Page Locality Management for CC-NUMA Multiprocessors *

Matthew Frank
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
mfrank@lcs.mit.edu

Page locality management is a technique for using the virtual memory system of a shared memory multiprocessor to insure that cache misses will be satisfied by the memory module closest to each processor. We present a model of cache behavior that demonstrates that the costs and benefits of managing locality are heavily dependent on application characteristics and argue that since any particular locality policy can deliver improved performance to only a limited set of programs, applications must be allowed to implement policies specific to their own needs. In addition we show results from a prototype page locality manager on the MIT Alewife shared memory multiprocessor that validate our model and demonstrate the utility of flexible locality policies.

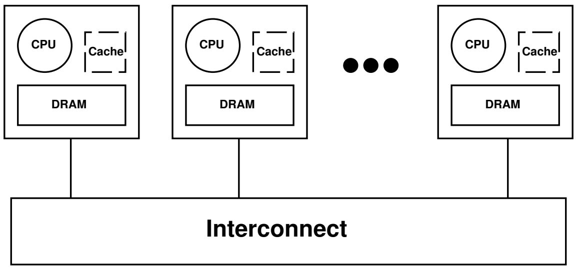

Figure 1: Architecture of a CC-NUMA Multiprocessor

1 Page Locality Management

A cache coherent non-uniform memory access (CC-NUMA) multiprocessor [1, 7] consists of a set of interconnected nodes, each with a processor, a small, high speed cache, and a portion of physical memory. The machine attempts to provide an efficient shared memory abstraction by automatically managing coherence between the caches. Applications that suffer a high cache miss rate, either because their working set is larger than the cache or because sharing problems cause repeated coherence messages, may observe large cache miss idle times if they need to fetch data from remote memory modules. A page locality manager attempts to reduce the total cache miss penalty by using the machine's virtual memory manager to move data into the memory modules collocated with the nodes that require that data.

Previous work in page locality management has concentrated on hiding any locality decisions from the application. Several research page locality managers have been implemented for NUMAs without caches [4, 6] or CC-NUMAs [3]. Each of these systems provided a single locality policy in the kernel. Their designers hoped that the heuristics they provided would be appropriate across a wide and general class of applications.

In fact, an end-to-end argument [9, 5] suggests that much of the work done by these implementations either duplicates work already being done by the application or is inappropriate to the needs of the application [2, 8]. An application has different needs for each of its internal data structures. For example, code pages can be fully replicated while stack pages should migrate with the thread that uses them. Applications with small working sets may not benefit at all from page replication because their cache miss rates are so low, while compilers for large static applications may be able to predetermine an optimal page placement. We model data movement costs and show that different reference patterns call for radically different policies.

* Matt Frank is advised by Anant Agarwal and Frans Kaashoek and is supported by an NSF Graduate Research Fellowship. The Alewife project is funded in part by ARPA contract # N00014-94-1-09885 and in part by NSF grant # MIP-9504399.

2 Reducing Capacity Miss Cost

We can model the tradeoffs between the cost of copying a page and the cost of servicing cache misses remotely by extending Cox and Fowler's model for PLATINUM [4]. We will assume that the number of cache misses, m, to a particular page is independent of page placement. Then it pays to copy a page to the local memory if $mr > ml + c_o + c_b s_p$ where r is the cost of a remote access, l is the cost of a local access, c_o is the cost of setting up a page transfer, c_b is the application observed cost of copying each byte of the page and s_p is the size of the page in bytes.

The number of misses per page, m, is dependent on page size, s_p, cache line size, s_c, and most importantly, application characteristics. We define an applications reference density to a particular page as $\rho = m/(s_p/s_c) = ms_c/s_p$ such that an application which misses to each cache line on a page exactly once will have a density of 1. The model can then be rephrased in terms of reference density as $\rho r > \rho l + c_o s_c/s_p + c_b s_c$. On Alewife [1] r is approximately 30 cycles for a remote access to an adjacent node with no contention and l is approximately 10 cycles, s_c is 16 bytes and s_p is 4096 bytes. Setting up a page transfer, c_o, requires approximately 2000 cycles while c_b is approximately 10/16 modulo interference between the processor and DMA device at the memory controller. Given these parameters a reference density of $\rho > 0.9$ will make copying a page effective. That the required reference density is below one, indicates that just the cost of cold cache misses makes it worth while to create local pages for densely referenced structures like the code and stack segments. For less densely referenced pages the correct strategy depends on application specific characteristics like working set size and data layout.

3 Reducing Coherence Overhead

In addition, page locality management can help alleviate "ping-pong" problems caused by the mismatch between the hardware invalidation based cache coherence protocol and application requirements. Here we focus only on producer-consumer sharing

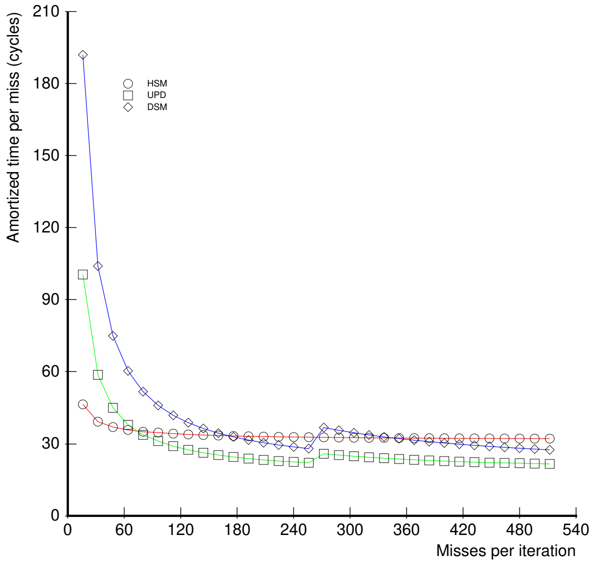

Figure 2: Performance of three page locality policies

commonly found in scientific applications. In this sharing pattern each read operation will miss in the cache, while each write operation is forced to incur the penalty of invalidating all the read copies. A page based update protocol, on the other hand, permits the application to incur only a single penalty for sending the data from the producer to each consumer.

As above, we will use s_p to denote the page size, s_c to denote the cache line length, l as the cost of satisfying a miss locally and r of satisfying a miss remotely or invalidating a remote cache, and c_o and c_b as the overhead and per-byte cost of moving a page. Then using the hardware protocol the cost for the producer to write a cache line is r and the cost for each consumer to fetch the cache line is also r. The total cost per iteration is then given by $(s_p/s_c)2r$.

If, instead, the producer sends a copy of the page to the consumer on each iteration the total cost per iteration is given by both the cost of accessing each cache line locally and the cost of sending the page. The cost of accessing the page locally will be $(s_p/s_c)l$ for both the producer and consumer. The cost of sending the page will be $c_o + c_b s_p$. Page replication will then improve performance if $(s_p/s_c)2r > (s_p/s_c)2l + c_o + c_b s_p$. For the numbers given above, this implies that as long as at least $1/4$ of the cache lines on the page are referenced on each iteration, then the page based update protocol will be preferable. For applications with finer grain communication, the hardware invalidation policy will offer better performance and pages should not be migrated.

4 A Prototype Page Locality Manager

We have implemented a flexible page locality manager for the MIT Alewife shared memory multiprocessor which we call ASP (Application Specific Page manager). ASP provides an interface that an application can use to implement a specific page locality policy for each region of memory. Figure 2 shows performance measurments for a simple producer-consumer benchmark using three different locality policies. In this benchmark the producer writes an array and then after a synchronization point the consumer reads the array. We have varied the size of the array being shared between 16 bytes (1 cache line) and 8192 bytes (512 cache lines). We examine three different locality policies. In the static policy (labeled HSM on the graph), the data pages are statically mapped to the memory on the producer node. The DSM policy is representative of the invalidation policies typically used in software shared memory systems. Finally the UPD policy is a page-based update policy specifically tailored to coarse grain producer-consumer sharing.

For small array sizes the high cost of initiating paged-based actions in the two software schemes (UPD and DSM) cannot be amortized, and the static scheme (HSM) is dramatically superior. For larger array sizes the overall reduction in cost for individual misses outweighs the software costs. The large software overheads are amortized across a large number of misses, and so the UPD scheme exhibits approximately a 25% performance improvement. Note that the software overhead increases in "jumps" each time the array becomes large enough to consume another page. This is the reason for the increase in cost between 256 and 257 cache misses, a page consists of 256 cache lines.

The results of this simple micro-benchmark validate our simple model. As predicted in section 3, a page based update policy becomes effective when more than about $1/4$ (64 out of 256) of the cache lines on a page are referenced on each iteration. In addition, we believe the ASP prototype demonstrates the need for page locality managers with flexible policies that can be tailored to application needs. On going work on the ASP locality manager involves identifying compiler algorithms to automatically determine the best policy for each application data structure.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[3] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–24, October 1994.

[4] Alan L. Cox and Robert 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, December 1989.

[5] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[6] Richard P. LaRowe, Jr. and Carla Schlatter Ellis. Page placement policies for NUMA multiprocessors. *Journal of Parallel and Distributed Computing*, 11(2):112–129, February 1991.

[7] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[8] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[9] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.