(4)

Memory Management for Large-Scale NUMA Multiprocessors

T.J. LeBlanc, B.D. Marsh, and M.L. Scott

Technical Report 311
March 1989

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

MEMORY MANAGEMENT FOR LARGE-SCALE NUMA MULTIPROCESSORS

Thomas J. LeBlanc
Brian D. Marsh
Michael L. Scott

Computer Science Department
University of Rochester
Rochester, New York 14627

leblanc@cs.rochester.edu
marsh@cs.rochester.edu
scott@cs.rochester.edu

March 1989

Abstract

Large-scale shared-memory multiprocessors such as the BBN Butterfly and IBM RP3 introduce a new level in the memory hierarchy: multiple physical memories with different memory access times. An operating system for these NUMA (NonUniform Memory Access) multiprocessors should provide traditional virtual memory management, facilitate dynamic and widespread memory sharing, and minimize the apparent disparity between local and nonlocal memory. In addition, the implementation must be scalable to configurations with hundreds or thousands of processors.

This paper describes memory management in the Psyche multiprocessor operating system, under development at the University of Rochester. The Psyche kernel manages a multi-level memory hierarchy consisting of local memory, nonlocal memory, and backing store. Local memory stores private data and serves as a cache for shared data; nonlocal memory stores shared data and serves as a disk cache. The system structure isolates the policies and mechanisms that manage different layers in the memory hierarchy, so that customized data structures and policies can be constructed for each layer. Local memory management policies are implemented using mechanisms that are independent of the architectural configuration; global policies are implemented using multiple processes that increase in number as the architecture scales. Psyche currently runs on the BBN Butterfly Plus multiprocessor.

This work was supported in part by NSF CER grant number DCR-8320136 and Darpa/ETL contract number DACA76-85-C-0001.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 311	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Memory Management for Large-Scale NUMA Multiprocessors		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) T.J. LeBlanc, B.D. Marsh, and M.L. Scott		8. CONTRACT OR GRANT NUMBER(s) DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department 734 Computer Studies Bldg University of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS D. Adv. Res. Proj. Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE March 1989 13. NUMBER OF PAGES 21
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* U.S. Army ETL Fort Belvoir, VA 22060		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Psyche, non-uniform memory, virtual memory shared memory

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Large-scale shared-memory multiprocessors such as the BBN Butterfly and IBM RP3 introduce a new level in the memory hierarchy: multiple physical memories with different memory access times. An operating system for these NUMA (NonUniform Memory Access) multiprocessors should provide traditional virtual memory management, facilitate dynamic and widespread memory sharing, and minimize the apparent disparity between local and nonlocal memory. In addition, the implementation must be scalable to configurations with hundreds or thousands

20. ABSTRACT (Continued)

of processors.

This paper describes memory management in the Psyche multiprocessor operating system, under development at the University of Rochester. The Psyche kernal manages a multi-level memory hierarchy consisting of local memory, nonlocal memory, and backing store. Local memory stores private data and serves as a cache for shared data; nonlocal memory stores shared data and serves as a disk cache. The system structure isolates the policies and mechanisms that manage different layers in the memory hierarchy, so that customized data structures and policies can be constructed for each layer. Local memory management policies are implemented using mechanisms that are independent of the architectural configuration; global policies are implemented using multiple processes that increase in number as the architecture scales. Psyche currently runs on the BBN Butterfly Plus multiprocessor.

MEMORY MANAGEMENT FOR LARGE-SCALE NUMA MULTIPROCESSORS

Abstract

Large-scale shared-memory multiprocessors such as the BBN Butterfly and IBM RP3 introduce a new level in the memory hierarchy: multiple physical memories with different memory access times. An operating system for these NUMA (NonUniform Memory Access) multiprocessors should provide traditional virtual memory management, facilitate dynamic and widespread memory sharing, and minimize the apparent disparity between local and nonlocal memory. In addition, the implementation must be scalable to configurations with hundreds or thousands of processors.

This paper describes memory management in the Psyche multiprocessor operating system, under development at the University of Rochester. The Psyche kernel manages a multi-level memory hierarchy consisting of local memory, nonlocal memory, and backing store. Local memory stores private data and serves as a cache for shared data; nonlocal memory stores shared data and serves as a disk cache. The system structure isolates the policies and mechanisms that manage different layers in the memory hierarchy, so that customized data structures and policies can be constructed for each layer. Local memory management policies are implemented using mechanisms that are independent of the architectural configuration; global policies are implemented using multiple processes that increase in number as the architecture scales. Psyche currently runs on the BBN Butterfly Plus multiprocessor.

1. Introduction

Large-scale, shared-memory multiprocessors with hundreds or thousands of processing nodes offer tremendous computing potential. This potential remains largely unrealized, due in part to the complexity of managing shared memory. NUMA (NonUniform Memory Access) architectures such as the BBN Butterfly [1] and IBM RP3 [17] further complicate matters by introducing a new level in the memory hierarchy: multiple physical memories with different memory access times. An operating system for NUMA multiprocessors should provide traditional virtual memory management, facilitate dynamic and wide-spread sharing, and minimize the apparent disparity between local and nonlocal memory. Conventional memory management assumptions must be re-evaluated, new techniques developed, and the interactions between paging, sharing, and NUMA management explored. In addition, the implementation must scale with the architecture.

This paper describes memory management in the Psyche multiprocessor operating system,[1] under development at the University of Rochester. Psyche is designed to support a wide variety of models for parallel programming. Although appropriate for use on bus-based, shared-memory multiprocessors such as the Sequent and Encore systems, Psyche is especially well-suited for use on large-scale NUMA multiprocessors.

Memory management plays a central role in the Psyche kernel. Protected procedure calls, the primitive communication mechanism in Psyche, are implemented using page faults, providing a hook for protection policies implemented in software. The kernel manages a transparent memory hierarchy consisting of multiple memory classes (including local and non-local memory) and backing store, giving users the illusion of uniform memory access times on NUMA multiprocessors. As in most systems, the kernel also implements virtual memory and demand paging.

Three goals of the Psyche memory management system are:

- *scalability* – the implementation should scale to multiprocessors with hundreds of processing nodes,
- *portability* – the implementation should be portable to any paged, shared-memory multiprocessor, and
- *generality* – specific details of the Psyche kernel interface should be isolated in the implementation as much as is possible.

We acheive these goals by structuring memory management into four abstraction layers. The lowest layer hides the details of the hardware, including hardware page tables, so that the implementation can be readily ported. The lowest layer also implements local memory management policies using mechanisms that are independent of the architectural configuration. A separate layer encapsulates the management of nonlocal memory, including page replication and migration, so that different policies for NUMA memory management may be incorporated easily. This layer implements global policies using processes that increase in number as the architecture scales. Additional layers separate demand paging and the management of backing store from the Psyche kernel interface. No low-level layer takes advantage of Psyche-specific abstractions, so changes in the kernel interface are localized in the implementation. No high-level layer takes advantage of particular hardware properties, enhancing portability.

[1] The evolution of the Psyche design is presented in a companion paper [19].

We present a brief overview of Psyche in section 2, followed by a description of the memory management abstraction layers in section 3. We then describe how the various layers cooperate to implement virtual memory (section 4) and NUMA memory management (section 5). Related work and conclusions are discussed in sections 6 and 7.

2. Psyche Overview

The Psyche programming model [19] is based on passive data abstractions called *realms*, which include both code and data. The code constitutes a protocol for manipulating the data and for scheduling threads of control. Invocation of protocol operations is the principal mechanism for accessing shared memory, thereby implementing interprocess communication.

Depending on the degree of protection desired, an invocation of a realm operation can be as fast as an ordinary procedure call, termed *optimized* invocation, or as safe as a heavyweight process switch, termed *protected* invocation. Unless the caller insists on protection, both forms of invocation look exactly like subroutine calls. The kernel implements protected invocations by catching and interpreting page faults.

To permit sharing of arbitrary realms at run time, Psyche arranges for all realms to reside in a uniform address space. The use of uniform addressing allows processes to share data structures and pointers without the need to translate between address spaces. Realms that are known to contain private data or realms that can only be accessed using protected invocations can overlap, ensuring that normal operating system workloads fit within the Psyche address space, while allowing very flexible and dynamic sharing relationships.

At any moment in time, only a small portion of the Psyche uniform address space is accessible to a given process. Every Psyche process executes within a *protection domain*, an execution environment that denotes the set of available rights. A protection domain's view of the Psyche address space, embodied by a hardware page table, contains those realms for which the right to perform optimized invocations has been demonstrated to the kernel. Processes move between protection domains, inheriting a new view of the address space and the corresponding set of rights, using protected invocations.

3. Memory Management Organization

3.1. Motivation

Traditional operating systems use memory management hardware and the ability to detect accesses to invalid pages to implement both address space protection and virtual memory. However, as the only hardware protection mechanism provided by most architectures and the sole indicator of dynamic memory reference patterns, memory mappings are often used to serve other purposes as well. For example, in BSD Unix, invalid page mappings are used to simulate reference bits [2]. In the Apollo Aegis system, memory management is used to implement a single-level store, thereby unifying memory and backing store [14]. Accent [10] and Mach [21] use memory mapping techniques to implement efficient communication, employing copy-on-write to avoid multiple physical copies of messages. Kai Li has used memory management to implement a distributed shared memory [16].

As the role of the memory management system has expanded within the operating system, it has become increasingly important to structure the implementation to separate the disparate functions supported by the mapping hardware. For example, Mach separates the machine-dependent and machine-independent portions of the memory management system [18], and the page replacement decision from the management of backing store [21]. To meet our goals of scalability, portability, and generality, we have extended this trend.

3.2. Layers

The Psyche memory mangement system is structured into four layers of abstraction. The NUMA (nonuniform memory) layer is responsible for local physical memory allocation and hardware page tables. The UMA (uniform memory) layer manages global physical memory and implements the illusion of uniform memory access time. The VUMA (virtual memory) layer manages backing store and implements the system's default pager. The PUMA (Psyche memory) layer implements the Psyche kernel interface. This layering is depicted in Figure 1.

The NUMA layer isolates the machine-dependent portion of the memory management system by exporting an interface analogous to the pmap interface in Mach [18]. The NUMA layer defines two abstractions: NUMA pages and NUMA address spaces. The NUMA page size is a multiple of the physical page size. The physical memory underlying a NUMA page may be allocated anywhere in main memory, as shown at the bottom of Figure 1; different NUMA pages in the same NUMA address space may exhibit different memory access times. Operations are provided to create, destroy, and copy a NUMA page, allocating and deallocating physical memory as needed. The reference and modify bits for a NUMA page may be examined by higher layers in order to implement global memory allocation policies. Operations are also provided to create an address space (and a corresponding hardware page table) on a specific node, activate an address space on a node, map and unmap NUMA pages in an address space, and set the protection attributes of a NUMA page in an address space.

In addition to hardware page tables, the NUMA layer implements a *core map* for use by memory allocation policies. The core map contains an entry for each NUMA page allocated in local memory. Each entry is linked into one of three lists that is maintained for each processor

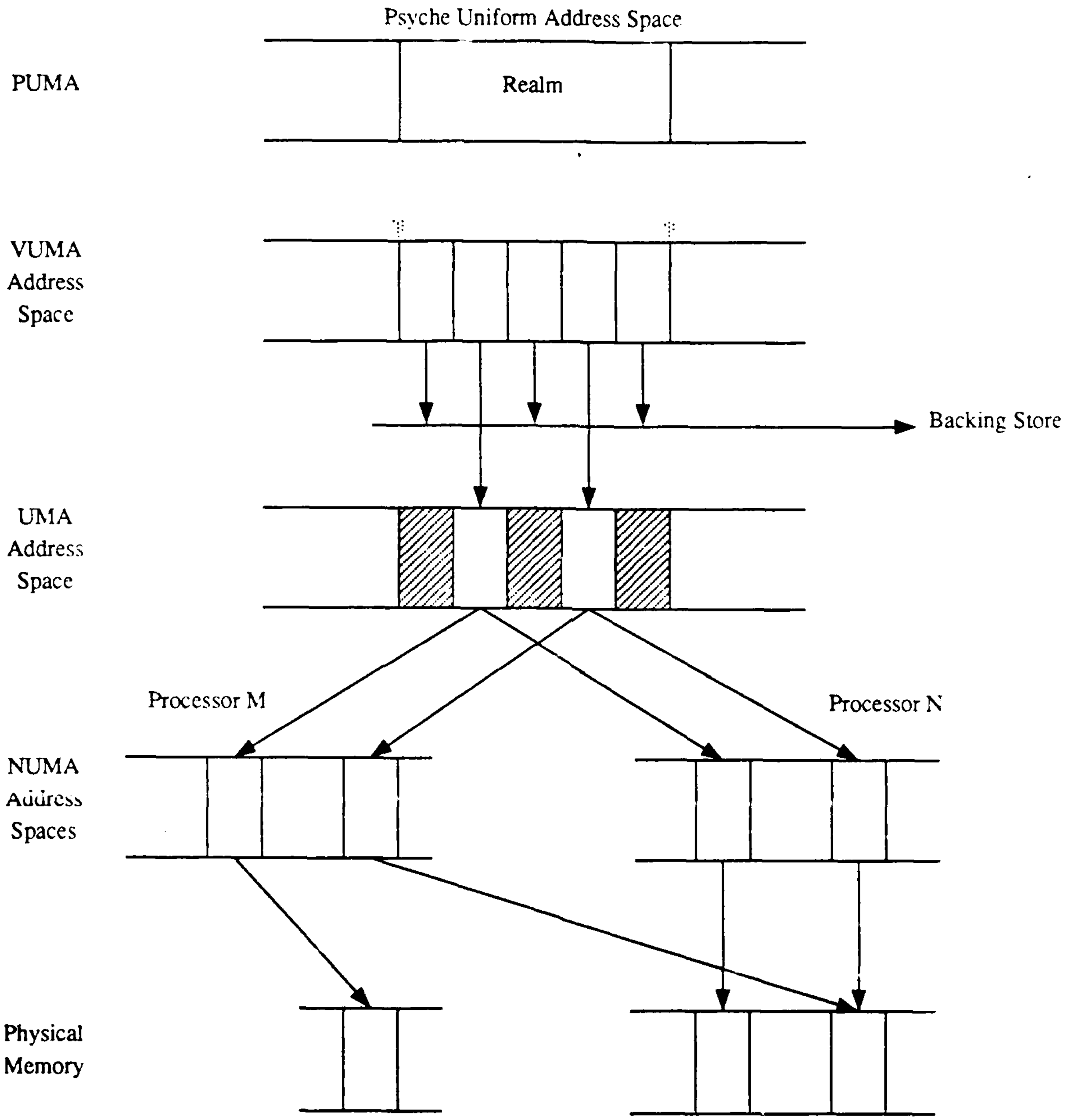

Figure 1: Psyche Memory Management Layers

node: allocated, free, and *in-transit*. (Pages selected for removal from local memory remain in-transit until actually moved to another location in memory.) Each entry also contains software reference and modify bits that reflect local usage, and a list of local hardware page tables that contain a mapping for the page.

The UMA layer is responsible for managing global physical memory. It uses local memory as a cache for shared data, thereby minimizing both memory access time and memory contention. The UMA layer also uses global memory as a cache for backing store, thereby reducing the number of I/O operations required. Global policies for physical memory allocation are implemented by the UMA layer based on information provided by local policies in the NUMA layer.

The UMA layer implements two abstractions: UMA pages and UMA address spaces. Each UMA page is implemented by one or more NUMA pages. The physical memory underlying an UMA page may be allocated anywhere in main memory, and may exist in several different locations simultaneously. Figure 1 shows two UMA pages, which are mapped into two NUMA address spaces on different processor nodes. One of the UMA pages is replicated in physical memory, and each address space maps a different copy. The other UMA page is represented by a single copy, which is mapped by both address spaces. Due to automatic page migration and replication, UMA pages in the same UMA address space appear to exhibit comparable memory access times.

To admit large, sparse address spaces, UMA address spaces are implemented using a hash table. Entries contain protection and global reference information for an UMA page, and the location of the various NUMA pages that implement the UMA page. Each UMA address space may be represented by several underlying NUMA address spaces and hardware page tables.

The VUMA layer implements virtual memory and demand paging. A VUMA address space augments an underlying UMA address space with entries for VUMA pages that may reside on backing store. Figure 1 shows five VUMA pages. Two VUMA pages are resident in memory and their contents are contained in two UMA pages. Three VUMA pages are resident on backing store. A default system pager in the VUMA layer manages backing store for user programs. Psyche also allows users to define Mach-style external pagers [21] that manage backing store for an application. The VUMA layer propagates requests for VUMA pages backed by a user-defined pager to the PUMA layer, where communication with external pagers is implemented.

The PUMA layer implements the Psyche kernel interface, including the Psyche uniform address space, protection domains, protected procedure calls, and communication with user-defined pagers. The uniform address space is represented as a splay tree containing blocks of the virtual address space. Virtual addresses within a block are either all allocated or all free. Operations are provided to allocate and deallocate areas of the shared virtual address space.

and to find the realm allocated to a particular virtual address. The latter operation is the most important since it is used each time a realm is accessed for the first time from a given protection domain. By using a splay tree, the find operation requires amortized time logarithmic in the number of blocks in the Psyche virtual address space, optimizing access to realms that are frequently used.

Each Psyche protection domain is implemented by a VUMA address space and a list of VUMA pages, describing the core image of the initial realm of the protection domain. In addition, each protection domain maintains a list of realms that have been accessed from within the protection domain, and a list of other protection domains that have accessed its initial realm. These lists are used to avoid authorization on subsequent calls, and to ensure consistency across protection domains when realms are destroyed.

The protection attributes defined by the PUMA layer are definitive, in that the protection associated with a page in lower layers may be more restrictive than the true protection. The true access rights to a page within a protection domain can only be determined by the PUMA layer. In particular, a protected procedure call will generate an invalid fault that can only be resolved by the PUMA layer, in cooperation with protection policies defined by the user.

3.3. Interaction Between Layers

Operations within the virtual memory system are induced by two kinds of events: user memory references that cause a page fault and system daemons that execute periodically. Events of both kinds occur at all layers in the system; each layer defines its own response to faults and the nature of the daemons that execute within the layer.

Page faults propagate upwards, from the lowest layer in the system (NUMA) to the highest (PUMA). Each layer in the system may use faults to implement demand-driven policies. When a layer elects to service a fault, the fault propagates no higher. The NUMA layer, which is responsible for local memory management, can use faults to implement second-chance reclamation schemes for "paged out" local memory. The UMA layer, which attempts to minimize the number of nonlocal references, can use faults to implement software caching schemes. The VUMA layer, which manages backing storage, can use faults to implement demand paging. Finally, the PUMA layer, which implements higher-level operating systems functions, can use faults to implement protected procedure calls. We describe how the layers cooperate to implement virtual memory and NUMA memory management in later sections.

Active agents, or daemons, are necessary for the implementation of certain policies. Daemons may operate at each of the levels in the memory management system, and are distinguished on the basis of function. Daemons within the NUMA layer help to implement the policy for local page replacement. Each daemon manages one local memory, allowing the architecture to scale without affecting page replacement. Daemons in the NUMA layer interact with the UMA layer using a buffered queue and upcalls. Daemons within the UMA layer help to implement the policy for global memory management and for NUMA memory management. Global UMA daemons service page replacement requests provided by the page daemons at the NUMA layer. Other daemons monitor dynamic reference behavior to implement page migration and replication policies. Daemons within the VUMA layer manage backing store, serving pageout requests from UMA daemons. One result of our layered organization is that daemons at each layer are independent of deamons in other layers, performing different functions and executing at different rates.

3.4. Rationale

Our particular division of functionality into layers reflects both the locality inherent in the memory hierarchy and our original goals of portability, scalability, and generality. Different levels in the memory hierarchy are isolated within different layers in the design, allowing each layer to tailor its implementation to one level of the hierarchy. Portability results from isolating machine-dependent details within the lowest level of the memory hierarchy. Scalability is supported by the separation of local and global resource management. Generality is achieved by isolating Psyche abstractions within the highest level of the memory hierarchy, separated from traditional memory management functions.

The separation between the NUMA and UMA layers results from the distinction between local and remote memory. We use simple, well-understood policies for local memory allocation, which serve as a building block for global policies. Since the NUMA layer uses only local information to implement its policies, it is unaffected as the architecture scales to larger configurations. Within the UMA layer, more complex global policies for page migration and replication are simplified by using local information provided by the NUMA layer.

The separation between the UMA and VUMA layers reflects the difference between main memory and backing store. The UMA layer uses physical memory as a large disk cache, avoiding disk operations whenever possible. The VUMA layer implements demand paging between memory and backing store. The default pager process and policies that optimize disk utilization are isolated within the VUMA layer.

The separation between the VUMA and PUMA layers isolates the effects of the Psyche kernel interface from the implementation of virtual memory. Given the relative novelty of multiprocessor operating systems, and the lack of long-term experience with multiprocessor programming, evolution in the kernel interface is to be expected. A narrow window between the VUMA and PUMA layers allows experimentation at the kernel interface level without a complete reimplementation of memory management. In addition, this separation ensures that techniques used in the NUMA, UMA, and VUMA layers can be applied to other multiprocessor operating systems.

The data structures in each of the layers are a cache for information maintained at a higher layer, usually in a more compact form. For each protection domain, the PUMA layer maintains a list of realms for which access has been authorized. The mappings for individual pages in the protection domain reside in the VUMA layer. Multiple copies of each page are described in the UMA layer. Multiple hardware page tables for each UMA address space are maintained in the NUMA layer.

As in Mach [18], we can discard information within a layer and use lazy evaluation to construct it. Whenever an invalid fault cannot be interpreted at a given layer, it is propagated to the next layer, which can reconstruct mappings in the lower layer. For example, the contents of hardware page tables can be reconstructed from corresponding UMA page tables. If a hardware page table is extremely large or if all page tables consume too much physical memory, page table entries may be discarded; subsequent invalid faults cause the corresponding portions of a page table to be reconstructed by the UMA layer. In addition, an individual mapping in a hardware page table can be discarded by the NUMA layer to force a reevaluation of a page placement decision. When a reference to the corresponding page occurs, the global policy implemented by the UMA layer is invoked to determine whether a page should be replicated or migrated. Finally, when an address space is created, we need not construct a representation for all pages in the address space at all levels. Lazy evaluation of mappings avoids mapping pages that are never referenced and postpones page placement decisions until reference information is available.

4. Virtual Memory

The virtual memory system attempts to maintain a pool of available physical memory on each processor node, while ensuring that virtual pages remain somewhere in memory for as long as possible. These two distinct goals are achieved using two different policies: local page daemons manage the physical memory on each processor node to ensure that local memory needs can be satisfied; global page daemons migrate pages among processor nodes and backing store to ensure that overall system needs are met.

Each processor node has a NUMA page daemon to manage its local memory. When a need for physical memory on a particular node arises, the least-recently-used (LRU) page on that node is chosen for replacement. We use the Clock algorithm [8] to implement LRU approximation. Clock is an example of a *global page replacement algorithm* [7], in that the algorithm considers a page for replacement without regard to which process owns the page. We chose a global page replacement algorithm, in contrast to a local page replacement algorithm such as Working Set (WS), because global policies are easier to implement and appear to be more appropriate for multiprocessors, wherein most of the processes in the system belong to a small number of applications and page usage will frequently span process boundaries.

The Clock algorithm requires that we maintain reference information for each page. Although this task is straightforward on a uniprocessor that supports reference bits, and is even feasible on a uniprocessor lacking hardware reference bits [2], several complications arise in the multiprocessor environment.

- Reference bits are associated with page table entries, and therefore record page map reference information, rather than page reference information. In a typical uniprocessor environment, where each page is mapped by a single page table entry, this distinction is not important. However, in a multiprocessor environment that encourages page sharing, page reference information may need to be synthesized from several hardware reference bits, some of which reside in non-local memory.
- The operating system must periodically update reference information by clearing hardware reference bits. Even if reference information is only computed for pages in local memory, page tables that map the page may be located anywhere in memory. In addition, translation lookaside buffer (TLB) entries may need to be flushed to maintain consistency with page table entries, introducing additional cost and complexity [3,20].
- To minimize average memory access time on a NUMA multiprocessor, a logical page might be represented by many physical copies. Page reference information for the logical page must be synthesized from the reference information for the various copies. This synthesis is expensive, but could be extremely important if it prevents the removal of the last copy of a page from memory when that page is being used.

In Psyche, local reference information is used and maintained by the NUMA page daemon, which runs when the available memory on a node falls below some threshold. The daemon executes a two-handed Clock algorithm [15] on the local core map to select a page to be replaced in the local memory. The core map contains node-local reference bits, which are set only if the page has been referenced locally. The daemon sets the node-local reference bit if the reference bit in any of the local page table entries for the page is set. The rate at which the core map is scanned depends on the amount of local physical memory that is available. A page is selected by the local page daemon if it has not been used recently by a process on the same node as the page.

Selected pages are placed on the *in-transit* list. When a page is placed on the in-transit list. all local mappings are made invalid. Any remote mappings for the page are left unchanged. By leaving the remote mappings intact, we put off the expense of remote invalidations until the page is actually removed from local memory. Immediate invalidation is not required for remote mappings because the contents of the physical page are unchanged and any global reference information that could be collected with an invalid remote mapping could not affect the decision to remove the page from local memory. If a page is referenced locally while on the in-transit list, it is removed from the list, mapped into the address space in which the fault occurred, and its node-local reference bit is set. In this case, we avoid the cost of remote invalidation altogether.

One result of the decision to postpone remote invalidations is that the NUMA layer does not need to trigger remote invalidation. All mappings to nonlocal memory are created by the UMA layer and page table consistency decisions are made by the UMA layer.

The in-transit list is processed by an UMA page daemon, which runs whenever the size of an in-transit list exceeds some threshold. The UMA page daemon examines pages on the in-transit list in order. The daemon must decide whether to move the page, delete it, or leave it in place. *On a multiprocessor with local and non-local memory*, the UMA page daemon will rarely, if ever, choose to leave a page in place. Since the page is not being used locally, either it is not in use at all, in which case it can be removed from physical memory, or it is being used remotely, and therefore can be moved without increasing the access time for the page.

The UMA page daemon can delete the page if there are other copies of the page in memory. In this case, all mappings for the physical page can be changed to refer to a different copy or made invalid. Subsequent faults provide an opportunity for reevaluating the need for a local copy.

If the UMA page daemon chooses to move the page, any remote mappings for the page are made invalid. The daemon moves the page to a node with available physical memory, preferably a node that already contains a mapping for the page. The node-local reference bit for the new copy is set, so that the page will not be chosen for removal from its new location until a complete Clock cycle of the corresponding NUMA page daemon has occurred. The page is then placed on the *pageout* list, from which main memory pageout decisions are made.

[2] Since we are also implementing the illusion of uniform memory access time in the UMA layer, poor choices by the global page daemon can be alleviated by page migration and replication policies.

A VUMA pageout daemon runs whenever the amount of available physical memory falls below some threshold. Pages on the pageout list are candidates for removal from main memory. However, since all mappings to a page on the pageout list have been made invalid, any reference to such a page will result in a page fault, allowing the kernel to collect global reference information. The overhead of maintaining global reference information is one page fault per page, and is incurred only to prevent the removal of a page from main memory. During a subsequent page fault, all hardware mappings are reconstructed and the page is removed from the pageout list. If the page is not referenced before it reaches the head of the pageout list, a flush to backing store is initiated (if necessary) and the physical page is reclaimed. As a result, a page is flushed from main memory only when there is little physical memory available anywhere in the system, the page has already been moved from one memory location to another (in effect, moving the page into the disk cache), and the page has not been referenced locally in its current location for a complete cycle of the local Clock algorithm.

When a page is requested, it may be found in one of several places: in local memory, in global memory, on an in-transit list, on a pageout list, or on backing store. The NUMA layer maps pages in local memory or on an in-transit list. The UMA layer maps pages in global memory or on a pageout list. The VUMA layer retrieves pages from backing store.

When a page is brought into memory from backing store, it is placed in the local memory that requested the page, assuming there is free memory. The requested page is mapped into the faulting address space. Prefetching is used to minimize disk traffic, but not the number of page faults. Prefetched pages are also stored in local memory as part of a single I/O operation, but are not mapped into any address space. Subsequent references to those pages may cause the page to migrate or replicate.

A major advantage of our virtual memory organization is that the management of local memory, global memory, and backing store are separated, with separate policies for each. The number and location of the various page daemons can vary with the complexity of the corresponding policies and the demands within the memory hierarchy. By buffering pages between daemons that implement the various policies, we can cluster the transfer of pages between one memory and another or between memory and backing store and amortize policy computations over several pages. In addition, we effectively "cache" pages between layers in the memory hierarchy, allowing them a second chance at either local memory or main memory. Furthermore, the separation of policies allows a separation of the paging rates for local and global memory: we can utilize well-understood policies and paging rates for local memory management that are independent of the computation of more complex global policies.

Our virtual memory implementation has several attractive properties as well. Pages are removed from main memory infrequently, and then only when there is a shortage of physical memory and the page is known not to have been referenced recently from anywhere in the system. Frequently used pages, even if copies, take precedence over unused pages; copies of a page are coalesced before the page is considered for removal from main memory. Global reference information, which is expensive to maintain, is collected only on pages that are being considered for removal from main memory. Finally, invalidation of remote page table entries and TLBs, which can be very complex and expensive, is done only when absolutely necessary (*i.e.*, when a page is actually moved or deleted).

5. Page Replication and Migration

The Psyche memory management system provides the illusion of uniform memory access time on a NUMA multiprocessor. We attempt to minimize the number of references to nonlocal memory by using page migration and replication. Migration causes pages that are write-shared to move close to a processor that accesses them frequently. Replication is used to minimize memory contention, giving each processor that reads a page a different (possibly nonlocal) copy, and minimize remote memory accesses, ensuring that each processor has a local copy.

Page replication is particularly well-suited to readonly pages, since duplicate copies always remain consistent. Page replication can also be applied to read-write pages, but the system must maintain consistency among copies. Cache coherence protocols, such as a directory-based write-invalidate or write-through protocol, can be implemented in software, maintaining consistency by propogating changes to all copies or merging copies when a change occurs. The policy that controls page replication must balance the benefits of replicating read-write pages with the overhead of maintaining consistency in software.

Page migration is also appropriate for readonly pages when physical memory constraints preclude replication. By placing the page close to the processor that uses it most frequently, we can minimize average access time for the page. Page migration can also be used with read-write pages, improving access time without the need for a consistency protocol.

Efficient page migration and replication policies are currently under investigation [4, 11]. Here we will describe the mechanisms provided by the Psyche memory management system and the simple policy we have implemented.

Any policy for effective page replication and migration requires extensive reference information. Unlike our page replacement algorithm, which performs local memory management and therefore requires only local reference information, global memory management requires global reference information. Ideally, page usage information for each processor would be available. Although potentially expensive to maintain, approximate page usage statistics based on aging reference bits can result in substantially better policy decisions than single reference bits [12]. An alternative is to base page placement decisions on more recent information, such as the last reference to the page.

Both short-term and long-term reference behavior can play a role in a global memory management policy. A single reference to a page is a good indication of future use, and therefore is an important consideration in the placement decision. In those circumstances where the placement decision has little global impact (for example, the decision to make a local copy of a readonly page), a single reference may be sufficient to justify the placement decision. In other situations, where the placement decision must balance references from many different nodes, long-term reference history will usually result in a better decision.

Since dynamic reference behavior and local memory mangement policies determine the optimal page location, static placement decisions are unlikely to be effective. Therefore, the system must periodically reevaluate page placement decisions. Fortunately, only mappings for pages in nonlocal memory need to be reevaluated, since accesses to pages in local memory are already optimal

A page is stored in nonlocal memory for one of two reasons: (1) at the time of the placement decision, there was no room in local memory for a copy of the page, or (2) the page could not easily be migrated to local memory. A page that is stored in nonlocal memory due to a temporary shortfall in local memory can be copied or migrated when local memory becomes available. A page that doesn't migrate to local memory due to recent global reference behavior may want to migrate if the reference behavior changes. Recognizing the availability of local memory is easy; recognizing a change in global reference behavior is much more difficult.

Both the UMA layer and VUMA layer are responsible for freeing pages in local memory, at the request of the NUMA page daemon. The UMA layer moves pages to nonlocal memory; the VUMA layer writes pages to backing store. When local pages are freed, the UMA daemon makes invalid all local mappings to nonlocal, readonly pages. Subsequent references to the page will cause a page fault, enabling a new placement decision to be made, which can result in the creation of a local copy.

To recognize a change in reference behavior, we must collect and evaluate global reference information. One form of reference behavior is already collected by the virtual memory implementation: pages that are not used locally become candidates for migration; pages that are migrated and not used globally become candidates for pageout. For pages that are in use, either locally or globally, we must choose between migration and replication.

A shared page can be in one of three states: readonly, read-dominant, and read-write. A readonly page is never modified, and therefore, may be freely copied on reference, subject to the availability of physical memory. A read-write page is often modified, and will typically have only one copy, which may migrate according to the frequency of references. A read-dominant page is infrequently modified, and can be either migrated or replicated, depending on the cost of maintaining consistency and recent references to the page. Over time, a read-write page may become read-dominant, if few modifications to the page occur. Similarly, a read-dominant page may become read-write if, over time, the frequency of modifications increases significantly. The system must monitor write operations to both read-write and read-dominant pages to identify these transitions.

No single action on the part of a user program causes a read-write page to become read-dominant. It is the absence of write operations over a period of time that signals this transition. Using the modify bit provided by the hardware, we periodically scan mappings and attempt to determine when a read-write page is no longer being modified. Such a page then becomes read-dominant and, depending on the particular global memory management policy, may be freely copied.

The NUMA layer implements a migration daemon that scans local mappings. The rate at which the daemon runs is determined by the global memory management policy implemented by the UMA layer.[3] The daemon examines all local mappings to shared pages that are read-write. A software modify bit in the core map is maintained based on the hardware modify bits in local page table entries that map the physical frame. The UMA layer is notified when a software modify bit has not been set since the last scan of the migration daemon. When the UMA layer has collected this information from all mappings for an UMA page, the page becomes read-dominant and subject to replication. Copies of a read-dominant page are mapped readonly. Subsequent modifications to the page result in a page fault, which can be used to signal a transition from read-dominant to read-write status.

Our current simple policy for page migration and replication is described in Figure 2. Pages can be either replicatable, mapped but not replicatable, or not mapped anywhere. Prior to the first write, a page is replicatable. All copies of a read-write page are initially mapped readonly to catch the first modification to the page. We maintain consistency by invalidating all copies when a page is written. A page that has been written recently cannot be replicated.

[3] Migration daemons do not execute at all if the policy does not require them.

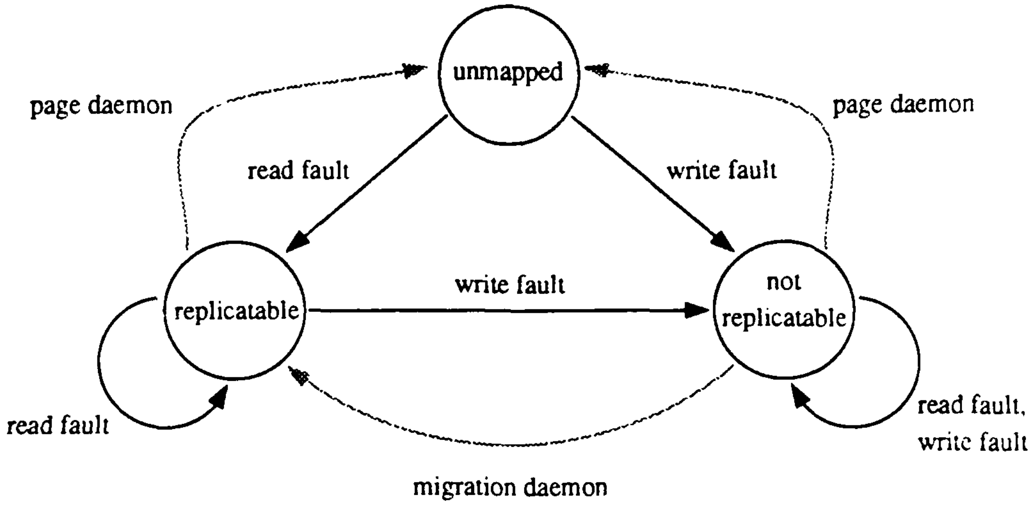

Figure 2: NUMA Policy State Diagram

The first write fault to a page causes copies of the page to coalesce at the location of the fault. Migration is not performed explicitly by our simple policy, but does occur implicitly as a result of the page replacement policy. When a page is removed from local memory, it is placed in nonlocal memory close to where it is being used, effectively migrating a read-write page.

Automatic page placement and virtual memory decisions often must interact. The page replication rate that results from the global memory management policy determines how readily physical memory is devoted to replicated pages, which in turn affects the pageout rate. The pageout daemon uses a feedback loop to control this affect. The replication rate is determined by the maximum number of copies allowed per page and the percentage of physical memory that may be devoted to replicated pages. When pageout occurs frequently, the replication rate is reduced, causing the global policy to favor migration and nonlocal references. As a result, we increase the potential for nonlocal memory references to reduce the number of I/O operations.

The global page daemon can also choose to coalesce copies to reclaim memory, providing a form of negative feedback to the page replication policy. Copies that go unused locally are discovered by the local Clock daemon and freed by the global page daemon. Future references cause page faults, which trigger a new decision on page placement.

6. Related Work

Several recent projects, including Mach [18], Symunix II [9], and Choices [6], have addressed various aspects of the memory management problem for multiprocessors. None of these systems, however, has attempted to address the full range of issues in the scope of a single system. Mach emphasizes portability, Symunix II focuses on scalability, and Choices is designed for modularity. We know of no system other than Psyche that implements an integrated solution to virtual memory management and NUMA memory management for large-scale, NUMA multiprocessors.

Mach was the first operating system to structure memory management into machine-dependent and machine-independent modules. The design of our NUMA layer was heavily influenced by the Mach pmap interface. However, we have directly addressed issues of scaling and NUMA management, which have not been a focus of the Mach development. The Mach pmap module exports a single uniform memory model; our NUMA layer exports a physical memory hierarchy, which is managed by the UMA layer. A Mach implementation for a NUMA multiprocessor is expected to provide a uniform memory abstraction within the machine-dependent module. Mach implementations on the RP3 [5] and ACE multiprocessor workstation [4] exploit global memory to implement the uniform memory model; page allocation and reclamation are performed in the machine-independent portion of the kernel without regard to local policies. Our approach allows us to separate the management of a multi-level memory hierarchy from the machine-dependent portion of the kernel and to separate local and global memory management policies.

Symunix II, the Ultracomputer version of Unix, has goals similar to ours, but takes a different approach. Symunix II does not implement standard demand paging: a process cannot run unless there is sufficient memory for the entire process. Unlike Psyche, Symunix II does not automatically cache data in local memory; instead, a cache interface is exported to the user. To avoid the costs of maintaining TLB consistency, Symunix II does not replace pages that are mapped into multiple address spaces. Finally, the interaction between demand paging and NUMA memory management is not addressed in Symunix II.

Choices implements a family of operating systems by encapsulating memory management abstractions into a class hierarchy of objects. Memory policies can be tailored to individual memory objects and architectures. Choices uses local page replacement schemes, and has not explored the interaction between global memory management and NUMA memory management. Choices has been implemented on an UMA multiprocessor and is now being ported to a NORMA (No Remote Memory Access) multiprocessor (the Intel Hypercube); the particular problems presented by a NUMA multiprocessor have not been addressed.

Our work is complementary to the experimental Platinum kernel [11] in use at the University of Rochester. Platinum is not intended to be a complete operating system; it is a platform for investigating nonuniform memory management. Platinum implements an abstraction called *coherent memory*, which is implemented in software as an extension of a directory-based caching mechanism using invalidation. Like the Psyche UMA layer, Platinum attempts to migrate and replicate pages close to processors that use them. Unlike Psyche, major portions of the Platinum kernel reside in coherent memory. Platinum is being used to empirically evaluate various migration and replication policies; we plan to incorporate the results of this evaluation into our UMA layer implementation.

Our work can also exploit the results of simulation and policy studies for NUMA management performed at Duke [12, 13]. The simulation studies use software page tables and software address translation to study the effects of reference history, page size, and page table locking on NUMA management schemes. The policy studies have considered static solutions to the page placement problem when working sets do not fit in local memory. Both of these studies have considered isolated aspects of the memory management problem for NUMA multiprocessors, while we have attempted an integrated solution.

7. Conclusions

Large-scale, shared-memory multiprocessors have a multi-level memory hierarchy that introduces new problems for the memory management system. Virtual memory can be used to migrate pages between main memory and backing store. NUMA memory management can be used to migrate or replicate pages into local memory. These solutions should be developed in tandem, since virtual memory and NUMA memory management will often interact.

We have described the Psyche memory management system, which address both issues and provides an integrated solution. In Psyche, policies for page replacement and page replication are carefully connected. System overhead is reduced by having a single daemon support both page replacement decisions and page placement decisions. Reference information used to guide page replacement is used to guide page placement. Policies and mechanisms are separated into layers that focus on different layers in the memory hierarchy. As a result, our implementation is portable and scalable.

References

1. BBN Advanced Computers Inc., Inside the Butterfly Plus, Oct 1987.

2. O. Babaoglu and W. Joy, "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits," *Proc. 8th ACM Symp. on Operating System Principles*, Pacific Grove, CA, Dec 1981, pp. 78-86.

3. D. Black, R. Rashid, D. Golub, C. Hill and R. Baron, "Translation Lookaside Buffer Consistency: A Software Approach," *Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Apr 1989.

4. W. J. Bolosky, M. L. Scott and R. P. Fitzgerald, "Simple But Effective Techniques for NUMA Memory Management," submitted to *12th ACM Symp. on Operating Systems Principles*, Mar 1989.

5. R. Bryant, *Private Communication*, RP3 Group, IBM T.J. Watson Research Center, Mar 1989.

6. R. Campbell, V. Russo and G. Johnston, "A Class Hierarchical Object-Oriented Approach to Virtual Memory Management in Multiprocessor Operating Systems," TR UIUCDCS-R-88-1459, Department of Computer Science, University of Illinois at Urbana-Champaign, Sept 1988.

7. R. Carr and J. Hennessy, "WSClock - A Simple and Effective Algorithm for Virtual Memory Management," *Proc. 8th ACM Symp. on Operating System Principles*, Pacific Grove, CA, Dec 1981, pp. 87-95.

8. F. J. Corbato, "A Paging Experiment with the Multics System," MIT Project MAC Report MAC-M-384, May 1968.

9. J. Edler, J. Lipkis and E. Schonberg, "Memory Management in Symunix II: A Design for Large-Scale Shared Memory Multiprocessors," Ultracomputer Note #135, Ultracomputer Research Laboratory, New York University, Apr 1988.

10. R. Fitzgerald and R. F. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems 4*, 2 (May 1986), pp. 147-177.

11. R. J. Fowler and A. L. Cox, "An Overview of PLATINUM: A Platform for Investigating Non-Uniform Memory," TR 262, Department of Computer Science, University of Rochester, Nov 1988.

12. M. A. Holliday, "Reference History, Page Size, and Migration Daemons in Local/Remote Architectures," *Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Apr 1989.

13. R. P. LaRowe and C. S. Ellis, "Virtual Page Placement Policies for NUMA Multiprocessors," Technical Report, Department of Computer Science, Duke University, Dec 1988.

14. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf, "The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in*

Communications SAC-1 5 (Nov 1983), pp. 842-857.

15. S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.

16. K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," YALEU/International Conference on Distributed Computing Systems/RR-492 (Ph.D. Thesis), Department of Computer Science, Yale University, Sept 1986.

17. G. R. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. 1985 International Conference on Parallel Processing*, St. Charles, IL, Aug 1985, pp. 764-771.

18. 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," *Proc. of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, Oct 1987, pp. 31-41.

19. M. L. Scott, T. J. LeBlanc and B. D. Marsh, "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors," submitted to *12th ACM Symp. on Operating Systems Principles*, Mar 1989.

20. P. J. Teller, R. Kenner and M. Snir, "TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors," *Proc. Twenty-First Annual Hawaii International Conference on System Science*, Kailua-Kona, HI, Jan 1988, pp. 184-192.

21. 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," *Proc. 11th ACM Symp. on Operating System Principles*, Austin, TX, Nov 1987, pp. 63-76.