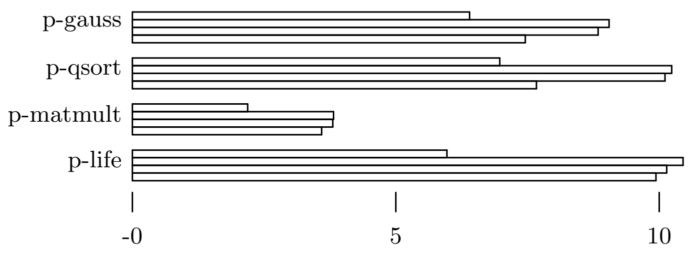

Figure 3: MCPR for Butterfly Hardware Parameters, PRESTO applications

relatively small number of remote references.

Even on a machine like the ACE, in which frozen pages are only twice as expensive to access as local pages, there is a large benefit in correctly placing pages. For all but the Presto applications, an optimal placement results in an MCPR below 1.23 on the ACE (as compared to 2 for static global placement) and 2.35 on the Butterfly (as compared to 14–15 for random placement). For a program that spends 50% of its time accessing data memory, these MCPR values translate to a 26% improvement in running time on the ACE, and a 56% improvement on the Butterfly, in comparison to naive placement, assuming no contention. As shown in the following section, our implementable policies achieve a substantial portion of this savings.

3.2.2 The success of simple policies

Both the ACE and Delay policies do well on the ACE. The MCPR for Delay is within 15% of optimal on all applications other than `plytrace`. The ACE policy similarly performs well for applications other than `plytrace`, `bsort` and `kmerge`. These programs all display modest performance improvements when some of their pages migrate periodically, and the ACE and Delay policies severely limit the extent to which this migration takes place.

All of the policies keep the MCPR below 4 for the non-Presto applications on the Butterfly, with the exception of ACE on `bsort`, and that case could be corrected by increasing the number of invalidations allowed before freezing. For all applications other than `plytrace`, PLATINUM stays near or below 2.5. This is quite good, considering that a random static placement would yield a number close to 15. The ACE and Delay policies perform slightly better than PLATINUM on applications that have only fine grained shared and private data (`e-hyd` and `e-fft`), but the cost of moving pages that should be frozen is low enough on the Butterfly that the difference between the policies in these cases is small.

The difference between the ACE and Delay policies displays a bimodal distribution. In most cases the difference is small, but in a few cases (`bsort` and `kmerge`) the difference is quite large. In essence, the additional hardware required by Delay serves to prevent mistakes.

3.2.3 The importance of programming style

The Presto applications have much higher MCPRs for both architectures, in both the on-line and optimal policies. This disappointing performance reflects the fact that these programs were not designed to work well on a NUMA. They have private memory but do not make much use of it, and their shared memory shows little processor locality. The shared pages in the EPEX `e-fft` and `e-hyd` programs similarly show little processor locality, but because these programs make more use of private memory, they still perform quite well.

The programs that were written with NUMA architectures in mind do much better. Compared to the Presto programs they increase the processor locality of memory usage, are careful about which objects are co-located on pages with which other objects, and limit the number of threads to the number of processors available. It is not yet clear what fraction of problems can be coded in a “NUMAticized” style.

3.2.4 The impact of memory architecture

From the discussions above it is clear that the difference in architecture between the ACE and Butterfly machines mandates a difference in NUMA policy. It pays to be aggressive about page moves on the Butterfly. Aggressiveness buys a lot for applications such as `plytrace` and `e-simp`, which need to move some pages dynamically, and doesn't cost much for applications such as `e-fft`, which do not. At the same time, aggressiveness is a bad idea on the ACE, as witnessed by the poor performance of the PLATINUM policy on many applications (`sorbyc, e-simp, matmult, e-fft, p-gauss`). In as much as the ACE and Butterfly represent only two points in a large space of possible NUMA machines, it seems wise to explore the tradeoffs between architecture and policy in more detail. This exploration forms the subject of the following section.

4 Variation of Architectural Parameters

Figures 4 and 5 show how the performance of the optimal policy varies with the cost of a page move (G or R), for remote and global access times comparable to those of the ACE and the Butterfly, respectively. Results for the Presto applications are not shown, because they are off the scale of

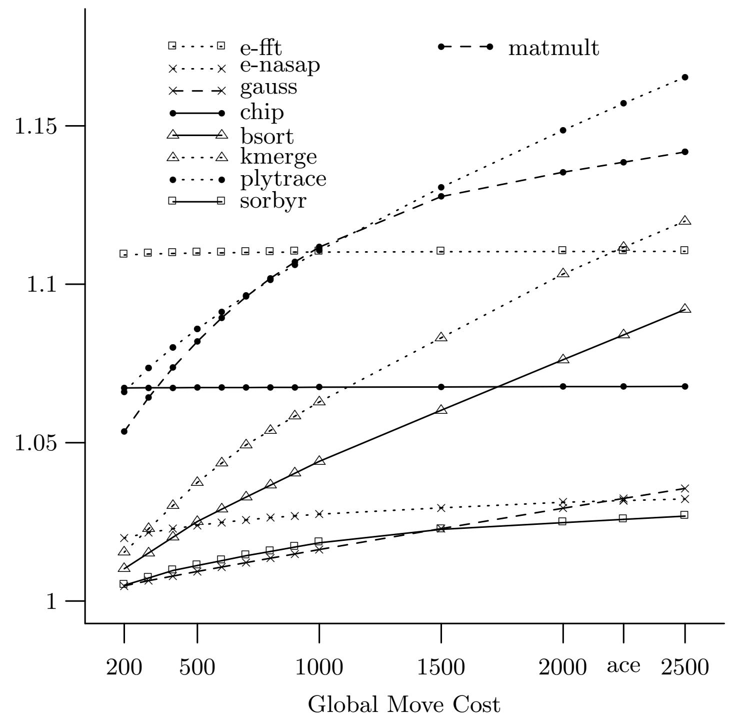

Figure 4: MCPR vs. G for optimal, g=2, r=5

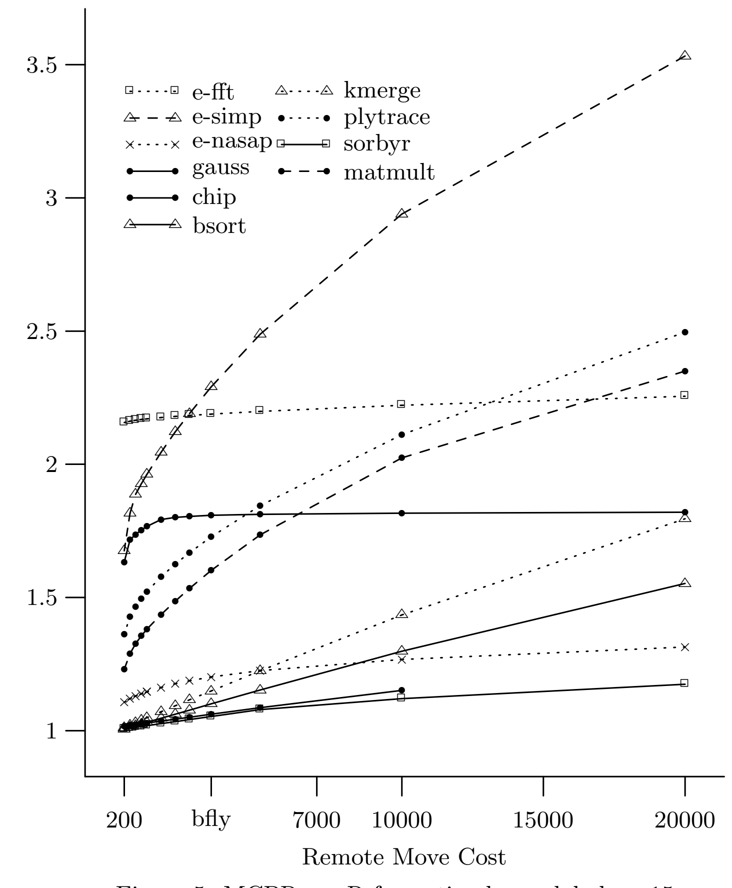

Figure 5: MCPR vs. R for optimal, no global, r=15

the graphs; however, their shape is not significantly different from the other applications.

The minimum page move time represented on each graph is 200, which is assumed to be a lower bound on the time required to process a fault and initiate a page move in the kernel. 200 therefore corresponds to an infinite bandwidth, zero latency hardware block transfer. The maximum page move times on the graphs are the page size times g or r, plus a more generous amount of overhead, corresponding to a less tightly coded kernel.

If R is considered to be a real-valued variable, then the cost of the optimal policy on a trace is a continuous, piecewise linear function of R. Furthermore, its slope is the number of page moves it makes, which in turn is a monotonically decreasing step function in R. Similar functions exist for G, g, and r, except that their slopes represent global page moves, global references, and remote references respectively. An important implication of continuity is that, given optimal placement, there is no point at which a small improvement in the speed of the memory architecture produces a disproportionately large jump in performance.

At a G or R of 0, page moves would be free. The optimal strategy would move all pages on any non-local reference. This means that for a G or R of 0 the optimal MCPR of any application must be 1, regardless of the values of g and r. Since the optimal cost is continuous, the curve for every application must fall off as G or R approaches 0. This means that all the curves in Figures 4 and 5 go smoothly to 1 below their left ends. For applications such as `e-fft` that don't show much benefit from G and R down to 200, this drop is very steep.

Though different machines require different policies, any given policy[6] will be oblivious to the speed of memory operations. The curve for a given policy will therefore be a straight line on a graph like Figure 4, and will lie on or above the optimal curve at all points (see Figure 7 for an example). Because the optimal curve is concave down, no straight line can follow it closely across its entire range. This means that no single real policy will perform well over the whole range of architectures. Thus, to obtain best performance over a range of page move speeds in Figures 4 and 5 in which the optimal line curves, one must change the real policies used accordingly. However, for the applications whose curves are largely flat lines, the same policy works over the entire range.

One can plot MCPR, g (or r), and G (or R) on orthogonal axes to obtain multi-dimensional surfaces. Figures 4 and 5 show two-dimensional cuts through these surfaces. They are interesting cuts in the sense that one can imagine spending extra money on a machine to increase the speed of block transfer relative to fixed memory reference costs. It makes less sense to talk about varying the memory reference costs while keeping the block transfer speed fixed. Moreover, *figures 4 and 5 capture all of the structure of the surfaces*, at least in terms of the relationship between page move cost and memory reference cost. We have proved that the behavior of the optimal algorithm on a given trace is completely determined by the ratios of $g-1$, $r-1$, G, and R. Scaling them by the same multiplicative factor s changes the MCPR m according to $m_{new} = 1 + s(m_{old} - 1)$, but does not change the optimal placement. Each multi-dimensional surface consists of rays pointing up and out from

[6]Here, "policy" is used in the strictest sense of the word. Changing t_1 or t_2 in PLATINUM would thus yield a new policy for our purposes.

$(g, r, G, R, m) = (1, 1, 0, 0, 1)$. If we slide figure 5 in toward the r origin, the curves retain their shape, but shrink in the R and MCPR dimensions.

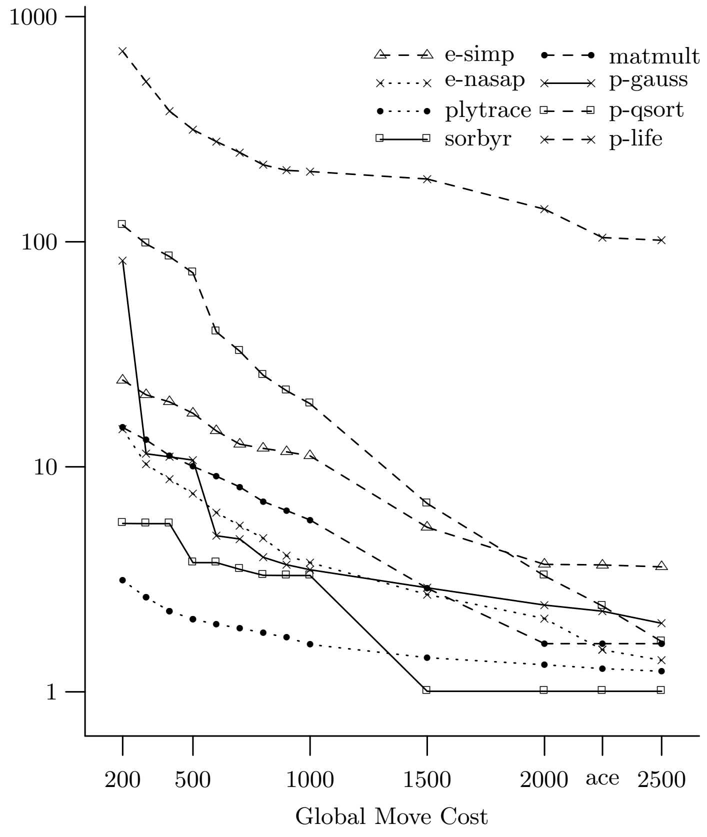

Figure 6: Mean Page Moves per Page for optimal, g=2, r=5

Figure 6 presents, on a logarithmic scale, the mean number of page moves per page as a function of G for an ACE-like machine. Many of the applications have large jumps in the number of moves made around 1024 and 512. These are points at which referencing each word on a page, or half of the words, is sufficient to justify a page move. Some applications show large jumps at other multiples or fractions of the page size, but large changes at other values of the page move cost are rare.

When designing a NUMA policy for a given machine, one should take into account where on our move cost spectrum the architecture lies. Machines to the left of jumps benefit from more aggressive policies, machines to the right from more conservative policies. A machine that lies near a jump point will run well with policies of varying aggressiveness. When designing a NUMA machine, the lessons are less clear. Obviously, faster machines run faster. Also, the marginal benefit of a small speedup increases at faster speeds. However, moving across a jump point will not produce a corresponding jump in performance: the jump is in the *slope* of the cost curve, not in the cost curve itself.

4.1 Case Study: Successive Over-Relaxation

To illustrate what is happening to the optimal placement as we vary page move speed, we examined one of the successive over-relaxation (SOR) applications, `sorbyr`, in some depth. `Sorbyr` is an algorithm for computing the steady-state temperature of the interior points of a rectangular object given the temperature of the edge points. It represents the object with a two-dimensional array, and lets each processor compute values in a contiguous band of rows. Most pages are therefore used by only one processor. The shared pages are used alternately by two processors; one processor only reads the page, while the other makes both reads and writes, for a total of four times as many references.

Almost all of `sorbyr`'s references are to memory that is used by only one processor. Thus, the MCPR values are all close to 1. However, this case study concentrates on the portion of references that are to memory that is shared, and the effects of management of this memory are still clearly visible in the results presented, and is fairly typical of shared memory in other NUMA applications.

The optimal placement behavior for a shared page depends on the relative costs of page moves to local, global and remote references. This behavior is illustrated in Figure 7 as a function of page move cost. In this graph the cost of the optimal policy is broken down into components for page moves, remote references, global references and local references. Since most pages are used by only one processor, the major cost component is local references; in this figure, however, the local section is clipped for readability.

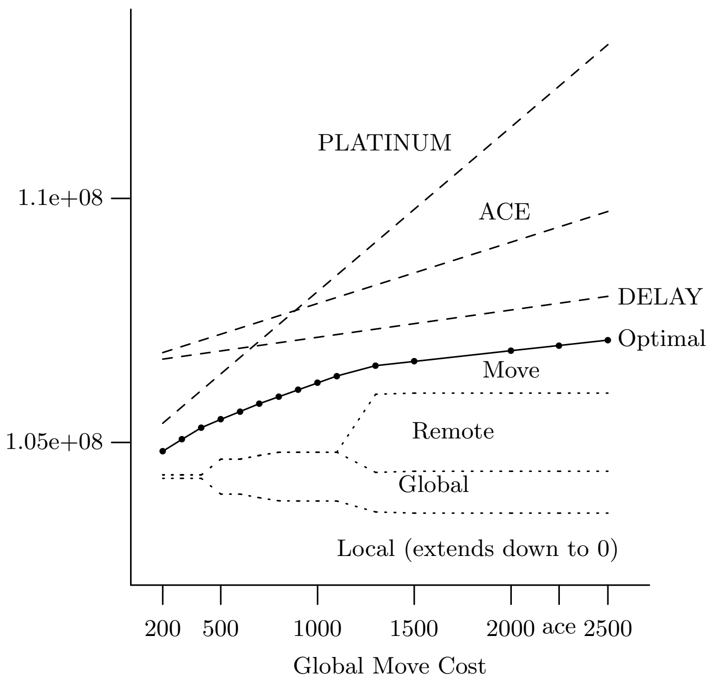

Figure 7: `sorbyr` Placement Cost vs. Page Move Cost w/ Optimal Breakdown, g=2, r=5

As page move cost decreases, remote references are traded for copies and global references, and then for more copies and local references. This can be seen in Figure 7 at points near $G = 1200$ and $G = 400$ respectively. It is important to note that while the cost breakdown of the optimal policy undergoes large sudden changes, the cost itself as a function of G is continuous.

The performance of the other policies is also included.

The PLATINUM policy works best for small G. This is expected, since it is designed for a machine with a relatively fast page move. However, since it must be above optimal at all points and is a straight line (*i.e.*, is architecture insensitive), it must be bad for large G in order to be good for small G. Conversely, for the ACE or Delay policies to do well for large G they must not do as well for small G.

4.2 Global Memory

Our results suggest that global memory is useful in NUMA machines. It is the cheapest place to put data that are shared on a sufficiently fine grain to preclude migration, and that are not accessed predominately by a single processor. With global memory available remote references would be reserved for data such the rows at the boundaries between processor bands in `sorbyc`, which are accessed by one processor significantly more than all the rest combined, and would not benefit from migration unless it was extremely fast.

In our application suite on an ACE-like machine ($g = 2$, $r = 5$), the largest use of remote references by the optimal strategy occurred in `sorbyr`, where even with an extremely slow page move ($G = 5000$) it made only 320K references remotely out of a total of 104M, or 0.3%. This does not imply that remote references are useless in the presence of global memory, but only that they need not be used often, and consequently increasing their cost will have only minor effects on overall program run time.

5 Summary and Conclusions

Our work has addressed issues in the design of kernel-based NUMA management policies. Our approach has been to use multiprocessor memory reference traces, from a variety of applications, to drive simulations of alternative policies under a range of architectural parameters. In the area of NUMA policy design, we have found that:

- The problem is important. In comparison to naive placement of shared data, optimal placement can improve overall program performance by as much as 25 to 50%.

- Good performance on NUMA machines depends critically on appropriate program design. Trace analysis supports the intuition that NUMA policies will achieve the best performance for applications that minimize fine-grain sharing and the *false sharing* that occurs when data items accessed by disjoint sets of processors are inadvertently placed on a common page.

- Given good program design, simple kernel-based NUMA policies can provide close to optimal performance. Averaged over all 16 applications on the ACE, the ACE policy achieved 82% of the savings achieved by the optimal algorithm over static global placement of shared data. Averaged over all applications on the Butterfly, the PLATINUM policy achieved 94% of the savings achieved by the optimal algorithm over random placement of shared data.

- Different memory architectures require different policies for high-quality NUMA management. Dynamic discovery of a good static placement works well on a machine in which page movement is expensive in comparison to the cost of remote (or global) access. As the cost of page movement decreases, it becomes increasingly profitable to move pages between program phases. The PLATINUM policy achieved only 40% of the optimal improvement on average on the ACE. The ACE policy achieved only 87% of the optimal improvement on average on the Butterfly.

In terms of architectural design, we have found that:

- Global memory provides an attractive location for data that are shared by several processors at a fine grain. It is cheaper for all processors to access intermediate-cost memory than for all processors but one to access costly memory.

- A policy that is (wisely) unaggressive about dynamic page placement due to high page move cost could use a mechanism such as our proposed per-page reference counter to significantly reduce the number of page-placement errors, thereby improving performance. On average on the ACE, the Delay algorithm achieved an additional 4.6% of the improvement of the optimal algorithm. On the Butterfly, it prevented two disastrous mistakes.

- Improving block transfer speed by some multiplicative factor f can lead to an improvement of more than f in memory cost, because a good policy is likely to move more pages when doing so is cheap. At the same time, there are no points at which a small improvement in block transfer speed produces a large improvement in memory cost. Fast block transfer will be most effective on machines in which remote memory references are comparatively expensive, and then only for certain applications; it is unlikely to be cost-effective on a machine with cheap remote memory.

The NUMA problem is likely to become increasingly important as improvements in CPU performance outstrip improvements in memory performance, forcing future processors to spend a larger fraction of their time waiting for memory. It is possible that scalable cache-coherent architectures will supplant machines with visibly-distributed shared memory, but many of the issues central to NUMA management will remain. To first approximation, hardware cache coherence can be captured by our model in the form of unusually small "pages" and very fast page moves. Contention may need to be addressed more carefully in the simulation of cache-based machines, but even in its current form our trace analysis system can provide valuable insights into such issues as the incidence of false sharing, the importance of program phase changes, the degree of cache-line replication, and the frequency of invalidations.

There appear to be two possible avenues for further improvements in the quality of NUMA management: better policies and better programs. There exist applications (`plytrace`, for example) in which the optimal policy performs significantly better than any of its on-line competitors. One might be able to narrow the gap with a smarter kernel or, more likely, with application-specific hints from the compiler, run-time system, or programmer. Hints might suggest times and places for page moves. They might also identify periods of false sharing during which replicas of a page should be allowed to grow inconsistent, for example in `sorbyc`. Our experience strongly suggests, however, that the largest improvements in NUMA management will come from changes in program behavior to increase memory reference locality.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. Summer 1986 USENIX*, July 1986.

[2] R. J. Anderson. An Experimental Study of Parallel Merge Sort. Technical Report 88-05-01, Univ. of Washington Dept. of Comp. Sci., May 1988.

[3] S. J. Baylor and B. D. Rathi. An Evaluation of Memory Reference Behavior of Engineering/Scientific Applications in Parallel Systems. Tech Report 14287, IBM, June 1989.

[4] BBN. *Inside the Butterfly–Plus.* BBN Advanced Computers, Cambridge, MA, October 1987.

[5] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Arichtectures. In *Proc. 17th Intl. Symp. on Comp. Arch.*, pages 125–134, 1990.

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

[7] David Black, Anoop Gupta, and Wolf-Dietrich Weber. Competitive Management of Distributed Shared Memory. In *Proc. Spring Compcon*, pages 184–190, San Francisco, CA, February 1989.

[8] David L. Black, David B. Golub, Karl Hauth, Avadis Tevanian, and Richard Sanzi. The Mach Exception Handling Facility. In *Proc., SIGPLAN/SIGOPS Workshop on Par. and Dist. Debugging*, pages 45–56, May 1988. SIGPLAN Notices 24(1),1/89.

[9] David L. Black and Daniel D. Sleator. Competitive Algorithms for Replication and Migration Problems. Technical report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA 15213, November 1989. CMU-CS-89-201.

[10] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proc. 12th ACM Symp. on Operating Systems Principles*, pages 19–31, December 1989.

[11] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Trans. on Computers*, 27(12):1112–1118, December 1978.

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

[13] Alan L. Cox and Robert J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proc. 12th ACM Symp. on Operating Systems Principles*, pages 32–44, December 1989.

[14] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical report, Lawrence Livermore Laboratory, 1978. UCID-17715.

[15] 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.

[16] M. A. Holliday. Reference History, Page Size, and Migration Daemons in Local/Remote Architectures. In *ASPLOS III*, April 1989.

[17] Mark A. Holliday. On the Effectiveness of Dynamic Page Placement. Technical report, Department of Computer Science, Duke University, Durham, NC 27706, September 1989. CS-1989-19.

[18] R. P. LaRowe and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. Technical report, Department of Computer Science, Duke University, April 1990. CS-1990-10.

[19] James R. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. To appear, Software: Practice and Experience.

[20] James M. Ortega and Robert G. Voigt. Solution of Partial Differential Equations on Vector and Parallel Computers. *SIAM Review*, 27(2):149–240, June 1985.

[21] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proc. 1985 Intl. Conf. on Parallel Processing*, pages 764–771, 1985.

[22] W. A. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, U.K., 1988.

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

[24] Craig B. Stunkel and W. Kent Fuchs. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. In *Performance Evaluation Review, 17(1)*, pages 70–78, May 1989.

[25] W. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *ASPLOS III*, April 1989.