Evaluation of Multiprocessor Memory Systems Using Off-Line Optimal Behavior*

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

September 1991

*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.

Abstract

In recent years, much effort has been devoted to analyzing the performance of distributed memory systems for multiprocessors. Such systems usually consist of a set of memories or caches, some device such as a bus or switch to connect the memories and processors, and a policy for determining when to put which addressable objects in which memories. In attempting to evaluate such systems, it has generally proven difficult to separate the performance implications of the hardware architecture from those of the policy that controls the hardware (whether implemented in software or hardware). In this paper we describe the use of off-line optimal analysis to achieve this separation. Using a trace-driven dynamic programming algorithm, we compute the policy decisions that would maximize overall memory system performance for a given program execution. The result allows us to eliminate the artifacts of any arbitrarily chosen policy when evaluating hardware performance, and provides a baseline against which to compare the performance of particular, realizable, policies. We illustrate this technique in the context of software-controlled page migration and replication, and argue for its applicability to other forms of multiprocessor memory management.

1 Introduction

In the study of multiprocessor memory system design, as in many fields of scientific endeavor, many factors interact in complex ways to make up the behavior of the system as a whole. Overall system performance can be evaluated in many ways, including simulation, analytical modeling, and real implementation, but it is not always easy to isolate the performance effects of individual components as they contribute to the whole. Suppose we are implementing a coherent, global view of memory. If we double the bandwidth of the interconnection network, but performance improves by 5%, should we be disappointed? Is the application unable to make use of the extra bandwidth, or is our coherence protocol simply inappropriate for the revised machine? Similarly, if we introduce a lock-down mechanism to reduce ping-ponging of large cache lines with fine-grain sharing, and performance improves by 5%, should we be pleased? Have we done as well as could reasonably be expected, or could some other policy have improved performance by 50%?

In an attempt to answer such questions, we have combined trace-driven simulation with an optimal, off-line memory management policy. We begin with a formal model of application, hardware, and policy performance. We then perform a post-mortem analysis of the application, using a hardware description and a memory reference trace, to generate the least cost possible policy decisions for that application running on that hardware. In comparison to system evaluations that embed a "real-life" policy:

1. Off-line optimal analysis allows us to evaluate hardware design decisions without biasing the results based on the choice of policy (none of which is likely to be optimal over the entire design space).

2. Performance results obtained with off-line analysis provide a tight lower bound, for a given hardware base, on the cost savings that can be achieved by any "real-life" policy. Rather than tell us how much time is being used by a given policy, they tell us how much time must be used by *any* policy. The difference between optimal performance and actual achieved performance is the maximum time that could possibly be saved through policy improvements.[1]

[1] This second point is essentially the rationale behind Belady's MIN algorithm for page-out in demand-paged virtual memory systems. The performance of realizable page-replacement policies can be placed in context by comparing them to Belady's MIN.

It is generally accepted that memory reference traces need to run into millions or even hundreds of millions of references to capture meaningful behavior. Any algorithm to compute an optimal set of memory management decisions must therefore make a very small number of passes over the trace—preferably only one. Our strategy has been to use dynamic programming to keep track of the cost, to that point, of each distinguishable state of the system, and to use architecture-specific knowledge to keep the number of such states within reason. We provide a concrete example that captures the most important characteristics of NUMA multiprocessors (those with visibly non-uniform memory access times), and that can be applied as well to machines with hardware cache coherence.[2]

A NUMA multiprocessor is a distributed memory machine in which remote single-word references are permitted, and in which data placement decisions such as replication and migration are not performed in hardware. A typical NUMA system consists of some collection of processors, memories, (non-coherent) caches, interconnections between these memories (switches or shared busses) and some software policies (in the kernel, run-time libraries, or application) that decide where to locate data dynamically within the memory system. When designing NUMA hardware, an optimal NUMA placement policy allows one to evaluate architectural alternatives (e.g. different page sizes, or different block transfer speeds) without worrying about whether a particular NUMA placement policy is making equally good use of each alternative. The types of actions taken by the optimal policy when running on the hardware model that is eventually chosen can then be used to guide the design of the real, on-line policy that will eventually be implemented. The performance of an optimal policy can be used as a measure against which potential real policies can be compared.

We present a formal model of dynamic multiprocessor data placement in section 2, and a tractable algorithm for computing an optimal placement in section 3. Section 4 presents experimental results, demonstrating the use of the optimal policy to answer several questions about NUMA memory management. We include a discussion of techniques used to establish confidence in our results despite inherent weaknesses in the formal model. The paper concludes with section 5, a general discussion of domains in which off-line optimal analysis is useful, and a description of our plans for continued work in multiprocessor memory management.

[2]This latter class is generally said to consist of UMA (uniform memory access) machines. Different memory locations have different access times, but the hardware does its best to conceal this fact.

2 A Model Of Memory System Behavior

This section describes a model designed to capture the latency-induced cost of memory access and data placement in a multiprocessor memory system that embodies a tradeoff between replication, migration and single-word reference. This model describes most NUMA machines and many coherently cached machines as well.[3] It does not attempt to capture any notion of elapsed wall-clock time, nor does it consider contention, either in the memory or in the interprocessor interconnection. Memories and caches are assumed to be as large as needed. Instructions are assumed to be local at all times; instruction fetches are ignored.

The basic concepts of the model are a machine, a trace, a placement, a policy, and a cost function.

2.1 Machines

A machine μ is defined by a set of processors and memories and some parameters that represent the speed of the various memory operations. The set of processors is denoted Π, the set of memories $M = \Pi$ or $M = \Pi \cup \{\textbf{global}\}$, and the parameters $r > 1$, $g > 1$, $R > 2r$, and $G > 2g$. Each parameter is measured in units of time equal to that of a single-word local memory reference. Lower-case r is the amount of time that it takes a processor to access a word from another processor's memory.[4] Capital R is the amount of time that it takes for a processor to copy an entire block from another processor's memory. If a machine has global memory (that is memory that is not associated with any particular processor, but rather is equidistant from all processors—this could be main memory in a cached machine or "dance hall" memory in a NUMA) then the amount of time to access a word in global memory is g, while G is the cost of moving an entire block from global memory to a local memory. The model requires that if g and G are not infinite, then $r \geq g$ and $G \leq R \leq 2G$. Otherwise, if $r < g$ then it would never make sense to use the global memory; if $R > 2G$ then one could make a copy from a remote memory by copying first to global and then from

[3] Throughout this section we use the word "block" for the unit of memory that can be moved from one location to another; this formalism applies equally well to pages and cache lines; "block" is meant to represent either, depending on context.

[4] "Another processor's memory" could be main memory associated with a particular processor in a NUMA system, or a cache line in a coherently cached machine or non-coherently cached NUMA; r may be infinite if no direct remote access is permitted.

there to the destination. The symbol $\sharp$ denotes the number of elements in a finite set. To eliminate trivial cases, there be more than one processor, i.e. $p = \sharp\Pi > 1$.

For the sake of performance in a distributed memory multiprocessor it is necessary that there will often be more than one copy of a single virtual page. However, the application programmer wants to think in terms of just one copy, so that when a change is made by any one processor, that change should be seen immediately by all other processors. To enforce this restriction, when a write is made to a particular virtual page that page may not be replicated anywhere in the system. This assumption guarantees that any subsequent read of the written location will see the new value, because the version of the page that is read must itself be a copy of the one that was written. We are currently engaged in work that relaxes this restriction, both to support remote-update schemes that maintain consistency among copies by multicasting writes, or that permit copies of a page to grow mutually inconsistent, so long as no processor ever accesses a stale word on a page. Both of these extensions make it difficult—perhaps impossible—to design a computationally tractable optimal policy; the issues involved are beyond the scope of this paper.

Some systems may not have all of the features described above. The BBN Butterfly, for example, has memory at each processor but no caches and no global memory; it can be modeled by setting G and g to infinity. In a coherently cached system where it is not possible to read a single word from a line stored in a different cache, r would be infinite. If lines could only be loaded from main memory and not directly from another cache, then R would be infinite also.

2.2 Traces

A *trace* T is a list (T_t) of references indexed by Time. These references are meant to capture all the memory activity of all processors, in order, over the lifetime of the program. We make the important simplifying assumption that a total ordering exists, and that it is invariant, regardless of hardware model and policy decisions.

The word "Time" (with a capital "T") represents the index of a particular memory reference within a trace; it is not directly related to the execution time of a program. Cost is our analogue of execution time. Thus, regardless of the policy or hardware considered

in a particular execution, the Time of the trace is the same. The Time set, τ, is a set of integers from 0 to $n-1$ where $n = \sharp\tau$, the number of references in the trace. A *reference* is a triple (a, j, w) where a is the memory address of the word being referenced, $j \in \Pi$ is the processor making the reference, and w is either **read** or **write**. If ρ is the set of all possible references, a trace $T \in \rho^{\tau}$. Trc(μ) denotes the set of all traces for machine μ.

In practice, a change 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; our approach ignores this. 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 [29], could result in better quality results, but only at a significant cost for maintaining a global notion of time (e.g. synchronizing on every simulated machine cycle). In our simulation environment we have performed a series of experiments designed to measure the sensitivity of our results to changes in instruction interleaving; we report on these experiments in section 4.2.

2.3 Placements and Policies

A trace describes an application without specifying the location(s) within the machine at which pages reside over Time. These locations are known as a placement; they are chosen by a policy. As noted above, we assume that memory and cache space is unlimited, that contention is not a significant contributor to cost, and that the references that make up a trace are not dependent on the placement chosen for data (more on this later). Placement decisions made for different pages therefore have no impact on one another, allowing us to assume that policies treat pages independently. We therefore limit our presentation, without loss of generality, to the references made to a single page. To obtain the overall

cost of an application, sum the costs for its pages.

Formally, a *placement* P is a Time-indexed list (P_t) of location sets, where $P_t \subseteq M$ and $\sharp P_t > 0$ and $(T_t.\text{type} = \textbf{write}) \Rightarrow (\sharp P_t = 1)$. That is, each placement set is non-empty, and is a singleton whenever the corresponding reference is a write. A *policy*, $\mathcal{P}$, is a mapping from traces to placements. Given a machine μ the set of all policies for that machine is denoted $\text{Pol}(\mu)$.

2.4 Cost

The function c maps a trace and a valid placement for that trace into an integer, called the *cost* of the placement for the trace. The cost of a placement on a trace is the sum of two components: the cost due to references and the cost due to page movement. The reference component c_{ref} is defined as:

$$c_{\text{ref}}(P,T) \equiv \sum_{t=0}^{n-1} \begin{cases} 1 & \text{if } T_t.\text{proc} \in P_t \\ g & \text{if } \textbf{global} \in P_t \text{ and } T_t.\text{proc} \notin P_t \\ r & \text{otherwise} \end{cases} \quad (1)$$

That is, each reference to a local page costs 1; g is the cost for each reference to a page that is global memory, but not in local memory (assuming that global memory exists); r is the cost for each reference that must be made to a page in some other processor's memory. The page movement component c_{mv} is the cost required to move from one location set to another.

$$c_{\text{mv}}(P,T) \equiv \sum_{t=1}^{n-1} \begin{cases} G \cdot \sharp(P_t \setminus P_{t-1}) & \text{if } \textbf{global} \in P_{t-1} \cup P_t \\ R \cdot \sharp(P_t \setminus P_{t-1}) & \text{otherwise} \end{cases} \quad (2)$$

The sum here runs from 1 to $n-1$ instead of from 0 to $n-1$, because no movement cost is charged for the initial placement of the page at $t = 0$. The movement component of the cost is simply what is required to move the page into any new locations that it assumes.

Finally, then, $c(P,T) \equiv c_{\text{ref}}(P,T) + c_{\text{mv}}(P,T)$. The related function $c_{\text{po}}(\mathcal{P},T) \equiv c(\mathcal{P}(T),T)$ maps policies and traces to cost. Since c and c_{po} are similar in meaning, and should be easy to tell apart from context, we will drop the "po" and use c for both.

2.5 Optimality

Given a machine μ and a trace $T \in \mathrm{Trc}(\mu)$, a placement $P \in \mathrm{Plc}(T)$ is said to be *optimal* if $\forall Q \in \mathrm{Plc}(T) : c(P,T) \leq c(Q,T)$. Similarly, a policy $\mathcal{P} \in \mathrm{Pol}(\mu)$ is optimal if $\forall \mathcal{Q} \in \mathrm{Pol}(\mu), \forall T \in \mathrm{Trc}(\mu) : c(\mathcal{P},T) \leq c(\mathcal{Q},T)$. That is, a placement for a trace is optimal if it has cost no greater than that of any other placement for that trace; a policy for a machine is optimal if it generates an optimal placement for any trace on that machine.

A policy $\mathcal{P} \in \mathrm{Pol}(\mu)$ is *on-line* if $\forall T, T' \in \mathrm{Trc}(\mu), \forall i \in 0..n-1 : (T_{0..i} = T'_{0..i}) \Rightarrow (\mathcal{P}(T)_{0..i} = \mathcal{P}(T')_{0..i})$. In other words, $\mathcal{P}$ is on-line if the portion of any placement generated by $\mathcal{P}$ for Time 0 to i depends only on the references made up to and including Time i; i.e. iff $\mathcal{P}$ uses no future knowledge. A policy is *off-line* if it is not on-line.

PROPOSITION: *Given machine μ, any optimal policy $\mathcal{O} \in Pol(\mu)$ is off-line.*

PROOF: Let machine μ with processor set Π, memory set M, and parameters r, g, R and G and optimal policy $\mathcal{O} \in \mathrm{Pol}(\mu)$ be given. Because $\sharp\Pi = p > 1$, we may choose distinct processors $p_1, p_2 \in \Pi$.

Consider trace T_1 defined to be $10R$ writes by p_1 followed by 1 write by p_2 followed by $10R$ writes from p_1. The only optimal placement P_1 for T_1 starts the page at p_1 at the beginning of the execution and leaves it there for the entire run. Consider now trace T_2 defined to be $10R$ writes by p_1 followed by $10R$ writes by p_2. The only optimal placement P_2 for T_2 starts the page at p_1 and moves it to p_2 at Time $10R$. Since $\mathcal{O}$ is optimal and P_1 and P_2 are the unique optimal placements for T_1 and T_2 respectively, $\mathcal{O}(T_1) = P_1$ and $\mathcal{O}(T_2) = P_2$. Since T_1 and T_2 are identical up to reference $10R+1$, but yet $\mathcal{O}(T_1)$ and $\mathcal{O}(T_2)$ differ at Time $10R$, we conclude that $\mathcal{O}$ is off-line. $\square$

THEOREM: *Given machine $\mu = (\Pi, M, g, r, G, R)$, $s > 0$, trace T and optimal policy $\mathcal{O} \in Pol(\mu)$, $\mathcal{O}(T)$ is an optimal placement for machine $\mu' = (\Pi, M, s(g-1)+1, s(r-1)+1, sG, sR)$.*

PROOF: Let machine μ, trace T, s and $\mathcal{O}$ be given and μ' defined as in the hypothesis. Define P to be $\mathcal{O}(T)$. Let placement Q for T be given. Because P is optimal on μ, $c(P,T) \leq c(Q,T)$. Define λ_P to be the number of local references made by P, ρ_P the number of remote references, γ_P the number of global references, Φ_P the number of remote moves and Γ_P the number of global moves. Define λ_Q, ρ_Q, γ_Q, Φ_Q and Γ_Q similarly for placement Q. By

definition of cost, $c(P,T) = r\rho_P + g\gamma_P + \lambda_P + R\Phi_P + G\Gamma_P \leq r\rho_Q + g\gamma_Q + \lambda_Q + R\Phi_Q + G\Gamma_Q = c(Q,T)$. Since $\lambda_P + \rho_P + \gamma_P = \sharp T = \lambda_Q + \rho_Q + \gamma_Q$ we may subtract them from both sides of the inequality, and since $s > 0$, we may multiply without changing the sense of the inequality, giving $s(r-1)\rho_P + s(g-1)\gamma_P + sR\Phi_P + sG\Gamma_P \leq s(r-1)\rho_Q + s(g-1)\gamma_Q + sR\Phi_Q + sG\Gamma_Q$. Adding in the terms equal to $\sharp T$ subtracted above and observing the definition of c' yields $c'(P,T) \leq c'(Q,T)$. Since placement Q was arbitrary, by definition of optimality P is optimal on μ'. □

COROLLARY: *Given μ, s, μ', T and $\mathcal{O}$ as in the previous theorem, and given optimal policy $\mathcal{O}'$ for μ', if n is the length of trace T, c is the cost function for μ and c' the cost function of μ', then* $\frac{c'(\mathcal{O}'(T),T)}{n} = 1 + s(\frac{c(\mathcal{O}(T),T)}{n} - 1)$.

PROOF: Define P to be the placement $\mathcal{O}(T)$ (for machine μ). Define λ, ρ, γ, Φ and Γ as in the previous proof. If c is the cost function for machine μ, then $c(P,T) = \lambda + r\rho + g\gamma + R\Phi + G\Gamma$. Since every reference made in the trace is either local, global or remote, $\lambda + \gamma + \rho = n$. Therefore, $c(P,T) = n + (r-1)\rho + (g-1)\gamma + R\Phi + G\Gamma$ and $\frac{c(P,T)}{n} = 1 + \frac{(r-1)\rho + (g-1)\gamma + R\Phi + G\Gamma}{n}$. By the previous theorem, P is optimal for T on machine μ'. Since by hypothesis $\mathcal{O}'$ is optimal, $c'(\mathcal{O}'(T),T) = c'(P,T)$. By definitions of cost and μ', $c'(P,T) = \lambda + (s(r-1)+1)\rho + (s(g-1)+1)\gamma + sR\Phi + sG\Gamma = n + s(r-1)\rho + s(g-1)\gamma + sR\Phi + sG\Gamma$. Dividing by n, we have $\frac{c'(P,T)}{n} = 1 + s\frac{(r-1)\rho + s(g-1)\gamma + sR\Phi + sG\Gamma}{n}$. Subtracting one from the final formula for $c(P,T)/n$ above and substituting yields $\frac{c'(\mathcal{O}(T),T)}{n} = \frac{c'(P,T)}{n} = 1 + s(\frac{c(P,T)}{n} - 1) = 1 + s(\frac{c(\mathcal{O}(T),T)}{n} - 1)$. □

3 Computing Optimal NUMA Placements

A placement can be thought of as a Time-ordered walk through the space of possible page replication sets. At each point in Time the fundamental question to be answered is whether to leave a page in remote or global memory, or to migrate or replicate it into global or remote memory. The global memory option may not exist on a NUMA machine. The remote reference option may not exist on a cache-coherent UMA machine. In any case, brute-force exploration of the search space is obviously impractical: the number of possible placements is on the order of n^{2^p}.

For the sake of expository clarity we present two versions of our algorithm, first employ-

```
for m ∈ M cost_so_far[m] ← 0
for t ← 0 to n − 1                                  /* for all references in trace */
    cheap_cost ← cost_so_far[global]
    C ← G; cheapest ← global
    for m ∈ (M \ {global})
        if cost_so_far[m] + R < cheap_cost + C
            cheap_cost ← cost_so_far[m]
            C ← R; cheapest ← m
    new_cost_so_far[T_t.proc] ← MIN (
        cost_so_far[T_t.proc] + 1,                  /* use copy already here */
        cost_so_far[cheapest] + C + 1)              /* get it now */
    new_cost_so_far[global] ← MIN (
        cost_so_far[global] + g,                    /* use global copy */
        cost_so_far[cheapest] + G + g)              /* migrate from cheapest */
    for m ∈ (M \ {T_t.proc ∪ global})
        new_cost_so_far[m] ← MIN (
            cost_so_far[m] + r,                     /* use copy already there */
            cost_so_far[cheapest] + C + r)          /* migrate from cheapest */
    cost_so_far ← new_cost_so_far                   /* update whole array */
return MIN_{m∈M} (cost_so_far[m])
```

Figure 1: Algorithm for computing optimal cost without replication

ing dynamic programming to make the complexity linear in n, and then making placement decisions for an entire read-run at the Time of the following write, to make the complexity linear in p. Both algorithms compute the cost of an optimal placement rather than the placement itself. Since the computations are constructive it is simple to extend them to produce the actual placement.

3.1 Computing Optimality Without Replication

We developed the first version of the optimal algorithm (Figure 1) by assuming that replications are prohibited. This algorithm resembles the solution to the full version of the problem, but is simpler and easier to understand. To fit it into the framework of the cost metric presented in section 2.4, we pretend that all references are writes.

The algorithm uses dynamic programming to determine, after each reference, the cheapest way that the page could wind up in each possible memory location. At Time t, for each memory, the cheapest way that the page could wind up there is necessarily an extension of the cheapest way to get it to some (possibly different) location at Time $t-1$. The principal

data structure, then, is an array of costs (integers), "cost_so_far," indexed on memories $m \in M$. At Time t, cost_so_far[m] contains the cost of the cheapest placement for the trace $T_{0..t}$ that would end with the page at m. At the end of the algorithm, the cost of the cheapest overall placement is the minimum over $m \in M$ of cost_so_far[m]. The key to dynamic programming, of course, is that while the algorithm never looks back in the trace stream, it does not know where the page might be located at the Time that a reference is made. Only at the end of the trace is the actual placement known.

The algorithm in Figure 1 runs in time $O(np)$. There exists another version that runs in time $O(n)$. It uses the observation that there is always an optimal placement that never moves a page to a processor other than the one making the current reference. The faster algorithm is not included because it is harder to follow and not much more interesting than the algorithm presented.

3.2 Incorporating Replication

The obvious extension for the general case with replication is simply to enlarge the set M to include all possible replication states, and to enforce coherence by assuming that the transitions into non-singleton states are of infinite cost when the reference is a write. Unfortunately, this extension increases the time complexity of the inner loops of the algorithm from $O(p)$ to $O(2^p)$ for the cases where the reference is a read. This is a severe penalty even on the 7-node machine used for experiments in section 4; for large machines it is out of the question.

Fortunately, it is not necessary to search this large state space. Name the Time interval between two writes with at least one read and no other writes between them a *read-run*. Because of the coherence constraint, at the beginning and end of a read-run the page state must be a singleton. There is no cost benefit in removing a copy of the page inside of a read-run, so we can ignore all such placements. Similarly, if the page will be replicated to a memory inside of the read-run, there is no cost penalty involved in making the replication on the first reference of the read-run. So, for any given read-run all that needs to be decided is the set of processors to which to replicate; there exists an optimal placement that replicates to these processors at the beginning of the read-run and destroys the replicates on the terminal write, without changing the replication state in between. Furthermore, the set of

```
FUNCTION read_run_cost (start : location; rep_set : set of location;
    reads_from : associative array [processor] of integer) : integer
running_total ← 0
for each j ∈ domain (reads_from)
    if j ∈ rep_set
        running_total + ← reads_made[j]
    else
        running_total + ← r * reads_made[j]
    if start ∈ rep_set
        running_total + ← R * (♯rep_set − 1)
    else
        running_total + ← R * ♯rep_set
return (cost_so_far[start] + running_total)          /* cost_so_far is global */
```

Figure 2: Function to compute the cost of a read-run, no global memory

processors to which to replicate during a given read-run depends only on the locations at the writes before and after the run, and on the number of reads made by each processor during the run.

Armed with these observations, we may extend the algorithm in Figure 1 to the general case. The new version appears in Figure 3. The function in Figure 2 computes the cost of a read-run, given the starting location, the replication set and the number of reads made by each processor during the run. For the sake of simplicity, this function assumes that there is no global memory. The modifications required to handle it are straightforward.

The new algorithm still uses dynamic programming, but while the state space was updated on every reference in the old version, it is only updated on writes in the new. The space that is tracked remains M. In addition, while formerly at each step we had to consider the possibilities of starting the page at the current location, or in the cheapest location among the rest of the processors, we must now also consider the possibility that a processor may effectively become the cheapest by virtue of a savings in references during the preceding read-run, even if these references do not justify replication.

```
refs_to_pay_for_repl ← R/(r − 1)
for j ∈ Π cost_so_far[j] ← 0
reads_from ← empty                                  /* associative array */
for t ← 0 to n − 1                                  /* for all references in trace */
    if T_t.type = read
        if T_t.proc ∈ domain (reads_from)
            reads_from[T_t.proc] + ← 1
        else
            reads_from[T_t.proc] ← 1
    else /* write */
        repl_procs ← {j ∈ domain (reads_from) | reads_from[j] > refs_to_pay_for_repl}
        cheapest ← j ∈ M such that cost_so_far[j] is least
        min_nonrep_proc ← j ∈ (Π \ repl_procs)
            such that cost_so_far[j] −(r − 1) * reads_from[j] is least
            /* if repl_procs = Π, pick an arbitrary processor */
        for j ∈ Π
            /* We follow one of three possible replication patterns: start where we finish,
               start at the place that was cheapest to begin with, or start at the place that
               was cheapest but not in the set of memories for which the number of reads
               was enough to offset the cost of replication by itself. */
            new_cost_so_far[j] ← MIN (
                read_run_cost (j, {j} ∪ repl_procs, reads_from),
                read_run_cost (cheapest, {cheapest, j} ∪ repl_procs, reads_from),
                read_run_cost (min_nonrep_proc, {min_nonrep_proc, j} ∪ repl_procs, reads_from))
            if T_t.proc = j /* write by ending processor */
                new_cost_so_far[j] + ← 1
            else /* write by another processor */
                new_cost_so_far[j] + ← r
        cost_so_far ← new_cost_so_far               /* update whole array */
        reads_from ← empty
/* The entire trace has been processed. Clean up if we're in a read-run. */
if T_{n−1}.type = write
    return MIN_{j∈Π} (cost_so_far[j])
repl_procs ← {j ∈ domain (reads_from) | reads_from[j] > refs_to_pay_for_repl}
cheapest ← j ∈ M such that cost_so_far[j] is least
min_nonrep_proc ← j ∈ (Π \ repl_procs)
    such that cost_so_far[j] −(r − 1) * reads_from[j] is least
    /* if repl_procs = Π, pick an arbitrary processor */
for j ∈ Π
    new_cost_so_far[j] ← MIN (
        read_run_cost (j, {j} ∪ repl_procs, reads_from),
        read_run_cost (cheapest, {cheapest, j} ∪ repl_procs, reads_from),
        read_run_cost (min_nonrep_proc, {min_nonrep_proc, j} ∪ repl_procs, reads_from))
return MIN_{j∈Π} (new_cost_so_far[j])
```

Figure 3: Optimal policy computation, no global memory

4 Experimental Results for NUMA Memory Management

The goal of a NUMA placement policy is to devote as little time as possible to accessing memory and to moving data from one memory to another. Several groups have studied implementable kernel-level policies that replicate and migrate pages, generally in response to page faults. Holliday explored migration based on periodic examination of reference bits [19], and suggested [18] that good dynamic placement of code and data offers little additional benefit over good initial placement. Black and Sleator devised a dynamic page placement algorithm with provably optimal worst-case behavior [8], and Black, Gupta and Weber simulated it on address traces [6], but their approach does not appear to exploit "typical" program behavior, and requires a daunting amount of hardware assistance. Cox and Fowler's PLATINUM system [13] for the BBN Butterfly freezes pages that move too often, but adapts to changes in program behavior by un-freezing pages periodically. LaRowe, Ellis, and Kaplan [26, 21] compared competing policies on the Butterfly by implementing many alternatives in their DUnX version of BBN's operating system. Our work with Bob Fitzgerald [9] on the IBM ACE multiprocessor workstation confirmed the value of a good static placement on machines with comparatively low remote access penalties, and argued that even very simple kernel-level policies are likely to achieve most of the benefits available without application-specific knowledge.

The study of NUMA management via real implementations is attractive in terms of concreteness: experimental results are of nearly unarguable validity. It is difficult, however, to experiment with different multiprocessor architectures, or to consider architectural features that have not yet been implemented. It is likewise difficult to construct good implementations of more than a small number of policies in a reasonable period of time. Most important, it is difficult to quantify results. In evaluating the ACE system, for example, we were able to measure performance when all data references were remote and to predict what performance would be if all data references were local (which of course they cannot be, because of coherency requirements). We could compare achieved performance to these extreme bounds, but not to any notion of the best *achievable* results.

Optimal analysis allows us to address these limitations. We explain our experimental environment, including the trace collection mechanism and application suite, in section 4.1. Section 4.2 describes a series of experiments designed to establish confidence in the valid-

ity of our analysis technique. The results in section 4.3 show the dependence of program performance on two basic NUMA hardware parameters: the relative cost of a block transfer (as compared to a series of individual remote accesses), and the size of a data page. Section 4.4 compares the performance achieved by several implementable policies with that of the optimal policy, and demonstrates how the placement decisions made by the optimal policy can be used to guide the design of an appropriate on-line policy for a given hardware architecture. Many of the results are drawn from previous work, in which we and some of our colleagues employed off-line optimal analysis to explore the extent to which NUMA policies should be tuned to architectural parameters [10].

4.1 Experimental Tools

4.1.1 Trace Collection

We collected our traces on an IBM ACE multiprocessor workstation [17] 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 [7] 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 [30][5] to two orders of magnitude faster [22].

The ACE tracer maintains a single global 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 from other processes, we ran our applications in single-user mode, with no other system or user processes running. Furthermore, all writable memory was placed in the ACE's global memory, to prevent "gaps" from appearing in the trace when the kernel decided to move a page.

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

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
mp3d	63.1	0
cholesky	38.7	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 I: Trace Sizes and Breakdowns (in millions of data references)

4.1.2 Application Suite

We traced a total of eighteen 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 I shows the sizes of our traces in millions of references. The Presto and EPEX systems 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 [28] is an extension to FORTRAN developed for parallel programming at IBM. EPEX applications are typically numeric. The programmer explicitly identifies private and shared data in the 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 or the SPLASH suite [27], and ported to the ACE. In the first two cases, 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 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 [16]; `sorbyr` and `sorbyc`, a pair of red-black successive over-relaxation programs [24] 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. `Cholesky` and `mp3d` are applications from the Stanford Parallel Applications for SHared memory (SPLASH) benchmark suite [27]. `Cholesky` does a Cholesky factorization, while `mp3d` simulates rarefied airflow over a wing particle by particle. In many of the C-Threads applications two-dimensional numerical matrices are represented as an array of pointers to rows, as recommended in *Numerical Recipes in C* [25]. In these programs the unit of data-sharing is the row, so data sharing patterns exhibit a fairly coarse grain.

Presto [5] 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 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 [4]. The prefix `p-` indicates a Presto application.

4.2 Validation of the Trace Analysis Technique

Our tracer slows down the execution of a program by a factor of 200 or more (depending on how many of the application's instructions make memory references, how much the floating accelerator is used, and so on). This will have some effect on the order in which references are made. While all processors are slowed uniformly, the *dilation* effect will bury any difference in execution times of the various machine instructions. On the ACE's processor, most instructions take only 1 cycle to execute. The notable exceptions are memory reference instructions and floating point operations, which take somewhat more time depending on the instruction, on whether the memory is busy, etc. Koldinger *et al.* [20] investigated these sorts of effect in the related area of coherent cache simulation, and found the performance differences due to dilation to be negligible. Since our optimal policy guarantees small changes in cost in response to small changes in the trace input (it is, in some sense, continuous), it is natural to expect its performance to be even less affected by dilation.

As noted in section 2.2, a more fundamental problem with the evaluation of multiprocessor memory systems based on static trace interleavings is a failure to capture the influence of the simulated system on the references that "should" have occurred. In our system, this feedback should appear in two forms: fine-grain changes in instruction interleaving, and coarse-grain "reference gaps" in the activity of individual processors. Instruction timings depend on whether the operands of loads and stores are local or remote. If two policies place a page in a different location at different points in time, then instructions will execute faster on some processor(s) and slower on others, and the interleaving of instructions from different processors will change. Similarly, when a policy decides to move a page, the processor performing the move will stop executing its user program until the move is complete. Since this could potentially take a long time (particularly in a system with large pages and/or large interprocessor latencies), other processes might make a large number of references in the interim. Since the times at which the page moves would occur are not known when the applications are traced, and in general depend on the parameters of the simulation later performed on the trace, no such gaps appear in the traces.

To evaluate the impact of changes in fine-grain instruction interleavings, independent of the changes in memory cost of the locality decisions that caused those changes, we wrote a

filter program that reorders individual references in a trace, with a probability that is high for nearby references, and drops off sharply for larger Time spans. More specifically, the filter keeps a buffer of 100 references from the incoming trace stream. Initially, this buffer is filled with the first 100 references. The filter then randomly chooses an entry from the buffer, emits the oldest buffered reference made by the processor whose entry was selected, and reads a new reference to replace it. We inserted the filter in front of our trace analyzer, and measured the degree to which it changed the cost of the optimal policy. The maximum difference in optimal performance among all of the applications for the ACE machine model was 0.007%. For the Butterfly machine model it was 0.1%.

To evaluate the impact of reference gaps, we wrote a filter that randomly introduces such gaps, and again re-ran the optimal policy. The filter operates by reading the unmodified trace, and with probability one in 30,000 introduces a "gap" on one processor for 4000 references. A gap is introduced by taking any references made by the chosen processor and placing them in a queue. Once the gap has ended, as long as there are saved references, one of them will be emitted instead of a fresh reference with probability 2/3. The values 30,000 and 4000 were selected arbitrarily, but were chosen conservatively in the sense that a page moves typically do not occur as often as every 30,000 references, and 4000 references is somewhat large for the amount of time for a page move. The 2/3 frequency is arbitrary. This filter induced performance changes up to .06% in the ACE model and 0.34% in the Butterfly model.

Table II displays the differences between filtered and unfiltered results for both filters and ACE and Butterfly models as a percentage of the total cost. Differences are absolute values; sometimes the filtered values were smaller, sometimes they were larger. Values less than 0.001% are reported as 0.

4.3 Evaluating Architectural Options Independent of Policy

4.3.1 Block Transfer Speed

It is often possible on a NUMA machine to construct a hardware-assisted block transfer operation that moves data through the interconnection network at a much higher bandwidth than can be achieved with a software copy loop. If nothing else, the block transfer

Application	ACE Local	Bfly Local	ACE Gap	Bfly Gap
e-fft	.002%	.012%	0	.18%
e-simp	.001%	.001%	.001%	.039%
e-hyd	0	0	0	.001%
e-nasap	.007%	.03%	.004%	.15%
gauss	0	0	.02%	.07%
chip	0	0	0	0
bsort	0	0	0	0
kmerge	0	0	0	0
plytrace	0	0	.01%	.06%
sorbyc	0	.01%	0	.01%
sorbyr	0	.007%	0	.07%
matmult	0	.03%	0	.25%
mp3d	0	0	0	.002%
cholesky	0	.004%	0	.03%
p-gauss	0	0	0	.24%
p-qsort	0	.01%	0	.24%
p-matmult	0	.01%	0	.04%
p-life	0	.005%	.002%	.34%

Table II: Percentage optimal performance change due to local and gap perturbations

operation can avoid network transaction set-up costs for each individual word. Cox and Fowler argue [13] that a fast block transfer is essential for good performance in a NUMA machine. Working with them [10], we employed off-line analysis to evaluate this claim on a machine resembling the Butterfly and on a machine resembling the ACE, in which the relatively fast performance of global memory makes aggressive page migration and replication less essential.

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 the graphs; 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

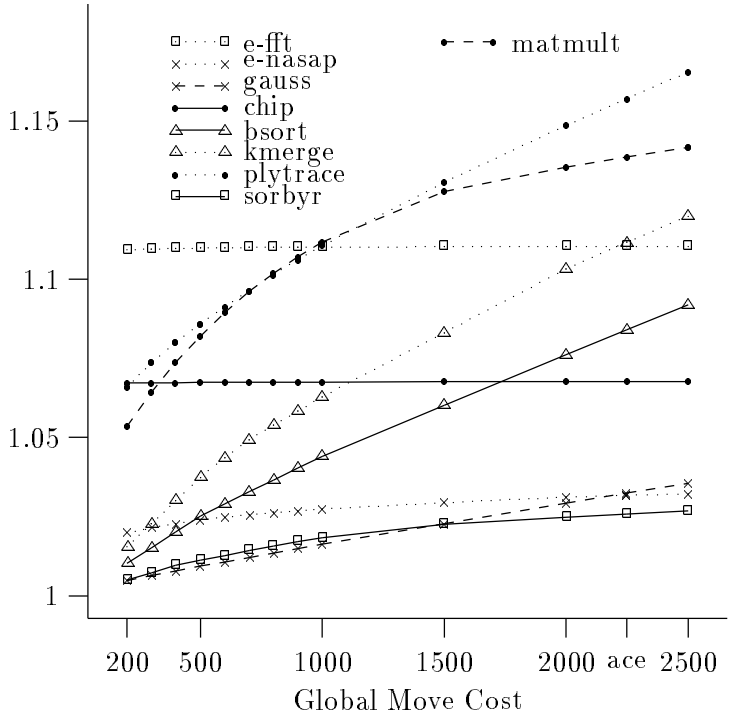

Figure 4: MCPR vs. G for optimal, g=2, r=5

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 of 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.

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. 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. Because of the theorem and its corollary presented in section 2.5, it is possible to derive the optimal performance

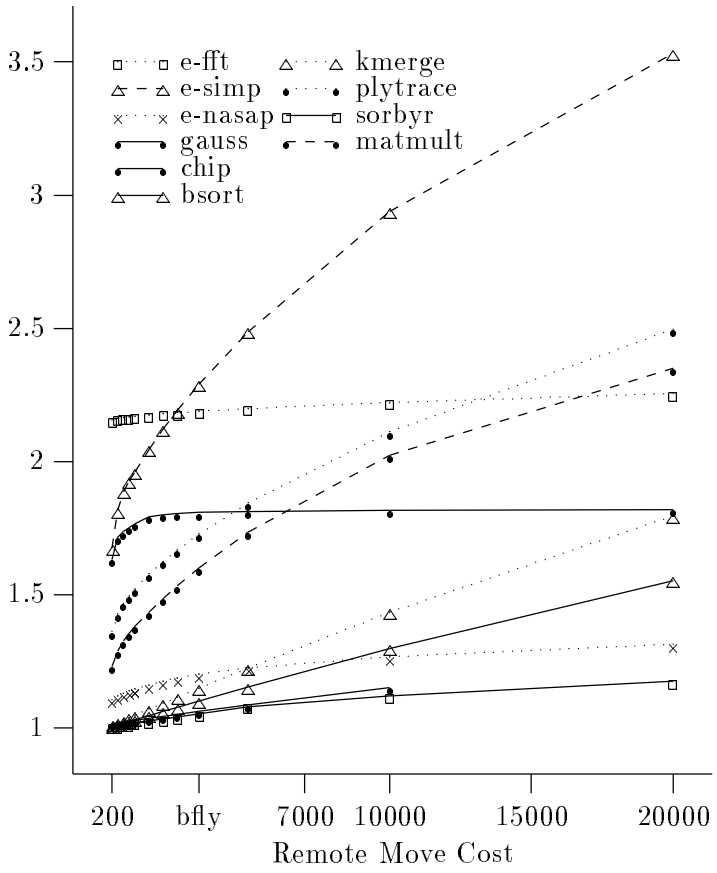

Figure 5: MCPR vs. R for optimal, no global, r=15

for all pairs of r and R values.

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 in the number of moves per page curve 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

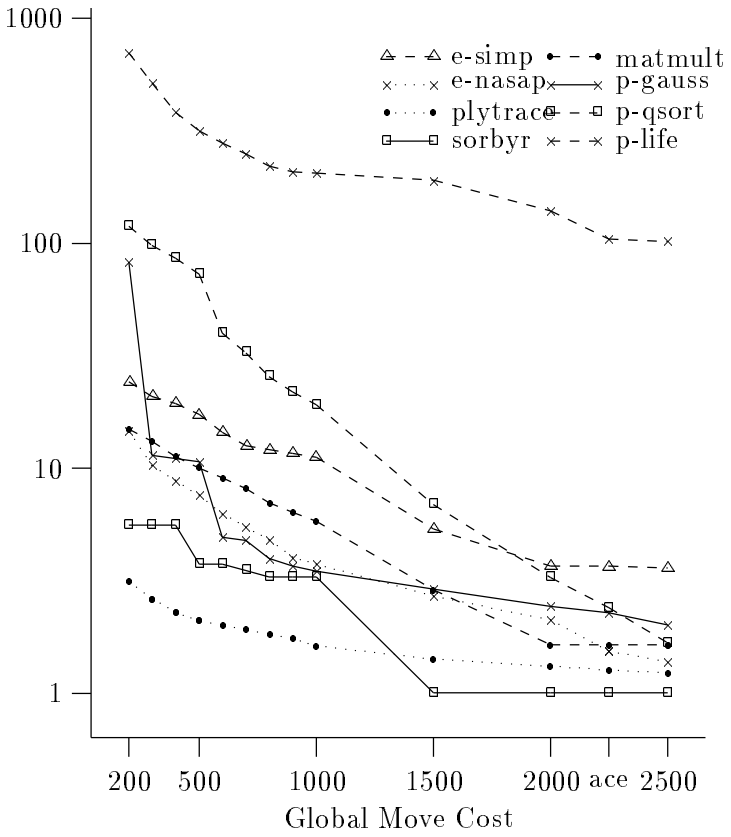

Figure 6: Mean Page Moves per Page for optimal, g=2, r=5

speedup increases at faster speeds. However, moving across a jump point will not produce a corresponding speedup in performance: the jump is in the *slope* of the cost curve, not in the cost itself.

4.3.2 Page Size

Another attribute of NUMA hardware that must be evaluated by the architect is the page size. The tradeoffs for small versus large pages are well known for uniprocessor systems: increased fragmentation in exchange for reduced overhead. Multiprocessor systems are complicated by the additional issue of false sharing: coherence operations may be caused not by sharing of data between processors, but simply by the accidental co-location on a page of data being used by different processors. Large pages may therefore *decrease* performance. Eggers and Jeremaissen [15] describe this effect in coherently cached systems; they report that in some applications it accounts for as much as 40% of all cache misses.

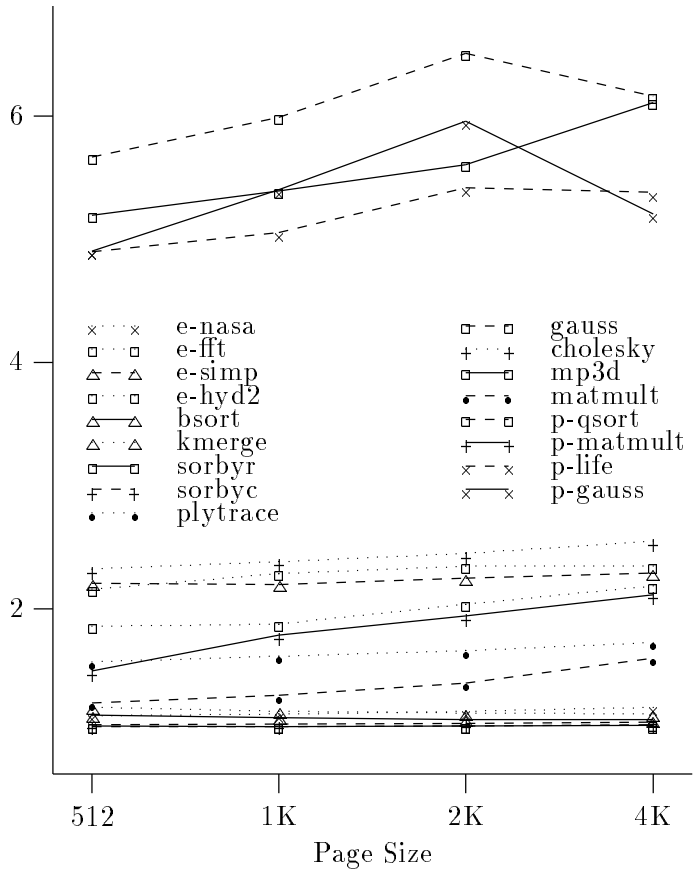

Figure 7: Optimal MCPR varying with page size for Butterfly model

We used optimal analysis to determine the effect of changing the page size in two different NUMA machines. The curves in Figure 7 show the variation in mean cost per reference (MCPR) of the optimal policy as the page size s varies from 512 bytes to 4K bytes on a machine resembling the Butterfly. Figure 8 shows similar curves (but with a log scale MCPR axis) for a hypothetical NUMA machine in which the inter-processor latency relative to local cache speed is much higher than on the Butterfly, as is likely for newer designs with faster processors and local memory/caches. In both models, G and g are set to ∞: there is no global memory. For the Butterfly, r and R are 15 and $3s/4 + 200$, respectively: remote references are slow but block transfers are relatively fast. The constant 200 allows for the overhead of handling a page fault or interrupt, mapping a page into some process's virtual address space, and possibly removing an out-dated copy of the page. For the high-latency machine, r is 100 and R is $s/2+200+75$: the latency for a remote reference is much higher, and while the bandwidth of the interconnect is better than the butterfly (thus the $s/2$

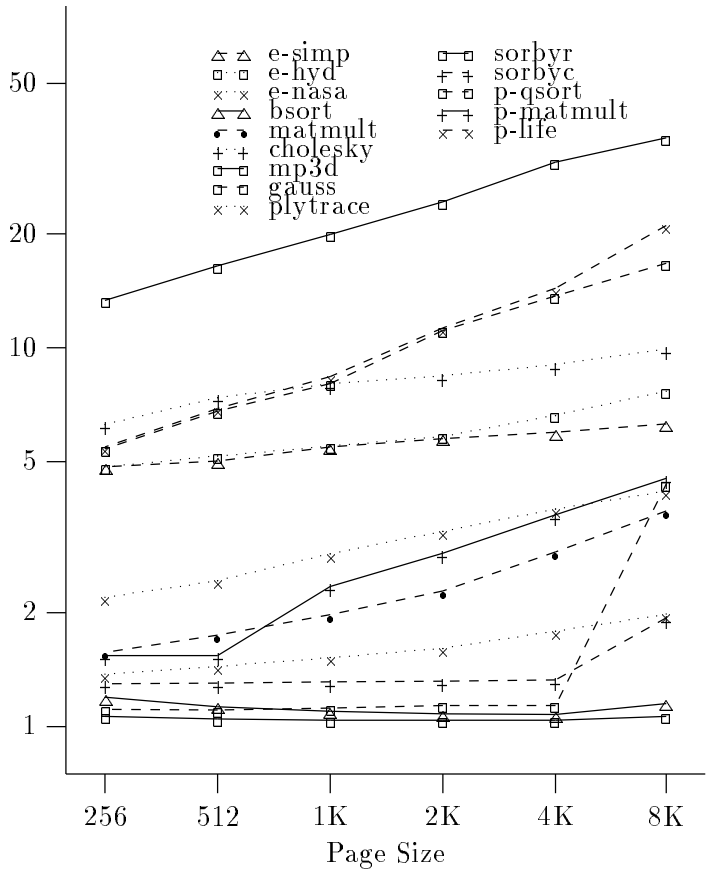

Figure 8: Optimal MCPR varying page size for high-latency NUMA model

instead of $3s/4$), the time to find the location from which to migrate becomes significant. We assume a directory-based coherence protocol in which two remote references are required: one to the home node of the data, in order to identify the owner, and one to the owner itself. These two references contribute 200 to the overhead; the kernel is assumed to be faster in handling the fault than it was on the Butterfly (partially because it can overlap page table changes with the very expensive remote references), and contributes only 75.

The amount that performance improves with smaller pages varies greatly between these two architectures. While there is a small effect on the Butterfly, in general it is insufficient to warrant exploiting. On the other hand, in the more modern, higher-latency NUMA, it has a predominant effect on performance. We are currently using optimal analysis to compare several system models, all with latencies and bandwidths comparable to the non-Butterfly machine presented here, but with varying page sizes, inter-processor latencies, software overheads, etc. Preliminary results indicate that page size is the most important

determinant of performance in these systems.

Without using optimal analysis, it would be difficult to determine whether any measured performance differences were really due to smaller pages, or simply to some artifact of the particular on-line policy chosen for the study. As it is, we can safely conclude that our result about page size is real, and that smaller pages could be very useful when building NUMAs with interprocessor latencies and bandwidths comparable to those in our non-Butterfly model.

4.4 Evaluating Implementable Policies Against the Optimal Baseline

4.4.1 A Set of On-Line Policies

In addition to the optimal policy, we have evaluated three implementable alternatives. Two of them have been used in real systems and are described in prior papers: the ACE policy [9] 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 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 ($G = 2 * pagesize + 200$). 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 small, fixed 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, slower remote memory than the ACE ($r = 15$), 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 proces-

sor 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, Cox and Fowler chose t_1 and t_2 to be $10ms$ and $1s$ respectively. Since time is unavailable 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 for a local/remote machine, but could use global memory to hold its frozen pages; we arrange for it to do so when simulating a machine like the ACE.

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 Delay policy is based on this mechanism: a counter in each of the TLB entries on each processor, 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 c. 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 [8] 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 $c = 100$ 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.

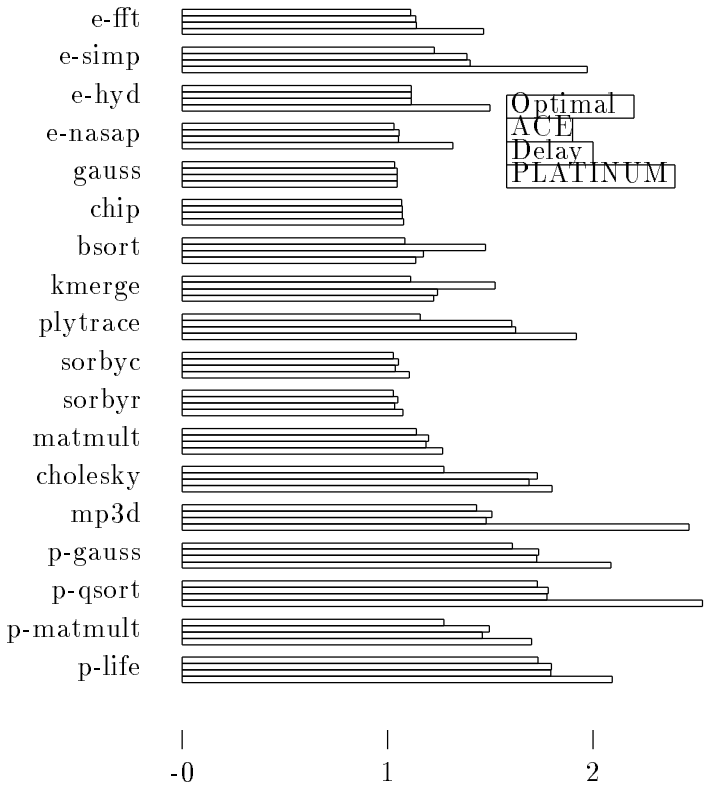

Figure 9: MCPR for ACE Hardware Parameters

4.4.2 Comparative Policy Performance

The performance of each of our policies on each of our applications, expressed as Mean Cost Per Reference (MCPR), appears in Figures 9 and 10–11, for architectures resembling the ACE and the Butterfly, respectively. Each application has a group of four bars, which represent the performance of the Optimal, ACE, Delay and PLATINUM policies, 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

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

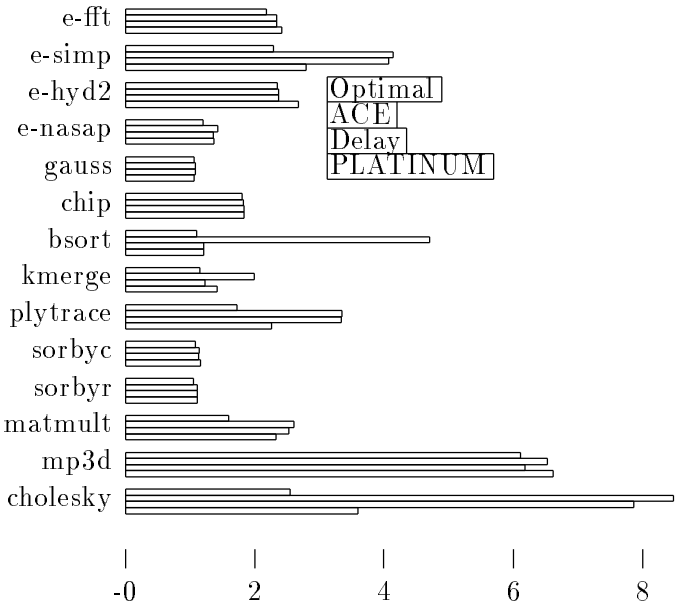

Figure 10: MCPR for Butterfly Hardware Parameters

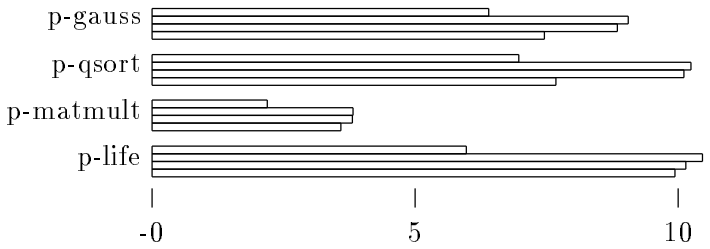

Figure 11: MCPR for Butterfly Hardware Parameters, PRESTO applications

ACE and Delay policies severely limit the extent to which this migration takes place. 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.

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.

Applications such as `e-fft` and `e-hyd`, which have only private and fine-grained shared data, will perform well with a reasonable static data placement, but this strategy will

not work well in other cases. 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 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). In [9] we estimate that programs running on the ACE spend from 25%–60% of their time referencing memory. Newer, more aggressive processor architectures will only increase this percentage, as processor improvements outstrip improvements in memory. For a program that spends 50% of its time accessing data memory, even our poorest 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.

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 machine. 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.

4.4.3 Learning from Optimal Behavior

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`).

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. The effects of management of this memory are still clearly visible in the results presented, and are 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 12 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.

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

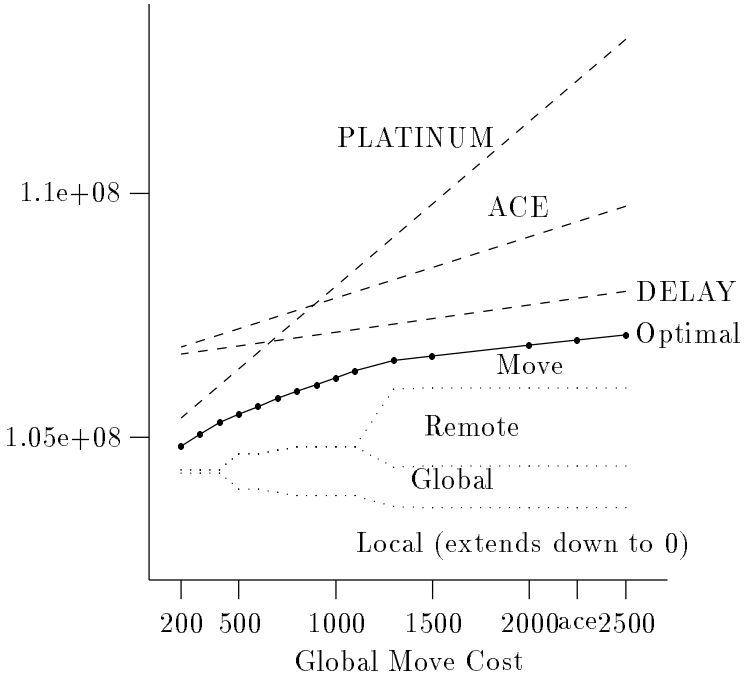

Figure 12: `sorbyr` Placement Cost vs. Page Move Cost w/ Optimal Breakdown, g=2, r=5

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. 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 12 at points near $G = 1200$ and $G = 400$ respectively. While the behavioral cost breakdown of the optimal policy undergoes large sudden changes, the cost itself as a function of behavior changes smoothly with G.

Any given policy will be oblivious to the speed of memory operations. Its curve will therefore be a straight line on a graph like Figure 4 or Figure 12, and will lie on or above the optimal curve at all points. 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. We illustrate this point in Figure 12 by including lines for on-line policies. The PLATINUM policy works best for small G, but at the cost of doing poorly for large G. Conversely, the ACE and Delay policies do well for large G, but poorly for small G. To obtain best performance over a range of page move speeds in Figures 4 and 5 (at least for the applications in which the optimal line curves

sharply), one must change the real policies accordingly.

5 Discussion

The technique of off-line optimal analysis allows evaluation of a relatively large range of shared memory multiprocessor architectures without biasing the results by selecting a particular data-movement policy. Conversely, it provides a baseline against which to compare real policy behavior. Reasoning in a formal model forces exposure of assumptions, and allows proof of theorems within that framework. All of these things provide a degree of rigor difficult to achieve in the real world. Using these methods, we have learned a number of things about NUMA systems and policies, which are summarized in this section.

From a functional point of view, NUMA machines closely resemble UMA machines with hardware cache coherence; the principal difference is that NUMA policies generally permit remote references, whereas cache coherence policies force a line to move on a cache miss or, in the case of non-cacheable data, access it only in global memory. NUMA machines implement data placement decisions in software, of course; cache-coherent machines implement them in hardware or firmware. Cache coherence policies are also likely to move data more often than NUMA policies, mainly because of the lack of remote references, but also because the comparatively small size of a cache line and the low start-up overhead of a hardware-initiated move make movement more attractive. Because they move data more, cache-coherent multiprocessors are likely to suffer more from interconnect contention.

Distributed virtual memory systems for NORMA (NO Remote Memory Access) machines also resemble hardware-based coherent caching at a functional level [23]. Since most distributed virtual memory systems don't permit remote access (they would have to implement it in the page fault handler), they may actually resemble hardware-based coherent caching more than NUMA systems do. All three kinds of systems may or may not support the remote update of multiple copies (that is, broadcasting a single word write to multiple remote copies); this is an orthogonal issue.

Our model of memory management cost can be used to describe data placement policies for UMA, NUMA, and NORMA machines, provided that contention is not a major factor in performance. Our algorithm for computing an optimal placement provides a performance

baseline for these policies, and allows us to evaluate the hardware on which they run, provided that we accept the coherence constraints, insisting that all copies of a page be up-to-date at all times, and insisting that only one copy exist at the time of a write. We are currently experimenting with heuristic off-line algorithms which, while not optimal, may arguably be used as a performance baseline for systems in which the coherence constraints are not enforced. We are particularly interested in the extent to which a policy might obtain improved performance by exploiting knowledge of false sharing in applications, allowing copies of pages written by more than one processor in a given time period, but containing no data objects used by more than processor during that same period, to grow temporarily inconsistent. We are also gathering data for a direct comparison of memory management costs on UMA, NUMA, and NORMA machines for a common set of applications. We expect hardware cache coherence to outperform software schemes in most (though not all) cases, but it appears likely that the differences will in many cases be small enough to cast doubt on the cost effectiveness of the hardware-intensive approach. If hardware placement policies indeed produce larger amounts of interconnect traffic than software placement policies, then a comparison that ignores contention is likely to be biased in favor of the hardware approach.

We hypothesize that off-line optimal analysis could fruitfully be employed in problem domains other than multiprocessor memory management. One might, for example, create a tractable algorithm for optimizing allocation of variables to registers in a compiler, given the references to the variables that are eventually made by a particular program (that is, a trace). It would then be possible not only to measure the performance of a compiler's register allocator, but also to determine the performance inherent in different register set designs (different numbers of registers, different registers for floating point, addresses and integers vs. general purpose registers, different sizes of register windows, etc.) without having to worry that effects are due to a particular compiler, and without having to worry about implementing register allocation schemes for all of the hardware variants.

For NUMA machines, off-line optimal analysis has allowed us to quantify the utility of a fast block transfer, assess the significance of varying page sizes, characterize the sorts of placement decisions that a good policy ought to be making on various sorts of machines, and estimate the extent to which policy improvements (presumably incorporating application-specific knowledge) might increase the performance of software data placement.

6 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 and Alan, 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] 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] 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.

[5] 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.

[6] D. Black, A. Gupta, and W.-D. Weber. Competitive Management of Distributed Shared Memory. In *Proc. Spring COMPCON*, pages 184–190, February 1989.

[7] D. L. Black, D. B. Golub, K. Hauth, A. Tevanian, and R. 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.

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

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

[10] 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.

[11] L. M. Censier and P. 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] A. L. Cox and R. 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] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. Technical Report 90-12-01, University of Washington, 1990.

[16] A. Garcia. *Efficient Rendering of Synthetic Images*. PhD thesis, Massachusetts Institute of Technology, February 1988.

[17] 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.

[18] M. A. Holliday. On the Effectiveness of Dynamic Page Placement. Technical report, Department of Computer Science, Duke University, September 1989. CS-1989-19.

[19] M. A. Holliday. Reference History, Page Size, and Migration Daemons in Local/Remote Architectures. In *3rd Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, April 1989.

[20] 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.

[21] R. P. LaRowe and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[22] J. R. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software: Practice and Experience*, 20(12):1241–1258, December 1990.

[23] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

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

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

[26] J. R. P. LaRowe, C. S. Ellis, and L. S. Kaplan. The Robustness of NUMA Memory Management. In *Proc., 13th ACM Symposium on Operating Systems Principles*, pages 137–151, 1991.

[27] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Available by anonymous FTP, April 1991.

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

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

[30] W. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *3rd Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, April 1989.