Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems*

Michael Marchetti, Leonidas Kontothanassis,
Ricardo Bianchini, and Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226

{mwm,kthanasi,ricardo,scott}@cs.rochester.edu

September 1994

Abstract

The cost of a cache miss depends heavily on the location of the main memory that backs the missing line. For certain applications, this cost is a major factor in overall performance. We report on the utility of OS-based page placement as a mechanism to increase the frequency with which cache fills access local memory in a distributed shared memory multiprocessor. Even with the very simple policy of first-use placement, we find significant improvements over round-robin placement for many applications on both hardware and software-coherent systems. For most of our applications, dynamic placement allows 35 to 75 percent of cache fills to be performed locally, resulting in performance improvements of 20 to 40 percent.

We have also investigated the performance impact of more sophisticated policies including hardware support for page placement, dynamic page migration, and page replication. We were surprised to find no performance advantage for the more sophisticated policies; in fact in most cases performance of our applications suffered.

1 Introduction

Most modern processors use caches to hide the growing disparity between processor and memory (DRAM) speeds. On a uniprocessor, the effectiveness of a cache depends primarily on the hit rate, which in turn depends on such factors as cache and working set sizes, the amount of temporal and spatial locality in the reference stream, the degree of associativity in the cache, and the cache replacement policy.

*This work was supported in part by NSF Institutional Infrastructure grant no. CDA-8822724 and ONR research grant no. N00014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

Two additional factors come into play on a multiprocessor. First, we need a *coherence protocol* to ensure that processors do not access stale copies of data that have been modified elsewhere. Coherence is required for correctness, but may reduce the hit rate (by invalidating lines in some caches when they are modified in others), and can increase the cost of both hits and misses, by introducing extra logic into the cache lookup algorithm. Second, because large-scale machines generally distribute physical memory among the nodes of the system, the cost of a cache miss can vary substantially, even without coherence overhead.

Efficient implementation of a coherent shared memory is arguably the most difficult task faced by the designers of large-scale multiprocessors. Minimizing the cost of coherence is the overriding concern. Given a good coherence protocol, however, the placement of data in the distributed main memory may still have a significant impact on performance, because it affects the cost of cache misses. A substantial body of research has addressed the development of good coherence protocols. This paper addresses the page placement problem. We focus our attention on *behavior-driven* OS-level migration of pages among processors. We limit our consideration to the class of machines in which each physical memory address has a fixed physical location (its *home node*), and in which the hardware cache controllers are capable of filling misses from remote locations.

Ideally, the compiler for a parallel language would determine the best location for each datum at each point in time, and would place data accordingly. Compiler technology has not yet advanced to the point where this task is feasible; the current state of the art assumes a user-specified distribution of data among processors (e.g. as in HPF [14]). Moreover, there will always be important programs for which reference patterns cannot be determined at compile time, e.g. because they depend on input data [16]. Even for those applications in which compile-time placement is feasible, it still seems possible that OS-level placement will offer a simpler, acceptable solution.

Our work shows that we can achieve effective page placement with no hardware support other than the standard address translation and page fault mechanisms. We also show that good placement is helpful regardless of whether coherence is maintained in hardware (on a CC-NUMA machine) or in kernel-level software (on a non-coherent NUMA machine). Finally we evaluate dynamic page migration and page replication (with invalidations for coherence) as further mechanisms to improve the performance of coherent shared-memory systems, but observe little or no performance benefit and often a performance loss for our application suite. We speculate that page replication may be useful for programs with large data structures that are very infrequently written.

By way of further introduction, we briefly survey related work in section 2. We then present our algorithms and experimental environment in section 3. We present results in section 4 and conclude in section 5.

2 Related Work

Page migration and replication has also been used on cacheless NUMA multiprocessors in order to take advantage of the lower cost of accessing local memory instead of remote memory [3, 4, 6, 11, 12]. By using efficient block-transfer hardware to transfer page-size blocks, these "NUMA memory management" systems reduce the average cost per reference. This paper addresses the question of whether similar policies are still effective on machines with per-processor caches.

Cache-coherent shared memory multiprocessors fall into two basic categories, termed CC-NUMA (cache coherent, non-uniform memory access) and COMA (cache only memory architecture). CC-NUMA machines are characterized by local per-processor caches, distributed main memory, scalable interconnection networks, and a protocol that maintains cache coherence. Examples of such machines include the Stanford DASH [13] and the MIT Alewife [1]. COMA machines are similar to CC-NUMAs, except that the local portion of main memory is organized as a cache (named attraction memory) and data is replicated to local memory as well as the cache on a miss. Examples of such machines include the commercially available KSR-1 [9] and the Swedish Data Diffusion Machine (DDM) [7].

COMA multiprocessors have a significant advantage over CC-NUMA multiprocessors when it comes to servicing capacity and conflict cache misses. Since the local memory of a node serves as a large secondary or tertiary cache, most such misses are satisfied locally, incurring smaller miss penalties and less interconnection traffic. CC-NUMAs can approach the behavior of COMA machines if data are laid out intelligently in main memory so that most misses are satisfied by a node's local memory. Past work [18] has shown that with additional hardware, or programmer and compiler intervention, data pages can be migrated to the nodes that would miss on them the most, achieving performance comparable to that of COMA machines. The advantages of this approach are its relative hardware simplicity and its lower overhead for data that are actively shared. Our approach is applicable to both NUMA machines with non-coherent caches and CC-NUMA machines, and requires little or no additional hardware.

Chandra *et. al.* have independently studied migration in the context of CC-NUMAs with eager hardware cache coherence [5]. They simulated several migration policies based on counting cache misses and/or TLB misses; some of the policies allowed a page to move only once, and others allowed multiple migrations to occur. One of their policies (single move on the first cache miss) is similar to our dynamic placement policy. They also found that a single-move policy can cause many cache misses to be performed locally, though our results are not directly comparable because we used different applications. We extend their work by considering replication strategies, as well as investigating the effects of placement on both eager (hardware) and lazy (software) coherent systems.

3 Algorithms and Experimental Environment

In this section we describe our simulation testbed, the coherence protocols with which we started, the changes we made to those protocols to implement page placement, and the set of applications on which we evaluated those changes.

3.1 Simulation Methodology

We use execution driven simulation to simulate a mesh-connected multiprocessor with up to 64 nodes. Our simulator consists of two parts: a front end, Mint [19, 20], which simulates the execution of the processors, and a back end that simulates the memory system. The front end calls the back end on every data reference (instruction fetches are assumed to always be cache hits). The back end decides which processors block waiting for memory and which continue execution. Since the decision is made on-line, the back end affects the timing of the front end, so that the control flow of the application,

System Constant Name	Default Value
TLB size	128 entries
TLB fill time	100 cycles
Interrupt (page fault) cost	140 cycles
Page table modification	320 cycles
Memory latency	12 cycles
Memory bandwidth	1 word / 4 cycles
Page size	4K bytes
Total cache per processor	16K bytes
Cache line size	64 bytes
Network path width	16 bits (bidirectional)
Link latency	2 cycles
Routing time	4 cycles
Directory lookup cost	10 cycles
Cache purge time	1 cycle/line
Page move time	approx. 4300 cycles

Table 1: Default values for system parameters, assuming a 100-MHz processor.

and the interleaving of instructions across processors, can depend on the behavior of the memory system.

The front end implements the MIPS II instruction set. Interchangeable modules in the back end allow us to explore the design space of software and hardware coherence. Our hardware-coherent modules are quite detailed, with finite-size caches, write buffers, full protocol emulation, distance-dependent network delays, and memory access costs (including memory contention). Our simulator is capable of capturing contention within the network, but only at a substantial cost in execution time; the results reported here model network contention at the sending and receiving nodes of a message, but not at the intermediate nodes. Our software-coherent modules add a detailed simulation of TLB behavior. To avoid the complexities of instruction-level simulation of interrupt handlers, we assume a constant overhead for page fault interrupt handling. The actual cost of a page fault is the sum of the interrupt, page table, and TLB overheads. Table 1 summarizes the default parameters used in our simulations.

The CC-NUMA machine uses the directory-based write-invalidate coherence protocol of the Stanford DASH machine [13]. This protocol employs an eager implementation of release consistency.

Our software-coherent NUMA machine uses a scalable extension of the work of Petersen and Li [15], with additional ideas from the work of Keleher et al. [8]. It employs a lazy implementation of release consistency, in which invalidation messages are sent only at synchronization *release* points, and processed (locally) only at synchronization *acquire* points.

At an acquire, a processor is required to flush from its own cache all lines of all pages that have been modified by any other processor since the current processor's last acquire. It is also required to unmap the page, so that future accesses will generate a page fault. At a release, a process is required

to write back all dirty words in its cache.[1]

To allow a processor to determine which pages to flush and un-map on an acquire, we maintain a distributed *weak list* of pages for which out-of-date cached copies may exist. When a processor first accesses a page (or accesses it for the first time after un-mapping it), the handler for the resulting page fault adds the page to the processor's page table and communicates with the page's home node to maintain lists of processors with read-write and read-only mappings. If the only previously-existing mapping had read-write permissions, or if the current fault was a write fault and all previously-existing mappings were read-only, then the page is added to the weak list. Full details of this protocol can be found in a technical report [10].

3.2 Page Placement Mechanisms

The changes required to add page placement to both the hardware and software coherence protocols were straightforward. The basic idea is that the first processor to touch a given page of shared memory becomes that page's home node. To deal with the common case in which one processor initializes all of shared memory before parallel computation begins, we created an executable "done with initialization" annotation that programmers can call at the point at which the system should begin to migrate pages. This annotation improves the performance of certain applications which have a single processor initialize shared data structures by preventing large amounts of shared data from migrating to that processor. To deal with the possibility that the pattern of accesses to shared memory might undergo a major change in the middle of execution, we also created a "phase change" annotation that programmers could call when the system should re-evaluate its placement decisions.

At the beginning of execution, shared memory pages are unmapped (this was already true for the software protocol, but not for the hardware one). The first processor to suffer a page fault on a page (or the first one after initialization or a phase change) becomes the page's home node. That processor requests the page from the current home, then blocks until the page arrives.

Ideally, one would want to place a page on the processor that will suffer the most cache misses for that page. Unfortunately, this is not possible without future knowledge, so we place a page based on its past behavior. We simulated a policy, based on extra hardware, in which the first processor to perform n cache fills on a page becomes the page's home node, but found no significant improvement over the "first reference" policy. The first reference policy does not attempt to determine which processor uses a page the most, but does ensure that no processor is home to pages that it does not use.

3.3 Application Suite

Our application suite consists of five programs. Two (`sor` and `mgrid`) are locally-written kernels. The others (`mp3d`, `appbt`, and `water`) are full applications.

`SOR` performs banded red-black successive over-relaxation on a 640×640 grid to calculate the temperature at each point of a flat rectangular panel. We simulated 10 iterations.

[1] Because there may be multiple dirty copies of a given line, non-dirty words must *not* be written back. To distinguish the dirty words, we assume that the cache includes per-word dirty bits.

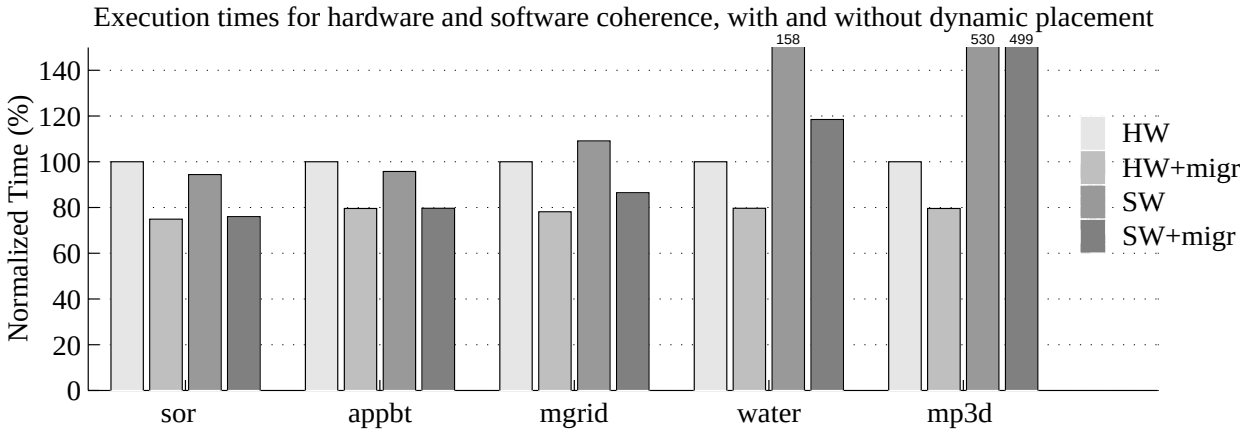

Figure 1: Normalized execution times, for 64 processors and 64-byte cache blocks.

`Mgrid` is a simplified shared-memory version of the multigrid kernel from the NAS Parallel Benchmarks [2]. It performs a more elaborate over-relaxation using multi-grid techniques to compute an approximate solution to the Poisson equation on the unit cube. We simulated 2 iterations, with 5 relaxation steps on each grid, and grid sizes from $64 \times 64 \times 32$ down to $16 \times 16 \times 8$.

`Mp3d` is part of the SPLASH suite [17]. It simulates rarefied fluid flow using a Monte Carlo algorithm. We simulated 20,000 particles for 10 time steps.

`Appbt` is from the NAS Parallel Benchmarks suite. It computes an approximate solution to the Navier-Stokes equations. It was translated to shared memory from the original message-based form by Doug Burger and Sanjay Mehta at the University of Wisconsin. We simulated a $16 \times 16 \times 16$ grid for 5 time steps.

`Water`, also from the SPLASH suite, simulates the evolution of a system of water molecules by numerically solving the Newtonian equations of motion at each time step. We simulated 256 molecules for 5 time steps.

These applications were chosen in order to encompass various common caching and sharing behaviors. The input sizes we chose, although small (due to simulation constraints), deliver reasonable scalability for most of our applications. We deliberately kept the cache sizes small, so that the ratio between cache size and working set size would be about the same as one would expect in a full-size machine and problem. As we will show in the next section, most of the applications exhibit behavior for which dynamic page placement is beneficial.

4 Results

4.1 Dynamic Page Placement

In this section, we show that the "first reference" page placement scheme can result in significant performance improvements in both hardware- and software-coherent systems. Figure 1 shows the execution time for each of the applications in our suite, under each of the coherence systems. The times for each application are normalized so that the hardware-coherent system without dynamic

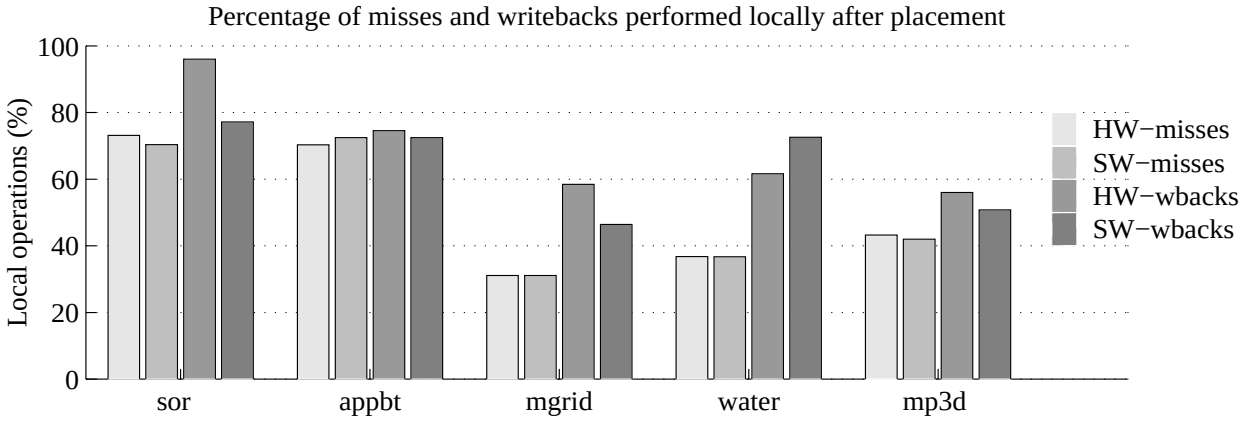

Figure 2: Local cache activity, for 64 processors and 64-byte cache blocks.

placement is at 100%. For most applications, placement improves performance by 20 to 40 percent, by allowing cache misses (and, secondarily, writebacks) to happen locally.

The software and hardware coherence systems generally exhibit comparable performance both with and without migration. Our applications exhibit coarse grained sharing and therefore scale nicely under both coherence schemes. The one exception is `mp3d`, which requires several modifications to work well on a software coherent system [10]. These modifications were not applied to the code in these experiments.

Figure 2 shows the percentage of cache misses and writebacks that occur on pages that are local after migration. Without dynamic placement, the applications in our suite satisfy less than two percent of their misses locally, as would be expected from round-robin placement on 64 processors. Dynamic placement allows 35 to 75 percent of cache misses and 50 to 100 percent of writebacks to be satisfied locally.

Figure 3 shows the average cache fill time for each application under both hardware and software coherence. Dynamic page placement reduces the average fill time by 20 to 40 percent for the hardware coherent system, and 30 to 50 percent for the software coherent system.

`Mgrid` and `sor` are statically block-scheduled, and exhibit pair-wise sharing. They obtain a benefit from dynamic placement even for cache fills and writebacks that are *not* satisfied locally, because neighbors in the block-scheduled code tend to be physically close to one another in the mesh-based interconnection network.

In most cases, the eager hardware-coherent system benefits more from dynamic placement than does the lazy software-coherent system. Our hardware-coherent system sends invalidation messages immediately at the time of a write, and waits for acknowledgments when a lock is released. The software system sends write notices at the time of a release, and invalidates written blocks at the time of an acquire. As a result, the hardware system incurs more misses caused by false sharing, and therefore exhibits a slightly higher miss rate. Thus, any reduction in the average cost of a miss has a greater impact on the hardware system's performance.

Our placement strategy seems to work well for a variety of cache block sizes. The performance gain from dynamic placement generally varies more with block size in the hardware coherent system than it does in the hardware-coherent system.

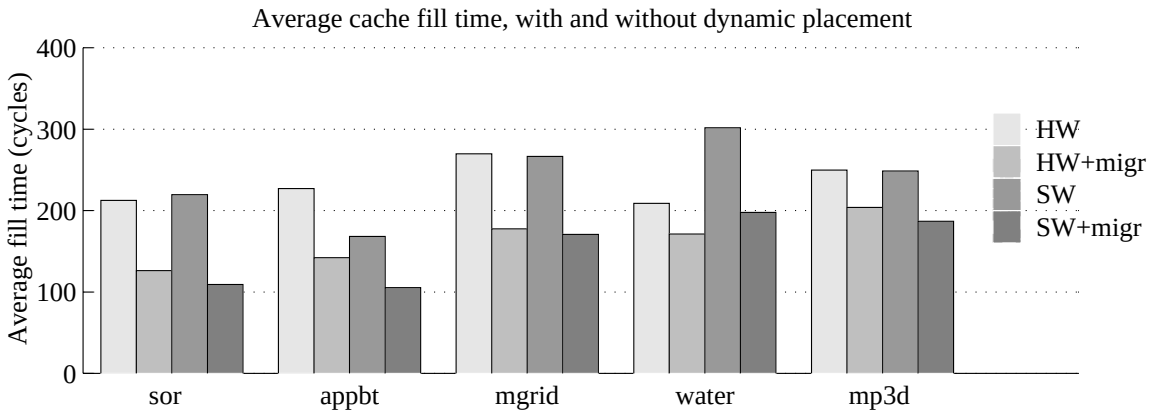

Figure 3: Average fill time, for 64 processors and 64-byte cache blocks.

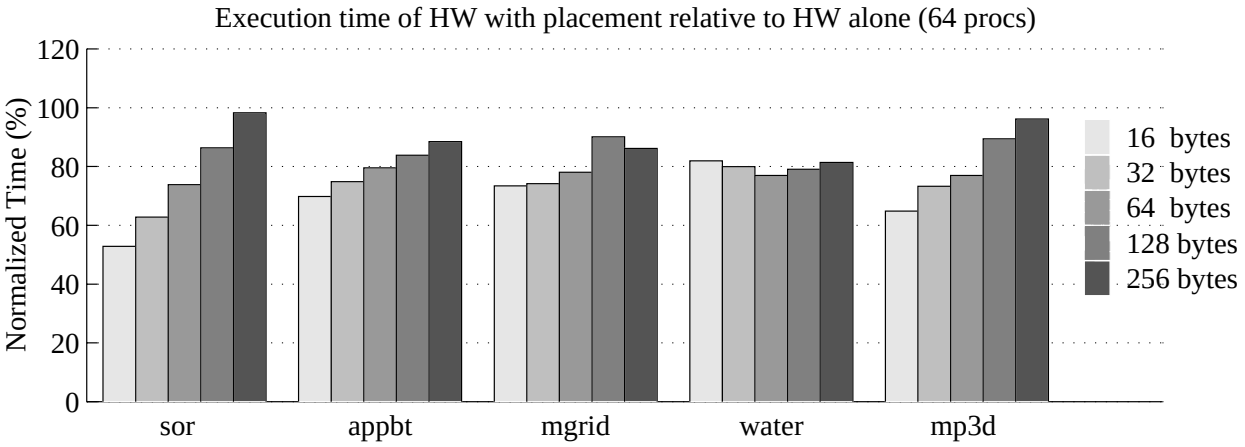

Figure 4: Normalized execution times for varying block sizes under hardware coherence.

Figures 4 and 5 show the performance of the hardware and software-coherent systems for block sizes ranging from 16 to 256 bytes. Each bar represents the execution time of an application for a particular block size; the height of the bar is the execution time with dynamic placement relative to the execution time without it for the same block size. For example, under both coherence systems, dynamic page placement provides more performance gain for `sor` when the cache blocks are small. For programs with good spatial locality, such as `sor` and `water`, increasing the block size decreases the miss rate, reducing the performance gain.

For small block sizes, cold-start misses are significant, as are evictions if the working set size is greater than the cache size. Dynamic placement speeds up cold-start misses by making one block transfer over the network and then performing the misses locally. Eviction misses always access blocks that were previously accessed; if the page containing those blocks is moved to the local memory, the misses can be serviced significantly faster. This is most effective if the local processor will perform more cache fills on the page than any other processor.

Large cache blocks amortize the latency of a miss over a large amount of data, but are more likely to suffer from false sharing and evictions. For programs with good spatial locality, fetching

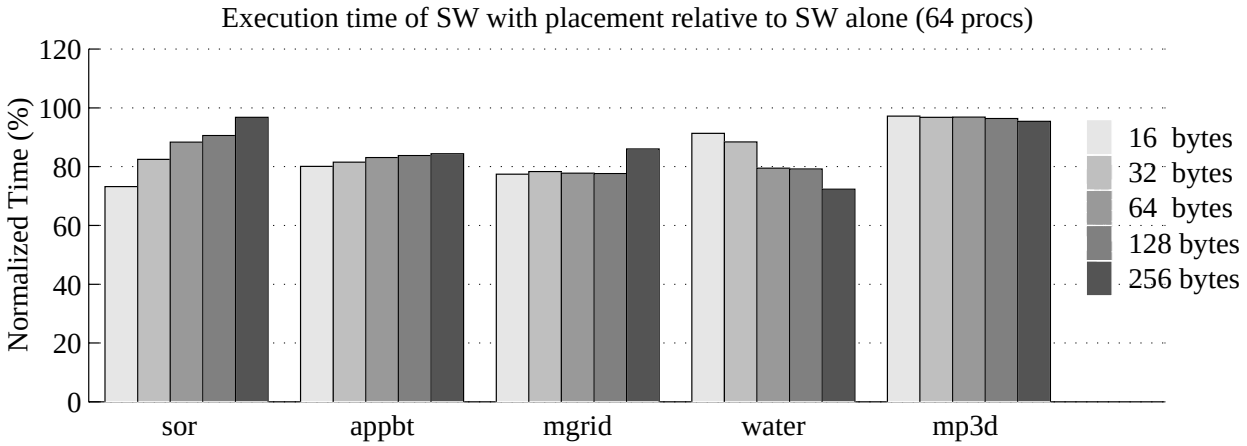

Figure 5: Normalized execution times for varying block sizes under software coherence.

large blocks reduces the miss rate but increases the cost of a miss. The miss rate is the dominant effect, making large cache blocks a net win, but the increased cost of misses mitigates this to some extent, so dynamic placement remains worthwhile.

4.2 Dynamic Migration and Replication

Though dynamic placement provides a significant performance gain for many applications, it seemed likely that the reference behavior of some programs may vary significantly during execution. Therefore we provided an executable "phase change" annotation which indicates to the operating system or runtime that the program behavior has changed. In our simulations, the runtime system uses this as a signal to discard all placement decisions and allow the pages to migrate to another processor.

Most of our applications do not have well-defined phase changes. The exception is `mgrid`, because its access pattern changes as the grid size changes. Adding the phase change annotation was simple, involving only two lines of code. However, dynamic migration did not improve the performance of `mgrid`; in fact, it reduced the performance by 13 percent. This is due to the fact that in `mgrid`, each phase uses eight times as much data as the previous (smaller) phase. Therefore data locality is primarily determined by the last phase. The cost of migrating pages to local memory for the smaller phases, and migrating them again for larger phases, exceeds the cost of performing remote cache fills for the smaller phases.

We have also investigated several policies for replicating pages of data. These are:

- **Time policy:** if a page remains mapped for n cycles, copy it to local memory the next time it is mapped.
- **Counter policy:** if n cache fills are performed on a page before it is unmapped, copy it to local memory copy the next time it is mapped. This requires some hardware support.
- **Counter-interrupt policy:** if n cache fills have been performed on a page since it was mapped, copy it to local memory immediately. This also requires hardware support.

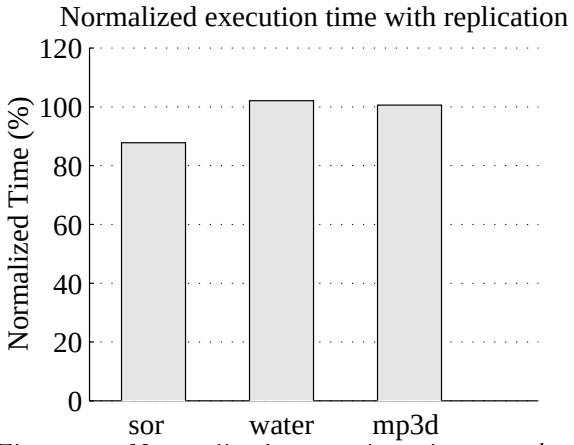

Figure 6: Normalized execution times under software coherence with page replication.

For our simulations, we selected several applications which we believed would be most likely to benefit from replication. For these applications, the policy which performed best was the counter policy. Figure 6 shows the relative performance of our applications with page replication. `SOR` is the only program for which we found a significant performance gain from replication (13%).

We believe that the failure of replication is a result of the sharing patterns exhibited by our applications. In particular, many replicated pages tended to be accessed very little before being written again by another causally-related processor, invalidating the copy. Even assuming high network and memory bandwidths (1 word per cycle), the high cost of replicating those pages caused performance degradation. Additionally, the reference patterns of some applications may contain frequent writes, which will not allow very many pages to be replicated. Replication may still be useful if it is limited to data structures that are mostly read, such as lookup tables written only during initialization. We are considering the use of program annotations to identify such data.

5 Conclusions

We have studied the performance impact of simple behavior-driven page placement policies under both hardware and software cache coherence. We find that for applications whose working sets do not fit entirely in cache, dynamic page placement provides substantial performance benefits, by allowing capacity misses to be serviced from local memory, thus incurring reduced miss penalties. We have also shown that a very simple policy suffices to achieve good results and that complicated hardware is not required in devising an effective page placement strategy. Finally we have investigated the performance impact of dynamic page migration and page replication on cache coherent multiprocessors but found no performance benefits for our application suite. We believe that the reference pattern favoring replication is uncommon in scientific applications, and that dynamic placement suffices to improve the miss penalties of the applications that run on these machines.

References

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. Directory-Based Cache Coherence in Large-Scale Multiprocessors. In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1992.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Report RNR-91-002, NASA Ames Research Center, January 1991.

[3] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, Litchfield Park, AZ, December 1989.

[4] 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 *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, Santa Clara, CA, April 1991.

[5] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. An Evaluation of OS Scheduling and Page Migration for CC-NUMA Multiprocessors. In *Fourth Workshop on Scalable Shared Memory Multiprocessors*, Chicago, IL, April 1994. Held in conjunction with ISCA '94.

[6] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32–44, Litchfield Park, AZ, December 1989.

[7] E. Hagersten, A. Landin, and S. Haridi. DDM—A Cache-Only Memory Architecture. *Computer*, 25(9):44–54, September 1992.

[8] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[9] Kendall Square Research. KSR1 Principles of Operation. Waltham MA, 1992.

[10] L. I. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. TR 513, Computer Science Department, University of Rochester, March 1994. Submitted for publication.

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

[12] R. P. LaRowe Jr., M. A. Holliday, and C. S. Ellis. An Analysis of Dynamic Page Placement on a NUMA Multiprocessor. In *Proceedings of the 1992 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Newport, RI, June 1992.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[14] D. B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, February 1993.

[15] K. Petersen and K. Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. In *Proceedings of the Seventh International Parallel Processing Symposium*, Newport Beach, CA, April 1993.

[16] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[17] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, March 1992.

[18] P. Stenström, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 80–91, Gold Coast, Australia, May 1992.

[19] J. E. Veenstra. Mint Tutorial and User Manual. TR 452, Computer Science Department, University of Rochester, July 1993.

[20] J. E. Veenstra and R. J. Fowler. Mint: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201–207, Durham, NC, January−February 1994.