[17] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[18] J. Stone and A. Norton. The VM/EPEX FORTRAN Preprocessor Reference. Research Report RC-11408, IBM Research Division, 1985.

[19] A. Tevanian et al. Personal communication. Comments on the inappropriateness of NUMA machines.

[20] A. W. Wilson. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proc. 14th Annual International Symposium on Computer Architecture*, pages 244–252, June 1987.

[21] M. Young et al. Personal communication. Comments on changes to Mach fault handling in versions that support external pagers.

[22] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proc. 11th Symposium on Operating Systems Principles*, pages 63–76, November 1987.

False sharing is an accident of colocating data objects with different reference characteristics in the same virtual page, and is thus beyond the scope of operating system techniques based on placement of virtual pages. Because shared pages are placed in global memory, false sharing causes memory references to be made to slower global memory instead of faster local memory. We found that false sharing could be reduced, often dramatically, by tuning application code. Additional work is needed at the language processor level to make it easier to reduce this source of performance degradation.

Acknowledgements

We are greatly indebted to Dan Poff for his tremendous help in making Mach run on the ACE, to Armando Garcia and David Foster for building the ACE and making it work as well as it does, to Bob Marinelli for early help in bring up the ACE, to the entire Mach crew at CMU for stimulating and lively discussion regarding the relationship of Mach, the pmap interface and NUMA machines, and to Felicia Ferlin, Dave Redell and the referees for helping to improve the presentation of this paper.

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 Conference*, July 1986.

[2] O. Babaoglu and W. Joy. Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits. In *Proc. 8th Symposium on Operating Systems Principles*, pages 78–86, December 1981.

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

[4] B. Beck and D. Olien. A Parallel Programming Process Model. In *Proc. Winter 1987 USENIX Conference*, pages 83–102, January 1987.

[5] N. Carriero and D. Gelernter. Applications Experience with Linda. In *Proc. PPEALS '88—Parallel Programming: Experience with Applications, Languages and Systems*, pages 173–187, July 1988.

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

[7] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proc. 12th Symposium on Operating Systems Principles*, December 1989.

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

[9] A. Garcia, D. Foster, and R. Freitas. The Advanced Computing Environment Multiprocessor Workstation. Research Report RC-14419, IBM Research Division, March 1989.

[10] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 422–431, May 1988.

[11] M. A. Holliday. Reference History, Page Size, and Migration Daemons in Local/Remote Architectures. In *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[12] IBM. *IBM RT/PC Hardware Technical Reference*, 1988. Part Numbers SA23-2610-0, SA23-2611-0 and SA23-2612-0, Third Edition.

[13] R. P. LaRowe and C. S. Ellis. Virtual Page Placement Policies for NUMA Multiprocessors. Technical report, Department of Computer Science, Duke University, December 1988.

[14] T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Memory Management for Large-Scale NUMA Multiprocessors. Technical report, Computer Science Department, University of Rochester, 1989.

[15] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proc. 5th Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[16] H. E. Mizrahi, J.-L. Baer, E. D. Lazowska, and J. Zahorjan. Introducing Memory into the Switch Elements of Multiprocessor Interconnection Networks. In *Proc. 16th Annual International Symposium on Computer Architecture*, pages 158–166, May 1989.

sors, the state in TLBs and instruction and data caches can make affinity desirable.

The scheduler that came with our version of Mach had little support for processor affinity. There was conceptually a single queue of runnable processes, from which available processors selected the next process to run. On the ACE this resulted in processes moving between processors far too often. We therefore modified the Mach scheduler to bind each newly created process to a processor on which it executed everything except Unix system calls. We assigned processors sequentially by processor number, skipping processors that were busy at the time unless all processors were busy. This approach proved adequate for our toy benchmarks and for a Unix-like environment with short-lived processes, but it is not a real solution. For load balancing in the presence of longer-lived compute-bound applications, we will need to migrate processes to new homes and move their local pages with them.

5 Future Work

The study of parallel programming on NUMA machines is still in its infancy. We have already mentioned several areas we think deserve further investigation. Chief among these is false sharing (Section 4.2) and what language processors can do to automate its reduction. The study of parallel applications on NUMA machines should continue. We need experience with a much wider application base than we have at present. Trace-driven analyses can provide much more detailed understanding than what we could garner through the processor-time based approach described in Section 3. Processor scheduling on NUMA machines (Section 4.7) remains to be explored, as does the value of having applications provide placement pragmas (Sections 4.3 and 4.4) to improve placement or to reduce automatic placement overhead.

The comparison of alternative policies for NUMA page placement is an active topic of current research[7, 11, 13]. It is tempting to consider ever more complex policies, but our work suggests that a simple policy can work extremely well. We doubt that substantial additional gains can be obtained in the operating system. It may in some applications be worthwhile periodically to reconsider the decision to pin a page in global memory. It may also be worth designing a virtual memory system that integrates page placement more closely with pagein and pageout, or that attempts to achieve the simplicity of our cache approach without requiring that all pages be backed by global memory or that local memory be used only as a cache[14].

The operating system itself is a parallel application worthy of investigation. We have not attempted to place any kernel data structures in local memory other than those required to be there by the hardware. Increasing the autonomy of the kernel from processor to processor and making private any kernel data that need not be shared should both improve performance on small multiprocessors and improve scalability to large ones.

6 Summary and Conclusions

We have explored what a virtual memory paging subsystem can do about the problem of data placement in multiprocessors with non-uniform memory access (NUMA) times. Our approach was:

- to simplify the application program view of memory by using automatic placement of virtual pages to hide the NUMA memory characteristics,
- to consider only a simple, two-level (local and global) memory hierarchy, even if the actual NUMA memory system is more complex, and
- to use a simple cache-like strategy for placing and replicating pages.

Our placement strategy was to replicate read-only pages, including those that could have been written but never actually were. Written pages were placed in local memory if only one processor accessed them or in global memory if more than one processor did.

We tested our approach in a version of the Mach operating system running on the IBM ACE multiprocessor workstation. We found:

- that our simple page placement strategy worked about as well as any operating system level strategy could have,
- that this strategy could be implemented easily within the Mach machine-dependent pmap layer, and
- that the dominant remaining source of avoidable performance degradation was *false sharing*, which could be reduced by improving language processors or by tuning applications.

We found our automatic page placement to be an adequate tool for coping with a NUMA memory system. It presented applications with a simple view of virtual memory that was not much harder to program than the flat shared memory of a traditional UMA multiprocessor. Correct application programs ran correctly and could then be tuned to improve performance. Our placement strategy was easy to predict, it put pages in appropriate places, and it ran at acceptable cost.

Remote references permit shared data to be placed closer to one processor than to another, and raise the issue of deciding which location is best. Unfortunately, we see no reasonable way of determining this location without pragmas or special-purpose hardware. Conventional memory-management systems provide no way to measure the relative frequencies of references from processors to pages. Tricks such as those of the Unix pageout daemon[2], detect only the presence or absence of references, not their frequency.

Without frequency of reference information, considering only a single class of physical shared memory is both a reasonable approach and a major simplification. On machines without physically global memory, every page will need to be local to some processor, and the lack of frequency of reference information will be more of a problem than it is on machines like the ACE. In practice we expect that machines with only local memory will rely on pragmas for page location, or accept the burden of mis-located pages.

If desired, our pmap manager could accommodate both global and remote references with minimal modification. The necessary cache transition rules are a straightforward extension of the algorithm presented in Section 2. With appropriate pragmas from the application, it might be worthwhile to consider this extension, though it is not clear whether applications actually display reference patterns lopsided enough to make remote references profitable. Remote memory is likely to be significantly slower than global memory on most machines.

It is hard to predict what sorts of memory architectures will appear on future multiprocessors, beyond the fact that they will display non-uniform access times. Individual processors will probably have caches on the path to local memory. It may be difficult to route remote references through these caches, so it may not be possible for a writable shared page to reside in the local memory of some processor. However, even if references to another processor's local memory are disallowed, it is entirely possible that the non-processor-local memory will itself be non-uniform[16], and so the problem of determining relative reference rates and thereby appropriate locations will remain.

4.5 Comparison to Hardware-Based Caches

It is not yet clear whether the NUMA problem is best attacked in hardware (with consistent caches) or in software. It may be possible to construct machines such as the Encore Ultramax[20] or Wisconsin Multicube[10] with hundreds or even thousands of processors. If these machines are successful they will avoid the overhead of software-based solutions and may also reduce the impact of false sharing by performing their migration and replication at a granularity (the cache line) significantly finer than the page. On the other hand, machines without hardware cache consistency are likely to be substantially cheaper to build, and with reasonable software (to move pages and to reduce false sharing) may work about as well.

4.6 Mach as a Base for NUMA Systems

Aside from the problem of the fixed-sized logical page pool, which forces a fixed degree of replication, Mach has supported our effort very well. It allowed us to construct a working operating system for a novel new machine in a very short period of time. The number of changes necessary in the system was surprisingly small, considering that no one was attempting to solve the NUMA problem at the time the pmap interface was designed. Of particular use was the ability to drop mappings or tighten protections essentially at whim; this relieved us of determining which faults were caused by our NUMA-related tightening of protections and which were caused by other things.

One problem that we had with our version of Mach is that much of the Unix compatibility code is still in the kernel. The CMU Mach project intends to remove this code from the kernel and parallelize it at the same time. At present, however, Mach implements the portions of Unix that remain in the kernel by forcing them to run on a single processor, called the "Unix Master." This causes two problems, one for all multiprocessors and one only for NUMA systems. The first is that execution of system calls can produce a bottleneck on the master processor. The second is that some of these system calls reference user memory while running on the master processor. It would be difficult and probably unwise to treat these references differently from all the others. Thus pages that are used only by one process (stacks for example) but that are referenced by Unix system calls can be shared writably with the master processor and can end up in global memory. To ease this problem, we identified several of the worst offending system calls (sigvec, fstat and ioctl) and made *ad hoc* changes to eliminate their references to user memory from the master processor.

4.7 Scheduling for Processor Affinity

Schedulers on NUMA machines should almost certainly maintain an affinity between processors and the processes that run on them because of the large amounts of process state that reside near the processors. Even on traditional UMA (uniform memory access) multiproces-

that is not writably shared, but that is on a writably shared page is *falsely shared*.

In a NUMA machine where writably shared pages are placed in slower global memory, accesses to falsely shared objects are slower than they would be if the objects had been placed in faster local memory. This performance degradation makes false sharing a problem.

Many applications put objects on pages with little regard for the threads that will access the objects. The resulting false sharing places an inherent limit on the extent to which the "NUMA problem" can be solved in an operating system, because falsely shared objects cannot all be placed nearest the processor that is using them.

Our efforts to reduce false sharing in specific applications were manual and clumsy but effective. We forced proximity of objects by merging them into a single large object. We forced separation by adding page-sized padding around objects. We separately coalesced cacheable and non-cacheable objects and padded around them, so that they would be placed in local or global memory as appropriate. We created new private objects to hold data that would otherwise be falsely shared. By eliminating false sharing, our changes improved the values of T_{numa} and γ, and also brought $T_{optimal}$ closer to T_{local}, by decreasing the amount of data that was "shared." These changes did not significantly change the ratio of T_{numa} to $T_{optimal}$.

As an example, consider the primes2 program, which tested for new primes by dividing by those previously found. An initial version of the program segregated most private and shared data, but used the output vector of previously found primes as divisors for new candidates. The output vector was shared because it was written by any processor that found a new prime, yet the potential divisors never changed once found. By modifying the program so that each processor copied the divisors it needed from the shared output vector into a private vector, the value of α (fraction of local references) was increased from 0.66 to 1.00.

Not all false sharing is explicit in application source code; a significant amount is created implicitly by language processors. Compilers create unnamed data, such as tables of addresses and other constants, that we cannot control. Loaders arrange data segments without regard to what objects are near to and far from each other.

We believe, and our experience to date confirms, that simple techniques for automatic page placement can achieve most of the benefit attainable by the operating system and can be a useful tool in coping with a NUMA multiprocessor. We have found that performance can be further improved by reducing false sharing manually. We expect that language processor level solutions to the false sharing problem can significantly reduce the amount of intervention necessary by the application programmer.

4.3 Making Placement Decisions

The two most important goals of automatic page placement are to place pages that are private and those that can be replicated in local memory, and pages that are writably shared in global memory. Our automatic placement strategy was reasonably successful at both.

There is a fundamental problem, however, with locality decisions based on reference behavior: it is hard to make a good decision quickly. A placement strategy should avoid pinning a page in global memory on the basis of transient behavior. On the other hand, it should avoid moving a page repeatedly from one local memory to another before realizing that it should be pinned. The cost of deciding that a page belongs in global memory also suggests that the decision to put it there should not be reconsidered very often[4].

Any locality management system implemented solely in the operating system must suffer some thrashing of writably shared pages between local memories. This cost is probably acceptable for a limited number of small data objects, but may be objectionable when a significant number of pages is involved. For data that are known to be writably shared (or that used to be writably shared but can now be replicated), thrashing overhead may be reduced by providing placement pragmas to application programs. We have considered pragmas that would cause a region of virtual memory to be marked cacheable and placed in local memory or marked non-cacheable and placed in global memory. We have not yet implemented such pragmas, but it would be easy to do so.

4.4 Remote References

Remote memory references are those made by one processor to the local memory of another processor. Such references are supported in several existing NUMA machines, including the BBN Butterfly and IBM RP3, as well as the ACE. On the ACE, remote references may be appropriate for data used frequently by one processor and infrequently by others. On the Butterfly and RP3, all memory belongs to some processor, so remote references provide the only means of actually sharing data.

[4] In fact, our system never reconsiders a pinning decision (unless the pinned page is paged out and back in). Our sample applications showed no cases in which reconsideration would have led to a significant improvement in performance, but one can imagine situations in which it would.

Application	S_{numa}	S_{global}	ΔS	T_{numa}	$\Delta S/T_{numa}$
IMatMult	4.5	1.2	3.3	82.1	4.0%
Primes1	1.4	2.3	na	17413.9	0%
Primes2	29.9	8.5	21.4	4972.9	0.4%
Primes3	11.2	1.9	9.3	37.4	24.9%
FFT	21.1	10.0	11.1	449.0	2.5%

Table 4: Total system time for runs on 7 processors.

that about 95% of its data references were to private memory[3]. Although there are differences in compilers and runtime support, we think that this supports our contention that our NUMA strategy has placed pages effectively and that the remaining global memory references are due to the algorithm in the application program.

PlyTrace[8] is a floating-point intensive C-threads program for rendering artificial images in which surfaces are approximated by polygons. One of its phases is parallelized by using as a work pile its queue of lists of polygons to be rendered.

Overall, our α and γ values were remarkably good, the exceptions being the Gfetch program, which was designed to be terrible, and the Primes3 program, which makes heavy legitimate use of writably shared memory. We see little hope that programs with such heavy sharing can be made to perform better on NUMA machines without restructuring to reduce their use of writably shared data.

3.3 Page Placement Overhead

The results in Section 3.2 reflect only the time spent by the applications in user state, and not the time the NUMA manager uses for page movement and bookkeeping overhead. Table 4 shows the difference in system time between the all global and NUMA managed cases. This difference is interesting because system time includes not only page movement, but also system call time and other unrelated overhead; since the all global case moves no pages, essentially no time is spent on NUMA management, while the system call and other overheads stay the same. Comparing this difference with the NUMA managed user times in Table 3 shows that for all but Primes3 the overhead was small. Primes3 suffers from having a large amount of memory almost all of which winds up being placed in global memory, but which is first copied from local memory to local memory several times. Since the sieve runs quickly, this memory is allocated rapidly relative to the amount of user time user, resulting in a high system/user time ratio.

We have put little effort into streamlining our NUMA management code. We were more interested in determining what constituted a good placement strategy than in how best to code that strategy. Streamlining should greatly reduce system time, as would fast page-copying hardware.

4 Discussion

4.1 The Two-Level NUMA Model

Supporting only a single class of shared physical memory (global memory) was our most important simplification of the NUMA management problem. We chose to use a two-level NUMA memory model (with only local and global memory) because there was an obvious correspondence between it and a simple application memory model that had only private and shared virtual memory. We saw a simple way for automatic page placement to put pages that were private or that could be replicated in local memory and to put pages that were shared in global memory. We hoped that such automatic placement would be a useful tool, making it easier to get parallel programs running, with subsequent tuning to improve performance possible when necessary.

Our experience with Mach on the ACE has been heartening in this respect. We found that we were largely able to ignore the quirks of the memory system and spent most of our energy while writing applications in debugging application errors, often in synchronization and concurrency control. Automatic page placement worked well enough and predictably enough that we could often ignore it and could make it do what we wanted when we cared.

4.2 The Impact of False Sharing

By definition, an object is *writably shared* if it is written by at least one processor and read or written by more than one. Similarly, a virtual page is writably shared if at least one processor writes it and more than one processor reads or writes it. By definition, an object

Application	T_{global}	T_{numa}	T_{local}	α	β	γ
ParMult	67.4	67.4	67.3	na	.00	1.00
Gfetch[3]	60.2	60.2	26.5	0	1.0	2.27
IMatMult[3]	82.1	69.0	68.2	.94	.26	1.01
Primes1	18502.2	17413.9	17413.3	1.0	.06	1.00
Primes2	5754.3	4972.9	4968.9	.99	.16	1.00
Primes3	39.1	37.4	28.8	.17	.36	1.30
FFT	687.4	449.0	438.4	.96	.56	1.02
PlyTrace	56.9	38.8	38.0	.96	.50	1.02

Table 3: Measured user times in seconds and computed model parameters.

3.2 The Application Programs

Our application mix consists of a fast Fourier transform (FFT), a graphics rendering program (PlyTrace), three prime finders (Primes1-3) and an integer matrix multiplier (IMatMult), as well as a program designed to spend all of its time referencing shared memory (Gfetch) and one designed not to reference shared memory at all (ParMult). FFT is an EPEX FORTRAN application, while the other applications are written using the Mach C-Threads package.

The Mach C-Threads package[6] provides a parallel programming environment with a single, uniform memory. All data is implicitly shared; truly private and truly shared data may be indiscriminately interspersed in the program load image by the compiler and loader. Any desired segregation of private and shared data must be induced by hand, by padding data structures out to page boundaries.

EPEX FORTRAN[18] provides a parallel programming environment with private and shared memory. Variables are implicitly private unless explicitly tagged "shared." Shared data is automatically gathered together and separated from private data by the EPEX preprocessor.

ParMult and Gfetch are at the extremes of the spectrum of memory reference behavior. The ParMult program does nothing but integer multiplication. Its only data references are for workload allocation and are too infrequent to be visible through measurement error. Its β is thus 0 and its α irrelevant. The Gfetch program does nothing but fetch from shared virtual memory. Loop control and workload allocation costs are too small to be seen. Its β is thus 1 and its α 0.

The IMatMult program computes the product of a pair of 200x200 integer matrices. Workload allocation parcels out elements of the output matrix, which is found to be shared and is placed in global memory. Once initialized, the input matrices are only read, and are thus replicated in local memory. This program emphasizes the value of replicating data that is writable, but that is never written. The high α reflects the 400 local fetches per global store (the memory references required for the dot product resulting in an output element), while the low β reflects the high cost of integer multiplication on the ACE. Had we multiplied floating point matrices, the even higher cost of floating multiplication would have overwhelmed the data reference costs.

The three primes programs use different parallel approaches to finding the prime numbers between 1 and 10,000,000. Primes1[4] determines if an odd number is prime by dividing it by all odd numbers less than its square root and checking for remainders. It computes heavily (division is expensive on the ACE) and most of its memory references are to the stack during subroutine linkage. Primes2[5] divides each prime candidate by all previously found primes less than its square root. Each thread keeps a private list of primes to be used as divisors, so virtually all data references are local. It also computes heavily, but makes local data references to fetch potential divisors.

The primes3 algorithm is a variant of the Sieve of Eratosthenes, with the sieve represented as a bit vector of odd numbers in shared memory. It produces an integer vector of results by masking off composites in the bit vector and scanning for the remaining primes. It references the shared bit vector heavily, fetching and storing as it masks off bits representing composite numbers. It also computes heavily while scanning the bit vector for primes.

The FFT program, which does a fast Fourier transform of a 256 by 256 array of floating point numbers, was parallelized using the EPEX FORTRAN preprocessor. In an independent study, Baylor and Rathi analyzed reference traces from an EPEX fft program and found

[3]Since Gfetch and IMatMult do almost all fetches and no stores, their computations were done using 2.3 for G/L. The other applications used G/L as 2 to reflect a reasonable balance of loads and stores.

to T_{local} instead. T_{local} is less than $T_{optimal}$ because references to shared data in global memory cannot be made at local memory speeds. T_{local} thus cannot actually be achieved on more than one processor. We measured T_{local} by running the parallel applications with a single thread on a single processor system, causing all data to be placed in local memory. We were forced to use single threaded cases because our applications synchronize their threads using non-blocking spin locks. With multiple threads time-sliced on a single processor, the amount of time spent in a lock would be determined by this time-slicing, and so would invalidate the user time measurements. None of the applications spend much time contending for locks in the multiprocessor runs.

A simple measure of page placement effectiveness, which we call the "user-time expansion factor," γ,

$$T_{numa} = \gamma T_{local}, \tag{1}$$

can be misleading. A small value of γ may mean that our page placement did well, that the application spends little of its time referencing memory, or that local memory is not much faster than global memory. To better isolate these effects, we model program execution time as:

$$T_{numa} = T_{local} \left\{ (1-\beta) + \beta \left[\alpha + (1-\alpha)\frac{G}{L} \right] \right\} \tag{2}$$

L and G are the times for a 32-bit memory reference to local and global memory, respectively. As noted in section 2.2, G/L is about 2 on ACE (2.3 if all references are fetches).

Our model incorporates two sensitivity factors:

- α is the fraction of references to writable data that were actually made to local pages while running under our NUMA page placement strategy.
- β is the fraction of total user run time that would be devoted to referencing writable data if all of the memory were local.

α resembles a cache hit ratio: cache "hits" are references to local memory and cache "misses" are references to global memory. α measures both the use of private (non-shared) memory by the application and the success of the NUMA strategy in making such memory local. A "good" value for α is close to 1.0, indicating that most references are to local memory. A smaller value indicates a problem with the placement strategy or, as we typically found in practice, heavy use of shared memory in the application. Application sharing would not have been included, had we been able to use $T_{optimal}$ instead of T_{local}.

β measures the sensitivity of an application to the performance of writable memory. It depends both on the fraction of instructions that reference writable memory and on the speed of the instructions that do not (which itself is a function of local memory performance and of processor speed). Small values of β indicate that the program spends little of its time referencing writable memory, so the overall run time is relatively unaffected by α.

In the T_{global} runs, all writable data memory references were to global memory and none to local, thus α for these runs was 0. Substituting T_{global} for T_{numa} and 0 for α in equation 2 yields a model of the all-global runs:

$$T_{global} = T_{local} \left\{ (1-\beta) + \beta \frac{G}{L} \right\} \tag{3}$$

Solving equation 3 simultaneously with equation 2 for α and β yields:

$$\alpha = \frac{T_{global} - T_{numa}}{T_{global} - T_{local}} \tag{4}$$

$$\beta = \left(\frac{T_{global} - T_{local}}{T_{local}}\right)\left(\frac{L}{G-L}\right) \tag{5}$$

Our use of total user time eliminates the concurrency and serialization artifacts that show up in elapsed (wall clock) times and speedup curves. We are not concerned with how much faster our applications run on eight processors than they do on one; we are concerned with how much faster they run with our NUMA strategy than they do with all writable data in global memory. The greatest weakness of our model is that, because we use T_{local} rather than $T_{optimal}$, it fails to distinguish between global references due to placement "errors", and those due to legitimate use of shared memory. We were able to make this distinction only through *ad hoc* examination of the individual applications. We have begun to make and analyze reference traces of parallel programs to rectify this weakness.

On the whole, our evaluation method was both simple and informative. It was within the capacity of the ACE timing facilities[2]. It was not unduly sensitive to loss of precision despite the arithmetic in Equations 4 and 5. It did require that measurements be repeatable, so applications such as simulated annealing were beyond it. It also required that measurements not vary too much with the number of processors. Thus applications had to do about the same amount of work, independent of the number of processors, and had to be relatively free of lock, bus or memory contention.

[2]The only clock on the ACE is a 50Hz timer interrupt.

global memory when a threshold (a system-wide boot-time parameter which defaults to four) is passed. The page then remains in global memory until it is freed.

In terms of Tables 1 and 2, our policy answers LOCAL for any page that has not used up its threshold number of page moves and GLOBAL for any page that has. Once the policy decides that a page should remain in global memory, we say that the page has been *pinned*.

2.3.3 Changes to the Mach pmap Interface to Support NUMA

Our experience with Mach on the ACE confirms the robust design of the Mach pmap interface. Although this machine-independent paging interface was not designed to support NUMA architectures[1], we were able to implement automatic NUMA page placement entirely within the ACE pmap layer with only three small extensions to support pmap-level page caching:

- pmap_free_page operations,
- min-max protection arguments to pmap_enter, and
- a target processor argument to pmap_enter.

The original machine-independent component of the Mach VM system did not inform the pmap layer when physical page frames were freed and reallocated. This notification is necessary so that cache resources can be released and cache state reset before the page frames are reallocated. We split the notification into two parts to allow lazy evaluation. When a page is freed, pmap_free_page starts lazy cleanup of a physical page and returns a tag. When a page is reallocated, pmap_free_page_sync takes a tag and waits for cleanup of the page to complete.

Our second change to the pmap interface added a second protection parameter to the pmap_enter operation. Pmap_enter creates a mapping from a virtual to a physical address. As originally specified, it took a single protection parameter indicating whether the mapping should be writable or read-only. Since machine-independent code uses this parameter to indicate what the user is legally permitted to do to the page, we can interpret it as the maximum (loosest) permissions that the pmap layer is allowed to establish. We added a second protection parameter to indicate the minimum (strictest) permissions required to resolve the fault. Our pmap module is therefore able to map pages with the strictest possible permissions—replicating writable pages that are not being written by provisionally marking them read-only. Subsequent write faults will make such pages writable after first eliminating replicated copies. The pmap layers of non-NUMA systems can avoid these subsequent faults by initially mapping with maximum permissions.

Our third change to the pmap interface reduced the scope of the pmap_enter operation, which originally added a mapping that was available to all processors. Mach depended on this in that it did not always do the pmap_enter on the processor where the fault occurred and might resume the faulting process on yet another processor. Since NUMA management relies on understanding which processors are accessing which pages, we wanted to eliminate the creation of mappings on processors which did not need them. We therefore added an extra parameter that specified what processor needed the mapping. Newer versions of Mach may always invoke pmap_enter on the faulting processor[21], so the current processor id could be used as an implicit parameter to pmap_enter in place of our explicit parameter.

[1] And at least some of the Mach implementors considered NUMA machines unwise[19].

3 Performance Results

To discover the effectiveness and cost of our NUMA management system, we measured several multiprocessor applications running on the ACE.

3.1 Evaluating Page Placement

We were primarily interested in determining the effectiveness of our automatic placement strategy at placing pages in the more appropriate of local and global memory. Since most reasonable NUMA systems will replicate read-only data and code, we focused on writable data. We define four measures of execution time:

- T_{numa} is total user time (process virtual time as measured by the Unix time(1) facility) across all processors when running our NUMA strategy.
- $T_{optimal}$ is total user time when running under a page placement strategy that minimizes the sum of user and NUMA-related system time using future knowledge.
- T_{local} is the total user time of the application, were it possible to place all data in local memory.
- T_{global} is total user time when running with all writable data located in global memory.

We measured T_{numa} by running the application under our normal placement policy. We measured T_{global} by using a specially modified NUMA policy that placed all data pages in global memory.

We would have liked to compare T_{numa} to $T_{optimal}$ but had no way to measure the latter, so we compared

Policy Decision	Logical Page State			
	Read-Only	Global-Writable	Local-Writable	
			on own node	on other node
LOCAL	copy to local	unmap all copy to local Read-Only	No action	sync&flush other copy to local Read-Only
GLOBAL	flush all Global-Writable	No action	sync&flush own Global-Writable	sync&flush other Global-Writable

Table 1: NUMA Manager Actions for Read Requests

Policy Decision	Logical Page State			
	Read-Only	Global-Writable	Local-Writable	
			on own node	on other node
LOCAL	flush other copy to local Local-Writable	unmap all copy to local Local-Writable	No action	sync&flush other copy to local Local-Writable
GLOBAL	flush all Global-Writable	No action	sync&flush own Global-Writable	sync&flush other Global-Writable

Table 2: NUMA Manager Actions for Write Requests

and returns a location: LOCAL or GLOBAL. Given this location and the current known state of the page, the NUMA manager then takes the action indicated in Table 1 or Table 2. In each table entry, the top line describes changes to clean up previous cache state, the middle line tells whether the page is copied into local memory on the current processor and the bottom line gives the new cache state. All entries describe the desired new appearance; no action may be necessary. For example, a processor with a locally cached copy of a page need not copy the page from global memory again. "sync" means to copy a Local-Writable page from local memory back to global memory. "flush" means to drop any virtual mappings to a page and free any copies cached in local memory (used only for Read-Only or Local-Writable pages). "unmap" means to drop any virtual mappings to a page (used only for Global-Writable pages). "own", "other" and "all" identify the set of processors affected.

These NUMA manager actions are driven by requests from the pmap manager, which are in turn driven by normal page faults handled by the machine-independent VM system. These faults occur on the first reference to a page, when access is needed to a page removed (or marked read-only) by the NUMA manager, or when a mapping is removed due to Rosetta's restriction of only a single virtual address per physical page per processor.

When uninitialized pages are first touched, Mach fills them with zeros in the course of handling the initial zero-fill page fault. It does this by calling the *pmap_zero_page* operation of the pmap module. The page is then mapped using pmap_enter and processing continues. Since the the processor using the page is not known until pmap_enter time, we lazy evaluate the zero-filling of the page to avoid writing zeros into global memory and immediately copying them.

2.3.2 A Simple NUMA Policy That Limits Page Movement

In our ACE pmap layer, the NUMA policy module decides whether a page should be placed in local or global memory. We have implemented one simple policy (in addition to those we used to collect baseline timings) that limits the number of moves that a page may make. We initially place all pages in local memory because it is faster. Read-only pages are thus replicated and private writable pages are moved to the processor that writes them. Writably-shared pages are moved between local memories as the NUMA manager keeps the local caches consistent. Our policy counts such moves (transfers of page ownership) for each page and places the page in

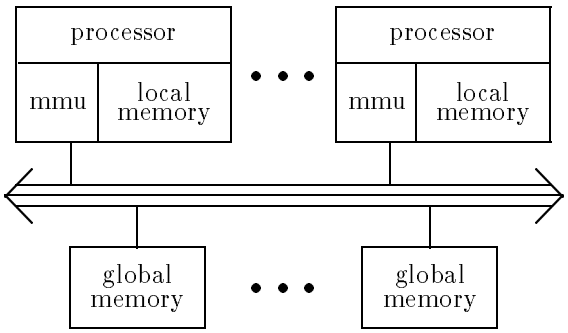

Figure 1: ACE Memory Architecture

Packaging restrictions prevent ACEs from supporting the full complement of memory and processors permitted by the IPC bus. The ACE backplane has nine slots, one of which must be used for (up to) 16 Mbytes of global memory. The other eight may contain either a processor or 16 Mbytes of global memory. Thus, configurations can range from 1 processor and 128 Mbytes to 8 processors and 16 Mbytes, with a "typical" configuration having 5 processors and 64 Mbytes of global memory. Most of our experience was with ACE prototypes having 4-8 processors and 4-16 Mbytes of global memory.

We measured the time for a 32-bit fetches and stores of local memory as 0.65μs and 0.84μs, respectively. The corresponding times for global memory are 1.5μs and 1.4μs. Thus, global memory on the ACE is 2.3 times slower than local on fetches, 1.7 times slower on stores, and about 2 times slower for reference mixes that are 45% stores. ACE processors can also reference each other's local memories directly, but we chose not to use this facility (see Section 4.4).

2.3 NUMA Management on the ACE

To minimize the costs of developing and maintaining kernel software, we followed the Mach conventions for machine-independent paging. We thus implemented our support for NUMA page placement entirely within the Mach machine-dependent pmap layer. Our pmap layer for the ACE is composed of 4 modules (see Figure 2): a pmap manager, an MMU interface, a NUMA manager and a NUMA policy. Code for the first two modules was obtained by dividing the pmap module for the IBM RT/PC into two modules, one of which was extended slightly to form the pmap manager, and the other of which was used verbatim as the MMU interface. The pmap manager exports the pmap interface to the machine-independent components of the Mach VM system, translating pmap operations into MMU operations and coordinating operation of the other modules.

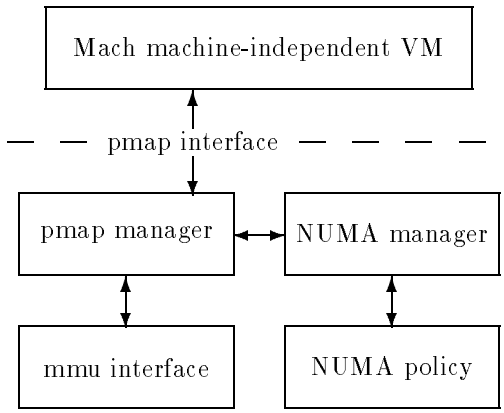

Figure 2: ACE pmap Layer

The MMU interface module controls the Rosetta hardware. The NUMA manager maintains consistency of pages cached in local memories, while the NUMA policy decides whether a page should be placed in local or global memory. We have only implemented a single policy to date, but could easily substitute another policy without modifying the NUMA manager.

2.3.1 The NUMA Manager

ACE local memories are managed as a cache of global memory. The Mach machine-independent page pool, which we call *logical* memory, is the same size as the ACE global memory. Each page of logical memory corresponds to exactly one page of global memory, and may also be cached in at most one page of local memory per processor. A logical page is in one of three states:

- *read-only* — may be replicated in zero or more local memories, must be protected read-only;
- *local-writable* — in exactly 1 local memory, may be writable; or
- *global-writable* — in global memory, may be writable by zero or more processors.

Read-only logical pages can be used for more than read-only code. A page of writable virtual memory may be represented by a read-only logical page if, for example, it contains a string table that is read but is never written. Similarly, a shared writable page may, over time, be local-writable in a sequence of local memories as the cache consistency protocol moves it around. The current content of a local-writable page is in the local memory and must be copied back to the global page before the page changes state.

The interface provided to the NUMA manager by the NUMA policy module consists of a single function, *cache_policy*, that takes a logical page and protection

we assume when a program begins executing that every page is *cacheable*, and may be placed in local memory. We declare that a page is *noncacheable* when the consistency protocol has moved it between processors (in response to writes) more than some small fixed number of times. All processors then access the page directly in global memory.

We believe that simple techniques can yield most of the locality improvements that can be obtained by an operating system. Our results, though limited to a single machine, a single operating system, and a modest number of applications, support this intuition. We have achieved good performance on several sample applications, and have observed behavior in others that suggests that no operating system strategy will obtain significantly better results without also making language-processor or application-level improvements in the way that data are grouped onto pages.

We describe our implementation in Section 2. We include a brief overview of the Mach memory management system and the ACE architecture. We present performance measurements and analysis in Section 3. Section 4 discusses our experience and what we think it means. Section 5 suggests several areas for future research. We conclude with a summary of what we have learned about managing NUMA memory.

2 Automatic Page Placement in a Two-Level Memory

2.1 The Mach Virtual Memory System

Perhaps the most important novel idea in Mach is that of machine-independent virtual memory[17]. The bulk of the Mach VM code is machine-independent and is supported by a small machine-dependent component, called the *pmap layer*, that manages address translation hardware. The *pmap interface* separates the machine-dependent and machine-independent parts of the VM system.

A Mach *pmap* (**p**hysical **map**) is an abstract object that holds virtual to physical address translations, called *mappings*, for the resident pages of a single virtual address space, which Mach calls a *task*. The pmap interface consists of such pmap operations as *pmap_enter*, which takes a pmap, virtual address, physical address and protection and maps the virtual address to the physical address with the given protection in the given pmap; *pmap_protect*, which sets the protection on all resident pages in a given virtual address range within a pmap; *pmap_remove*, which removes all mappings in a virtual address range in a pmap; and *pmap_remove_all*, which removes a single physical page from all pmaps in which it is resident. Other operations create and destroy pmaps, fill pages with zeros, copy pages, etc. The protection provided to the pmap_enter operation is not necessarily the same as that seen by the user; Mach may reduce privileges to implement copy-on-write or as part of the external paging system[22].

A pmap is a cache of the mappings for an address space. The pmap manager may drop a mapping or reduce its permissions, e.g. from writable to read-only, at almost any time. This may cause a page fault, which will be resolved by the machine-independent VM code resulting in another pmap_enter of the mapping. This feature had already been used on the IBM RT/PC, whose memory management hardware only allows a single virtual address for a physical page. We use it on the ACE to drive our consistency protocol for pages cached in local memory.

Mappings can be dropped, or permissions reduced, subject to two constraints. First, to ensure forward progress, a mapping and its permissions must persist long enough for the instruction that faulted to complete. Second, to ensure that the kernel works correctly, some mappings must be permanent. For example, the kernel must never suffer a page fault on the code that handles page faults. This second constraint limits our ability to use automatic NUMA page placement for kernel memory.

Mach views physical memory as a fixed-size pool of pages. It treats the physical page pool as if it were real memory with uniform memory access times. It is understood that in more sophisticated systems these "machine independent physical pages" may represent more complex structures, such as pages in a NUMA memory or pages that have been replicated. Unfortunately, there is currently no provision for changing the size of the page pool dynamically, so the maximum amount of memory that can be used for page replication must be fixed at boot time.

2.2 The IBM ACE Multiprocessor Workstation

The ACE Multiprocessor Workstation[9] is a NUMA machine built at the IBM T. J. Watson Research Center. Each ACE consists of a set of processor modules and global memories connected by a custom global memory bus (see Figure 1). Each ACE processor module has a ROMP-C processor[12], Rosetta-C memory management unit and 8Mb of local memory. Every processor can address any memory, with non-local requests sent over a 32-bit wide, 80 Mbyte/sec Inter-Processor Communication (IPC) bus designed to support 16 processors and 256 Mbytes of global memory.

Simple But Effective Techniques for NUMA Memory Management

William J. Bolosky[1] Robert P. Fitzgerald[2] Michael L. Scott[1]

Abstract

Multiprocessors with non-uniform memory access times introduce the problem of placing data near the processes that use them, in order to improve performance. We have implemented an automatic page placement strategy in the Mach operating system on the IBM ACE multiprocessor workstation. Our experience indicates that even very simple automatic strategies can produce nearly optimal page placement. It also suggests that the greatest leverage for further performance improvement lies in reducing *false sharing*, which occurs when the same page contains objects that would best be placed in different memories.

1 Introduction

Shared-memory multiprocessors are attractive machines for parallel computing. They not only support a model of parallelism based on the familiar von Neumann paradigm, they also allow processes to interact efficiently at a very fine level of granularity. Simple physics dictates, however, that memory cannot simultaneously be located very close to a very large number of processors. Memory that can be accessed quickly by one node of a large multiprocessor will be distant from many other nodes. Even on a small machine, price/performance may be maximized by an architecture with non-uniform memory access times.

On any Non-Uniform Memory Access (NUMA) machine, performance depends heavily on the extent to which data reside close to the processes that use them. In order to maximize locality of reference, data replication and migration can be performed in hardware (with consistent caches), in the operating system, in compilers or library routines, or in application-specific user code. The last option can place an unacceptable burden on the programmer and the first, if feasible at all in large machines, will certainly be expensive.

We have developed a simple mechanism to automatically assign pages of virtual memory to appropriately located physical memory. By managing locality in the operating system, we hide the details of specific memory architectures, so that programs are more portable. We also address the locality needs of the entire application mix, a task that cannot be accomplished through independent modification of individual applications. Finally, we provide a migration path for application development. Correct parallel programs will run on our system without modification. If better performance is desired, they can then be modified to better exploit automatic page placement, by placing into separate pages data that are *private* to a process, data that are shared for reading only, and data that are *writably shared*. This segregation can be performed by the applications programmer on an *ad hoc* basis or, potentially, by special language-processor based tools.

We have implemented our page placement mechanism in the Mach operating system[1] on a small-scale NUMA machine, the IBM ACE multiprocessor workstation[9]. We assume the existence of faster memory that is *local* to a processor and slower memory that is *global* to all processors, and believe that our techniques will generalize to any machine that fits this general model.

Our strategy for page placement is simple, and was embedded in the machine-dependent portions of the Mach memory management system with only two man-months of effort and 1500 lines of code. We use local memory as a cache over global, managing consistency with a directory-based ownership protocol similar to that used by Li[15] for distributed shared virtual memory. Briefly put, we replicate read-only pages on the processors that read them and move written pages to the processors that write them, permanently placing a page in global memory if it becomes clear that it is being written routinely by more than one processor. Specifically,

[1] Department of Computer Science,
University of Rochester, Rochester, NY 14627.
internet: bolosky@cs.rochester.edu, scott@cs.rochester.edu
[2] IBM Thomas J. Watson Research Center,
PO Box 218, Yorktown Heights, NY 10598-0218.
internet: fitzgerald@ibm.com