# Experimental Comparison of Memory Management Policies for NUMA Multiprocessors*

Richard P. LaRowe Jr.
Carla Schlatter Ellis

Department of Computer Science
Duke University
Durham, NC 27706

Technical Report CS-1990-10

April 1990

**Abstract**

Non-uniformity of memory access is an almost inevitable feature of the memory architecture in shared memory multiprocessor designs that can scale to large numbers of processors. One implication of NUMA architectures is that the placement and movement of code and data become crucial to performance. As memory architectures become more complex and the non-uniformity becomes less well hidden, system software must assume a larger role in providing memory management support for the programmer. This paper investigates the role of the operating system.

We take an experimental approach to evaluating a wide-range of memory management policies. The target NUMA environment is BBN's GP-1000 multiprocessor. Extensive local modifications have been made to the memory management subsystem of BBN's nX operating system to support multiple policy implementations. Policy comparisons are based on the measured performance of real parallel applications.

Our results show that there are memory management policies implemented in our system that can improve the performance of programs written using the simpler uniform memory access (UMA) programming model. While achieving the level of performance of a highly tuned NUMA program is still a difficult problem, some examples come close. There appears to be no single policy that can be considered the best over our set of test applications. Investigations into the contributions made by individual policy features toward overall behavior of the workload provides some insight into the design of a set of effective policies.

*This research was supported in part by NSF grants CCR-8721781 and CCR-8821809.

# Contents

**1 Introduction** **3**
1.1 The NUMA Problem . . . 3
1.2 Page Placement Issues . . . 4
1.3 Related Work . . . 6
1.4 Organization of this Paper . . . 8

**2 DUnX Memory Management Mechanism** **8**
2.1 Dynamic User-Level Policy Selection . . . 8
2.2 Virtual Memory Subsystem Data Structures . . . 9
2.3 The DUnX Page Fault Handler . . . 10
2.4 The DUnX Page Scanners . . . 11
2.5 The DUnX Policy Module . . . 13

**3 DUnX Memory Management Policies** **13**
3.1 Definitions . . . 13
3.2 Fault-Time Decisions . . . 14
3.3 Page Scanner Decisions . . . 16

**4 Experiments** **18**
4.1 Information Gathering . . . 18
4.2 Workload . . . 20
4.3 Basic Operation Timings . . . 25

**5 Discussion of Experimental Results** **29**
5.1 Effectiveness of Dynamic Page Placement . . . 31
5.2 The Effects of Page Replication . . . 33
5.3 Push vs. Pull Based Page Migration . . . 36
5.4 Constant-Space vs. Linear-Space Push-Migrate Policies . . . 38
5.5 Migration Control Methods . . . 40
5.6 Replication Control Strategies . . . 44
5.7 Comparison of Triggering Tests . . . 48
5.8 Comparison of Triggering Methods . . . 51
5.9 Page Scanner Effects . . . 54

**6 Conclusions** **59**

# 1 Introduction

## 1.1 The NUMA Problem

Non-uniformity of memory access is an almost inevitable feature of the memory architecture in shared memory multiprocessor designs that can scale to large numbers of processors. Distributing the shared memory of a large scale multiprocessor among the processors so that each location can be considered close to some processor(s) while being distant from others has significant price/performance advantages. Examples of current NUMA (Non-Uniform Memory Access time) machines include the BBN Butterfly family [4, 6, 8], Cedar at the University of Illinois [22, 42], the IBM RP3 [35], and Cm* and PLUS at Carnegie-Mellon University [39, 26, 11].

One implication of this architectural design decision is that the placement and movement of code and data become crucial to performance. Requiring the applications programmer to explicitly manage the locality issues in the execution of a program represents a significant burden. In addition to designing and implementing a parallel algorithm to solve the problem at hand, the programmer also has to tune for the particular memory architecture. This involves trying to initially place shared data in the memory of the node where it is expected to be used the most, scattering data among memories to reduce contention, making copies of read-only or slowly changing data to encourage local access, worrying about the consistency of any copies made, and moving data when reference patterns are known to change during execution.

Small-scale shared memory multiprocessors generally promote a simpler UMA (Uniform Memory Access time) programming model through a memory hierarchy based on transparent caches with some hardware-supported consistency protocol. There are several recent proposals for scalable shared memory architectures that build upon the UMA approach. The VMP-MC [17] system and the Encore Ultramax architecture proposed by Wilson [41] are based on hierarchical busses and caches. The Multicube project [23] also attempts to extend consistent caches into a more scalable configuration of processors. However, the potential performance gains from making concessions to the actual non-uniformity of the memory architecture become more evident as these systems grow. For example, the benefits of taking advantage of locality in the Ultramax cluster architecture are acknowledged in [41]. Lessons learned in the explicitly NUMA environments are likely to become increasingly relevant in all scalable shared memory designs.

As memory architectures become more complex and the non-uniformity becomes less well hidden, system software must assume a larger role in providing memory management support for the programmer. Our approach is to investigate the role of the operating system. The focus is on the set of policies that determine the physical placement of pages from the virtual address space of a parallel program, including dynamic migration and replication. This paper addresses the questions of whether operating systems level memory management policies can deliver performance for an untuned program that approaches the performance of a tuned NUMA program, and which of those policies are most effective.

We take an experimental approach to evaluating a wide-range of memory management policies. The target environment is BBN's GP-1000 multiprocessor (a member of the Butterfly family of machines) which is a NUMA architecture with a 15:1 remote to local access time ratio (in the absence of contention factors). All memory in the GP-1000 is distributed among the processor nodes (i.e. there is no separate global memory). Thus, each memory location is physically local to some processor, although globally accessible by any processor. All remote memories are equally distant across a multistage switching network. Extensive modifications have been made to the memory management subsystem of BBN's nX operating system (a derivative of Mach) to support multiple policy implementations while preserving the interface expected by existing GP-1000 programs. Our local kernel is called DUnX (Duke University nX — pronounced "ducks"[1]). Policy comparisons are based on the measured performance of a workload consisting of real applications solicited from various user communities.

Admittedly, architectural enhancements and compiler assistance are also important components of a complete solution to the NUMA problem. However, exploring the capabilities and limitations of the operating system to address the placement issue at run-time with currently available hardware and language support can provide valuable insight into what else the architecture or compiler should provide.

### 1.2 Page Placement Issues

In the virtual memory system of a NUMA multiprocessor, the question that the page placement policy must answer is, "in which frame(s) of physical memory should a particular virtual page reside?" The primary purpose of this section is to develop a framework to simplify discussion and analysis of the many different policy choices possible. We establish several criteria by which a placement policy can be characterized and define subpolicies that make it easier to talk about particular aspects of the behavior generated by a policy.

A page placement policy can either be *static* or *dynamic*. Once a static policy decides to place a page in a particular frame that decision is never reevaluated. The page remains in the same frame until it is selected for removal by the page replacement policy. A dynamic page placement policy, on the other hand, periodically reevaluates placement decisions and moves pages from one physical frame to another.

Page placement policies can also be characterized by the way in which frame selection is handled. *Fixed-node* policies fix the processor node on which a page can be placed. *Preferred-nodes* policies are designed to encourage placement of a page on a particular subset of the processor nodes, but allow placement on other nodes in some cases. Finally, *random-node* policies place pages in frames on any node without concern for page locality.

Still another way to characterize a page placement policy is by the number of physical copies of a single virtual page that it allows. In general, there are *single-copy* and *multiple-copy* policies. A policy that allows multiple copies of a single virtual page is given the additional task of ensuring that those copies remain consistent (i.e. the same data values are seen regardless of which copy is accessed) or exhibit only well-defined forms of inconsistency.

It is useful to think of a page placement policy as comprised of three subpolicies, each responsible for one aspect of the placement decision. The *frame selection* policy is responsible for selecting physical memory frames to store virtual pages. It is this subpolicy that determines whether a page placement policy is a fixed-node, preferred-nodes, or random-node policy. The *page replication* policy determines if and when multiple physical copies of a virtual page are allowed. A single-copy page placement policy uses the trivial "never replicate" replication policy. Any non-trivial replication policy requires a *page coherence* policy. The coherence policy is responsible for ensuring that all physical copies of a virtual page contain consistent data. The reevaluation of earlier placement decisions and subsequent movement of pages from one physical frame to another is controlled by the *page migration* policy. Since a static page placement policy never reevaluates its decisions, it uses the trivial "never migrate" migration policy.

Note the distinction being made between the page replication and page migration subpolicies. This formulation of the policies seems to capture the issues better than the view of migration as

[1]The "n" is silent. Ducks migrate. The alternative pronunciation, "dunks", sounds like basketball and there is enough of that at Duke already.

a degenerate form of replication (i.e., replicate and immediately invalidate the old copy). It also encompasses a wider range of placement strategies.

It is also useful to characterize a page placement policy by the information it uses to make policy decisions. The *source*, *type*, and *amount* of the information are all important characteristics. Possible information sources include hardware reference bits or counters, usage counters obtained through software aging hardware reference bits, and hints or directives from the applications programmer or compiler. Page faults themselves are important information sources. The type of information used to make policy decisions is either *local* or *global*. A local policy uses only local information (e.g. local memory reference counts or the fact that a page fault has occurred) when making policy decisions. The processor on which the decision is made has no information about memory usage patterns on the other processor nodes. Global policies, on the other hand, have some system-wide information available (e.g. reference counts to a page for each processor or knowledge about page faulting behavior on other processor nodes). It is also important to consider the amount of information required by a policy, and how that amount grows as the number of processors increases. For example, a global policy that relies on per-processor usage counts for each page has space requirements that grow linearly with the number of processors. We call this a *linear-space* policy. On the other hand, a global policy that summarizes remote reference histories with a single counter has constant space and time requirements. Clearly, the latter policy (which we call a *constant-space* policy) is more scalable than the former.

A policy may restrict the type of page (e.g. code, private data, shared data, etc.) it is willing to replicate, or the type of data it is willing to migrate from one physical frame to another. Similarly, some policies may require that certain types of pages always be replicated. Technically, this is yet another kind of information that the policy may use to make decisions. It may be convenient to view these semantically based placement decisions as restrictions upon a general policy.

Dynamic page placement policies can be further characterized by the way in which placement decisions are reevaluated. In general, a policy can be *pull*, *push*, or *put* based, although it is easy to develop hybrid strategies. A pull-based page placement policy is one in which processors tend to pull pages towards themselves. Such a policy decides that a copy (perhaps the only copy) of a given page should reside in a local frame, and then pulls the page over. Push-based placement policies work in nearly the opposite way. A processor that has a copy of a page decides when a copy of that page should reside in a frame on some other node. The policy may migrate its own copy or create another on the destination processor node. When policy decisions may be made by a third processor, it is a put-based policy. A put-based policy would likely be implemented as a special daemon whose job is to reevaluate page placement decisions. It is difficult to imagine any reasonable put-based policy that doesn't require global linear-space information, so the feasibility of such policies is questionable.

Hybrid policies are easily developed using a *trigger* mechanism. In such a policy, the decision making task is broken into two subtasks. One subtask, the trigger, decides when the reevaluation of a placement decision is necessary. The other subtask responds to the trigger by actually reevaluating the page placement. For example, a push-based trigger might work by invalidating remote mappings to its copy of some page, leaving the actual pull-based decision to the remote processes that later fault on the page.

The terminology and organization developed in this section serve as our basis for policy descriptions. We often talk collectively of groups of page placement policies, such as static single-copy fixed-node strategies, or dynamic preferred-nodes policies with pull-based page replication. We may also speak of replication or migration subpolicies collectively in terms of the information they use when making decisions.

### 1.3 Related Work

The question of how to best manage the memory of MIMD multiprocessors is an area of active research. Relevant work includes the study of cache management strategies for UMA shared memory multiprocessors, the implementation and study of shared memory abstractions on distributed memory machines or loosely-connected networks of workstations, and of course, research on NUMA-specific memory management problems.

The management of caches in shared memory systems has received much attention. The hardware caching literature is the primary source of information on memory coherence strategies for ensuring the consistency of cached shared data [1, 2, 21, 38]. Trace-driven simulation and analytic methods have been used to compare different schemes. The workload models discussed in this literature are relevant to our work, since one goal is to make it possible to efficiently run UMA programs on a NUMA architecture. Since the local memory of a NUMA machine can be thought of as cache space for some global (distributed) shared memory, this body of work is clearly relevant. Key differences lie in the size of the unit of coherence (cache line vs. page size), the speed at which different coherence operations are possible (hardware vs. software implementations), and the ability (in a NUMA architecture) to directly access remote memory. Software-controlled hardware caching with a large cache line size has been studied in the context of the VMP multiprocessor both experimentally and with simulation [18].

The idea of providing a shared memory abstraction on distributed memory, message-passing architectures in software has been studied by several groups. Kai Li's work [32, 33], based on page-granularity caching (pull-based copying of pages between memories) with a directory-based invalidation coherency scheme, provides a complete shared memory abstraction to the programmer. Clouds [36] is an object-oriented system that provides a shared memory abstraction on a network of workstations, but attempts to minimize the number of invalidation messages that result from active write sharing of data (e.g. synchronization data). All of this work is important because a NUMA machine can be viewed as a distributed memory machine with the added ability to reference remote memories. The results of this work can be directly applied to the NUMA multiprocessor domain, but it is expected that the ability to directly reference remote memory can be exploited to further improve performance.

The Munin system [10] is another system that provides a global shared memory abstraction on a distributed memory system. A novel feature of Munin is that it uses (potentially) different memory coherence strategies for each shared memory object in an application. The authors argue [9] that different memory objects are referenced in different ways, and as such the coherence policies should differ. Policy selection is currently done by the programmer, but at least partial control from the compiler as well as automatic policy selection by Munin itself is envisioned. The idea of having different policies for different memory objects extends to the NUMA multiprocessor domain as well.

There has been a considerable amount of work related specifically to memory management for NUMA shared memory multiprocessors.

Black, Gupta, and Weber introduce provably competitive migration and replication policies for NUMA architectures in [13]. Unfortunately, their algorithms require hardware support (a reference counter for each frame-processor combination) not available in most existing machines. Such hardware requirements present a problem of scalability as the number of processors grows. The PLUS system currently being developed at CMU [12, 11] is a NUMA shared memory multiprocessor with such hardware support for page replication. PLUS uses *weak ordering* and a write-through memory coherence policy to ensure correctness. Emphasis of the PLUS project has been on hardware support for page replication and coherency maintenace, as opposed to replication control policies.

Page migration based on page *pivoting* has been studied by Scheurich and Dubois [37]. Reference counters for each dimension of the communications network are used to pull pages towards the nodes referencing them. While this algorithm (with appropriate hardware support) lends itself nicely to some connection networks (e.g. meshes or hypercube interconnects), it isn't as appropriate for multi-level switch interconnects such as the one used on BBN Butterfly systems.

Holliday studied mechanisms for doing migration of shared data pages on a Butterfly-type architecture [25]. His study concentrated on issues such as page size, reference bit aging, the number of special migration daemons needed, and the sleep delays between daemon runs. In a related paper [24], he discussed the management and organization of the page tables used to support migration.

Fowler and Cox have been studying NUMA memory management on a Butterfly Plus multiprocessor using their Platinum operating system [19]. The approach they use is based on an extended software caching scheme with a directory-based invalidation coherency protocol. In an attempt to avoid frequent invalidations and excessive migration (page bouncing), pages are *frozen* in memory when they are invalidated too frequently. A daemon runs periodically to *defrost* those pages, so that the system can react to a changing reference pattern.

The port of the Mach operating system to the IBM ACE multiprocessor by Bolosky, Scott, and Fitzgerald is discussed in [14]. The ACE memory hierarchy consists of memory modules local to each processor (but accessible by any processor) as well as a large global memory. The approach taken uses the local memory modules as cache space for the global memory. Read-only pages may be replicated in local memories. Writable pages may initially be moved from one local memory to another, but when a threshold on the number of migrations is reached, the page is considered actively write-shared and ends up being permanently placed in the global memory to avoid page bouncing.

The Psyche operating system being developed by LeBlanc, Scott, and Marsh separates NUMA memory management into several layers [31]. Their design separates node-level physical memory management (the NUMA layer) from the higher-level management that controls the placement of shared data (the UMA layer) and from the main / secondary storage virtual memory management (the VUMA level). The main issues related to memory management that have been addressed by the Psyche project, so far, have been the layered structure of the memory management system and its implications.

The Xylem operating system [34] developed for the Cedar multiprocessor uses programmer-defined memory attributes to provide mechanism by which the applications programmer can control software caching like strategies. The operating system does little in the way of automatic memory management, but does provide a flexible selection of hooks to aid the applications programmer.

Mats Brorsson has studied page placement policies for the IBM RP3 [16]. His emphasis, however, has been on the placement of code and private data pages, which are clearly best placed in local memory. Each of the policies he considers keeps shared data pages in the RP3 global (interleaved) memory, regardless of the reference patterns to those pages.

Two important features distinguish our work from the work of others. We place a heavy emphasis on policy comparisons. Rather than implementing one or two similar policies and discussing their virtues, we implement a large number of policies so that quantitative comparisons can be made. These include (at least approximations of) policies proposed in cited related work for which hardware support is adequate. We then analyze and study the quantitative results, in order to come to a qualitative understanding of why particular policies perform as they do under different workloads. We believe that our work will lead to a greater understanding of the issues involved in managing the memory of a NUMA multiprocessor.

The other distinguishing feature of our work is that we treat page placement as a single problem. Rather than separating the problem of managing the secondary-main memory hierarchy from that of managing the local-remote memory hierarchy, we consider it a single problem. We view page replication and migration as a part of the page placement policy, rather than as distinct problems.

### 1.4 Organization of this Paper

The next section of this paper describes the mechanisms provided in the memory management subsystem of the DUnX kernel that makes it an effective platform for the investigation of a wide range of NUMA page placement policies. This is followed, in Section 3, by a discussion of how policies can be composed in DUnX and the subset of policies under consideration in this study. Section 4 presents our experimental method, including the workload and instrumentation facilities built into the system. Then the results from an extensive set of experiments are presented. Finally, we summarize.

## 2 DUnX Memory Management Mechanism

The DUnX operating system is derived from BBN's nX operating system. The principal differences lie in the addition (in DUnX) of support for dynamic multiple-copy page placement (page migration and page replication with assured data consistency) and dynamic user-level control of memory management policy.

We begin the section with a discussion of dynamic user-level memory management policy selection. The following subsections discuss the underlying data structures in the virtual memory subsystem, the page fault handler, and the page scanners. Emphasis in these latter subsections is on the mechanisms used to support a variety of different page placement strategies. The final subsection describes the organization of the policy decision-making module.

Keep in mind that only mechanism is discussed in this section. The actual memory management policies considered are described in section 3.

### 2.1 Dynamic User-Level Policy Selection

A conventional way for multiple users to share the resources of a GP-1000 machine is to assign each user a dedicated subset of the processor nodes (called a *cluster*). DUnX expands upon this by providing users with the ability to specify and dynamically change the memory management policies used in each cluster of the machine. This feature serves two useful purposes.

First, dynamic user-level policy selection makes policy comparison a far simpler task. It is a key feature in making the DUnX kernel a conducive environment for experimental evaluation of policy alternatives. A single shell script can create a cluster in which to run a suite of tests, and then run an application several times, changing the memory management strategies used each time. No system rebooting is required, and other users of the machine are unaffected (as far as policy is concerned) by the ongoing tests.

We also claim that dynamic user-level policy selection is a useful feature to have in a production operating system. Our results (presented in Section 5) indicate that no single policy works best in all cases. Until some perfect policy is developed, we believe it is desirable to provide a variety of options from which the user or programmer can choose.

On each processor node, the current memory management attributes for the cluster to which the node currently belongs are recorded in the local element of the `my_cluster_mmattr` array (the array is distributed across the processor memories so that one element is local to each node). The memory management attributes are changed through use of the `cluster_ctl` system call. As we shall see, policy routines in the virtual memory management subsystem use the contents of `my_cluster_mmattr` to drive policy decisions. At the shell-level, users control the memory management attributes through use of the `cluster` and/or `clusterctl` commands.

### 2.2 Virtual Memory Subsystem Data Structures

The overall structure of the nX virtual memory subsystem, described in [3], provides an attractive basis for our modifications designed to support the goals of the DUnX virtual memory (VM) subsystem.

The five central data structures in the VM subsystem are: per-process address *maps* comprised of *map entries*, which describe the address space of a process; memory *objects*, which describe a region of (potentially shared) memory; *frame table entries*, each representing a physical page frame available on the machine;[2] and *pmaps*, the physical page tables for some process. For our purposes, memory objects and memory frames are of primary concern. More detailed descriptions of the other data structures can be found in [3].

A memory object describes a region of abstract memory. The object data structure contains the information needed to find resident pages of the object as well as pages on backing store. A memory object may be shared by multiple processes (referenced by multiple address maps), and/or mapped into different virtual addresses within the same or different address maps. Each memory object has an associated *page node*, which indicates the processor node considered to be "best" for storing pages of the object. The page node field can be set by the applications programmer or compiler, but its default value indicates that pages of the object should initially be placed on whichever node faults them into main memory. Memory objects also contain a flag indicating whether or not movement (replication or migration) of its pages from frame to frame is allowed. We suspect that more extensive policy control at the object level may be useful, although we have not yet identified the appropriate constructs and implemented them in the DUnX kernel.

The frame entry is used to describe a single physical page frame. Together, the frame entries form a distributed frame table. Each frame has a *status* that indicates how the frame is currently being used. Status possibilities important for our purposes are listed below:

- *Free* — A free frame is one that is currently not in use. The contents of the frame are uncertain.
- *Zeroed* — A zeroed frame is a free frame not in use, but is known to contain all zeroes. Since newly allocated memory is zero-filled, keeping track of such frames separately is a useful optimization.
- *Active* — An active frame is one that contains a valid virtual page thought to be in use by a process on the local processor.
- *Remactive* — A remactive (short for remotely-active) frame contains a valid virtual page that may be in use only by processes executing on remote processors.
- *Replicated* — A replicated frame contains one of at least two copies of some virtual page. The frame is assumed to be in use by a process on the local processor.
- *Inactive* — An inactive frame contains a virtual page not currently in use by any process (all virtual to physical mappings to the frame have been invalidated).

[2]In nX and Mach, frames are called resident pages. We prefer to call them by the more traditional name, as it helps eliminate confusion between virtual and resident pages. Thus, a resident page (of the virtual address space) occupies a frame.

Other status values are available to be used for other purposes, such as keeping track of frames used for kernel text or page tables.

Every frame can typically be found on several lists.[3] All of the frames holding resident pages of a memory object are linked together, a distributed object/offset hash table serves to quickly locate resident pages of a memory object, and status lists on each node serve to collect frames with similar status values. Frames holding copies of the same virtual page are linked together on still another list.

Each frame entry also contains a list of *reverse pte* entries. Each entry describes a valid pmap entry to that frame, making it easy for the VM system to determine which pmaps need to be modified when the virtual to physical mapping of a page is invalidated.

## 2.3 The DUnX Page Fault Handler

Page faults are the primary initiators of virtual memory management operations. Page placement (including replication and migration) and page replacement decisions may all be required of the page fault handler.

When a page fault occurs, a lookup in the faulting process's address map results in a pointer to the memory object to which the address belongs, as well as an offset into that memory object. The fault handler immediately acquires a free page frame that can be used to service the fault, if necessary. Acquiring the free frame early simplifies the locking protocols. If the frame is not needed to service the fault, it is simply returned to the appropriate free frame list.

Selection of a free frame is based on several factors. The preferred node from which to acquire a frame is determined by the page node field in the memory object and the memory management attributes in force within the cluster. For some policies, the preferred node is the faulting node, regardless of the page node field, and in other policies, it is defined as the page node (which often defaults to the faulting node anyway). If a frame is readily available (i.e. there is at least one frame on the free or zeroed list) on this preferred node, such a frame is acquired. If not, a frame borrowing operation [29] is considered.

A frame is borrowed from another node when four conditions are met: the memory object allows frame borrowing (this is specified by the programmer or compiler), the user-selected memory management policies for the current cluster allow frame borrowing, a thrashing situation has been detected (see [28] for a discussion of frame borrowing triggers), and there exists a node with an abundance of free frames.

If a free frame cannot be borrowed from another node, the fault handler is forced to acquire a frame on the preferred node. This operation may include the replacement of a page. The DUnX replacement policy acquires the first frame it finds by searching the frame status lists in the following order: free/zeroed (which of these two lists is examined first depends on whether or not the fault requires a zeroed page), inactive, replicated, remactive, and active. The page scanner (discussed later), which may or may not be enabled, is responsible for movement of frames among these lists. The result is an approximate per-node LRU replacement policy.

Once a free frame is acquired (just in case one is required to service the fault), the fault handler continues by searching an object/offset hash table to determine whether a copy of the desired page is already resident in main memory.

When a page fault is taken on a non-resident page, the fault is serviced either by providing a zero-filled page or by bringing in a copy of the page from backing store. In either case, the necessary adjustments to the hardware page tables are made and processing continues.

[3] Frequently, we will use phrases like "frames linked together". Technically, we mean that the frame entries corresponding to those frames are linked together.

Additional decision-making is required when a page fault is taken on a potentially shared page that is already resident in some physical frame. Three policy functions are used to make migration and replication policy decisions in the fault handler.

1. `vm_policy_fault_inactive` — This function is called whenever a page fault is taken on a page in an inactive frame to decide whether or not to migrate the page. If the function determines that migration is appropriate, the page is migrated to a local frame and the remote source frame is freed. Otherwise, the inactive frame is reactivated and used to satisfy the fault.

2. `vm_policy_fault_active` — This decision function is called whenever a fault is taken on an active or remactive frame and decides whether to migrate the page, replicate the page, or neither. If the decision is to migrate, the page is copied to a local frame and the source frame is freed (after invalidating all virtual to physical mappings to that frame). If the function decides upon replication, the page is copied to a local frame which is used to satisfy the fault. The status of both frames is changed to replicated and write access is disallowed. The remaining option is that the active or remactive frame is used to satisfy the fault.

3. `vm_policy_fault_replicate` — This function is called whenever a fault is taken on a replicated page. However, before calling the function the fault handler selects one of the replicas to best service the fault. If a local copy exists, it is the frame chosen.

   The action the fault handler takes depends not only on the return value of the decision function, but also on the type of memory access (read or write) that triggered the page fault.

   If the page fault was triggered by a write memory access, the coherence policy must be used to ensure data consistency. The DUnX coherence policy is based on a simple directory-based invalidation scheme. The frames containing copies of the page are linked together and each frame contains the information necessary to invalidate mappings to itself, so the invalidation phase is fairly straightforward. A single copy of the page survives. The policy routine chooses between using the single remaining copy of the page to satisfy the fault and migrating that copy to a local frame. In the case of migration, the original frame is freed.

   If the page fault was triggered by a read memory access, no special data consistency processing is required. Either the replica chosen by the fault handler is used to trivially satisfy the fault or an additional copy is created to satisfy the fault.

In all cases, the frame status of the frame(s) involved is updated depending on the locality of the frame used with respect to the faulting process (active or remactive status), on whether multiple copies exist (replicated status), and on whether the frame has been the source of a migration or victim of a coherency fault (free status) .

The DUnX page fault handler supports dynamic multiple-copy page placement, but all fault-time migration or replication policy decisions are left to these three policy functions. Isolating policy decisions from the bulk of the VM subsystem has been a DUnX design goal, as it greatly simplifies the task of implementing new policy options.

### 2.4 The DUnX Page Scanners

A virtual memory system for a NUMA multiprocessor that relies entirely on page faults to invoke policy operations is severely limited in the range of policy options that can be implemented. In such a system, even if it were possible to detect that a page is poorly placed, there is no way to gain control unless a page fault occurs. Thus, DUnX provides each processor node with a page scanner.

The user-selectable memory management policy determines whether the scanner on a particular node is enabled or not. When a page scanner is enabled, it runs periodically (this is guaranteed) in the `idle_proc` context.

A page scanner has several responsibilities. It maintains software reference counters by aging the hardware reference bits, it moves frames among the frame status lists to support the page replacement policies, and it performs assorted tasks to support dynamic page placement.

Each frame contains two approximate usage counts: one for references from the local processor and one for references from remote processors. One page scanner phase ages both counts by shifting the counters right one bit for each local frame. The next phase is a scan through the local pmap data structures, where hardware reference bits are checked and then cleared. If a hardware reference bit is set, a high-order bit is or'ed into the corresponding (local or remote) software reference counter. During this page scanner phase, a policy support routine (`vm_policy_count`) is called so that special policy-specific information (e.g. per-node reference counts) can be collected if desired.

The third page scanner phase runs less frequently than the other phases in order to minimize overhead. The third phase is responsible for examining each local frame and then determining whether or not the status of that frame should change. As the scanner loops through the local frames, its actions depend on the current status of the frame, the software reference counts for that frame, and the return values from policy decision-making functions.

In its role of maintaining the frame status lists, the scanner considers each frame, comparing its local reference count to predefined thresholds in order to identify pages that are accessed locally sufficiently often to be classified as active, pages that may tentatively be classified as remactive, and pages (including replicas) that appear to be locally inactive. The status of a remactive frame can be changed to inactive after all virtual to physical mappings to the frame have been invalidated. An unused replicated frame is freed after all virtual to physical mappings to it have been invalidated. Whenever a frame has been inactive for sufficiently long (the software usage counters are cleverly used to detect this occurrence) and the page in the frame is clean, the page is freed. Since the page scanner never actually performs a pageout operation (this is left to the fault handler), dirty pages remain in inactive frames until they are reactivated (as a result of a fault), replaced and written to disk by the page fault handler, or freed as a result of the termination of the processes using them.

A policy support routine (`vm_policy_scan_page`) is called once for each frame, so that policy-specific operations done on every frame regardless of status can be performed.

The `vm_policy_push_trigger` routine is called for each active frame. It may direct the page scanner to invalidate *all* virtual to physical mappings to the frame and change its status to inactive, to invalidate all *remote* virtual to physical mappings to the frame (leaving local mappings intact and the frame status active), or to do nothing to trigger additional faults. Triggering page faults by such invalidations can force reevaluation of the earlier page placement decisions.

Another policy function `vm_policy_push_migrate` is called whenever `vm_policy_push_trigger` decides not to trigger faults on an active frame and whenever a remactive frame is encountered that appears to be correctly categorized as primarily accessed from remote nodes. It determines whether or not to push the page to a free frame on some other node. If `vm_policy_push_migrate` returns a valid processor node number, the page scanner migrates the page in that frame to a free frame on the specified processor node. All mappings to the source frame are invalidated at this time. References to the page subsequently result in page faults. How the resulting page faults are handled depends entirely on the `vm_policy_fault_active` policy function, but it is expected that the recently migrated copy of the page would be used to satisfy the faults.

As with the DUnX page fault handler, the DUnX page scanners provide support for implementing dynamic page placement without making any policy-specific decisions. The scanner has the ability to push migrate pages as well as the ability to induce page faults which then trigger the reevaluation of the page placement decision.

### 2.5 The DUnX Policy Module

In the previous two subsections, seven policy functions were introduced. These functions isolate page placement policy decisions from the fault handler and the page scanners.

Decisions made by the functions are based on input parameters, node status variables (e.g. current page fault frequency or remote reference frequency), and by the current cluster memory management attributes. Assorted reference history information (e.g. per-node usage counts, invalidation histories, aged modification bits, etc.) can be maintained by the policy routines and stored in the frame data structures.

The memory management attributes (stored in `my_cluster_mmattr`) have primary control over the processing done in each policy function. The 32-bit `my_cluster_mmattr` is divided into eight four-bit nibbles. Seven of the nibbles are used to control each of the seven policy functions. The final nibble is used for more generic control over the decision making processes.

Each policy function begins by extracting its four-bit nibble of policy-control information, which is then used to select from one of sixteen possible policies. Since each policy function is controlled independently, this organization is capable of supporting over four billion different memory management strategies, selected dynamically on a per-cluster basis. We have implemented and experimented with over 40 different policies to date.

The flexible policy control mechanisms and implementation framework in DUnX has given us a way to collect information regarding shared memory usage patterns as well. These special policies are described in section 4.1.

## 3 DUnX Memory Management Policies

In this section, we introduce the collection of memory management policies available in DUnX. We take a bottom-up approach in our discussion, starting with some definitions and then a description of "policy pieces" that are combined to develop entire policies. During our discussion, we introduce the policy naming conventions that we subsequently use.

### 3.1 Definitions

As described in Section 1.2, the migration subpolicy determines when placement decisions are reevaluated and, thus, when pages may become candidates for movement or replication. Under some dynamic policies, a *frozen* page is one that temporarily cannot be moved or copied (this name is adopted from [19]). A page that isn't frozen is said to be *warm*. The act of changing the status of a warm page to frozen is called *freezing*, and the opposite change is called *warming* or *defrosting*. The rationale behind freezing pages is to avoid page bouncing in the presence of volatile access patterns.

Several of the policies under consideration rely on the page scanner invalidating the mappings to some pages in order to cause page faults which may in turn result in page migration or replication operations. The test used by the page scanner to determine when to perform such invalidation is called a *trigger*. A page fault caused in this way is called a *triggered* page fault.

The impact of different types of information about page usage is an essential aspect of our policy evaluations. Several of the page scanner triggers used by DUnX policies are based (entirely or partly) on the local and remote usage counts maintained for each frame by the page scanners. The remote usage count represents a constant-space, global measure of page usage. A *simple* test is just a single comparison of the two counts that succeeds when the remote usage count exceeds the local usage count. A more conservative version of the test succeeds only after the remote usage count has been at least as high as the local usage count for several (the default is four) page scanner iterations in a row. This latter version is called the *remote-wins* test.

Some of our test policies partly base decisions on a node's *remote reference measure*. The remote reference measures (one for each node) are maintained by the page scanners. Each is the sum over each of the local frames of the number of remote processes that referenced that frame in the past eight one-second intervals. It serves as a concise estimate of the overall amount of remote usage of local frames.

The DUnX page scanners are able to age hardware modification bits to detect phases in which writable pages are really used in a read-only fashion. This is a potentially powerful capability, since pages that aren't being modified can easily be replicated. To take advantage of this capability, some DUnX policies define a *read-only* page as one that is either never writable (the conventional definition) or one that has not been modified in the recent past (the aged modify count goes to zero).

Memory object type information helps determine whether a page is actually affected by a dynamic placement policy running in its cluster. For example, code pages are always placed locally, regardless of policy. Kernel code and data structures are never migrated, and some data are actually wired into main memory permanently. The default page node for memory objects is the faulting node, so private data is nearly always initially placed locally. Copy-on-write and copy-on-reference (from a remote node) mechanisms are employed by all of the policies to minimize access times to unshared memory inherited from the parent.

### 3.2 Fault-Time Decisions

When a page fault occurs on a page resident in a remote memory frame, some policies migrate or replicate the page to a local frame. Fault-time migrations are predominantly pull-based. The policies considered handle the situation in one of the following ways (note that all migration and replication operations are subject to destination frame availability):

1. Never migrate or replicate a page (trivial no migration and single-copy replication subpolicies). Policy variations within this class derive from the choice of fixed-node or preferred-node frame selection strategies for initial placement (our previous work [29] ruled out random-node placement for further consideration in this study).

2. Always migrate, always replicate, or always do one or the other depending on the type of memory access that caused the page fault.

3. Always migrate the page if it resides in a borrowed frame. This implies a preferred-nodes frame selection subpolicy and is called a *recall* migration.

4. Always migrate, always replicate, or always do one or the other whenever the page fault was intentionally triggered by a page scanner (a flag in the frame entry is set when a scanner attempts to trigger faults on a page). This is called a *triggered* migration or replication.

5. Always migrate, always replicate, or always do one or the other whenever the page faulted on is warm. Whenever a copy of a warm page must be invalidated (when the page is migrated or when one of multiple copies is modified), the page is frozen. This is called a *warm* migration or replication.

| Abbr. | Policy Option |
|---|---|
| N | *never migrate or replicate* |
| Ar | *always replicate* |
| Am | *always migrate* |
| Arm | *always replicate or migrate* |
| Rm | *recall migrate* |
| Tr | *triggered replicate* |
| Tm | *triggered migrate* |
| Trm | *triggered replicate or migrate* |
| Wr | *warm replicate* |
| Wm | *warm migrate* |
| Wrm | *warm replicate or migrate* |
| Pr | *paced replicate* |
| Pm | *paced migrate* |
| Prm | *paced replicate or migrate* |
| Rr | *read-only replicate* |
| Bm | *balance migrate* |

Table 1: Abbreviations for Page Fault Handler Policy Options

6. If a page is warm and it hasn't been invalidated recently (any time within the past $t_1$ milliseconds or within the interval since the page was last defrosted, whichever is smaller), migrate, replicate, or do one or the other depending on the type of memory reference that caused the fault. If the page is warm and it has been invalidated recently, the page is frozen. Since a page is frozen only if invalidations occur too frequently, we call this a *paced* migration or replication.

7. On a read reference, replicate read-only pages. Here, the definition of a read-only page is the one given in the previous subsection.

8. Migrate a page if the node on which it currently resides has a remote reference measure more than twice that on the local node. A migration performed in this way is called a *balance* migration, since the goal is to evenly distribute (i.e. balance) the actively shared pages among the processor nodes using them.

9. Some combination of the above options. While not all combinations are possible, most of the interesting combinations (e.g. read-only replicate with balance migrate and read-only replicate with warm migrate) have been implemented.

Our policy naming conventions are based on sets of abbreviations. For example, the preceding options for handling page faults on remote resident pages are abbreviated as shown in table 1. Combinations are abbreviated by concatenating the component abbreviations. For example, a policy that combines read-only replication with balance migration would be abbreviated as RrBm or BmRr. The order of the components is significant, as it specifies which of the options is considered first.

## 3.3 Page Scanner Decisions

The DUnX page scanners are essential components of the dynamic page placement algorithms. The page scanners affect dynamic page placement by triggering page faults (intentionally or otherwise) and/or push-migrating pages to different frames. The page scanners also serve an important job as data collectors, for much of the information about dynamic access patterns used by the placement policies is collected by the page scanners.

A page scanner may trigger a placement decision reevaluation (by completely inactivating a frame or by simply invalidating the remote virtual to physical mappings to the frame) or it may actually migrate a page itself. The following options are available to the scanners:

1. Never trigger or migrate anything.
2. Trigger whenever the simple test (as defined in Section 3.1) succeeds. This is called the *simple* trigger.
3. Trigger when the remote-wins test succeeds. We call this the *remote-wins* trigger.
4. Trigger whenever a frozen page is encountered. The page is defrosted at this time, so it may be migrated and/or replicated again. This is called a *timed defrost* trigger, since the decision to defrost the page is based only on the fact that the page has been frozen for some amount of time bounded by the maximum interval between successive page scans (four seconds is the expected time until defrost in the default case).
5. Trigger (and defrost) any frozen page when the simple test succeeds. This is called the *simple defrost* trigger.
6. Trigger (and defrost) any frozen page for which the remote-wins test succeeds. This is called the *remote-wins defrost* trigger.
7. Trigger any frozen page that has been frozen for a sufficiently long time (currently defined as sixteen page scanner runs). This is called the *long time defrost* trigger.
8. Trigger whenever a writable page appears to have become read-only (i.e. when the aged modify count goes to zero). We call this the *read-only* trigger.
9. Push migrate a page whenever the simple test succeeds. The page is pushed to the remote node that last updated the remote usage count. This is called the *simple push* migrate.
10. Push migrate a page whenever the remote-wins test succeeds. The page is pushed to the remote node that last updated the remote usage count. This is called a *remote-wins push* migrate.
11. Push migrate whenever a per-node counter associated with the frame exceeds some threshold. The per-node counters are maintained by the page scanners as follows: whenever a reference to a page is detected (when the scanners look at the hardware page tables), the counter for that node on that frame is incremented, and some other node's counter for that frame is decremented. This is a rough approximation to the competitive page migration algorithm presented in [13]. Though our approximation is not itself competitive, we call this a *competitive push* migrate.

| Abbr. | Policy Option |
|---|---|
| N | *never trigger or migrate* |
| Si | *simple inactivate trigger* |
| St | *simple remote invalidate trigger* |
| Ri | *remote-wins inactivate trigger* |
| Rt | *remote-wins remote invalidate trigger* |
| TDi | *timed defrost inactivate* |
| TDt | *timed defrost trigger* |
| SDt | *simple defrost trigger* |
| RDt | *remote-wins defrost trigger* |
| LTDt | *long time defrost trigger* |
| ROt | *read-only trigger* |
| Sp | *simple push migrate* |
| Rp | *remote-wins push migrate* |
| Cp | *approximate competitive push migrate* |

Table 2: Abbreviations for Page Scanner Policy Options

12. Some combination of the above options. As with the fault-time options, only the more reasonable possibilities (e.g. a combination simple defrost and read-only trigger) have been implemented in DUnX.

The page scanner options are abbreviated as shown in table 2. As with the page fault options, combinations are abbreviated by concatenating the component abbreviations.

Names for complete DUnX page placement policies consist of three required fields (a hyphen is used to separate fields) and one optional field. The first field specifies how a fault on a page in an inactive frame is handled and the second indicates how a fault on any other resident page is handled. It is common for a policy to handle these two cases differently, since an inactive frame is assumed to be not in use on its local node. Each of the first two fields takes on values from table 1. The third required field describes the behavior of the page scanners. Legal values are obtained from table 2. The optional fourth field is used for *ad hoc* specifiers. Possible values include NB (indicating that the policy does not allow frame borrowing), NS (indicating that the policy does not use the page scanners), and NR (indicating that the page scanners do not do ordinary page replacement).

Examples of valid policy names include N-N-N-NBNS (the default DUnX policy which never migrates or replicates and uses fixed-node frame selection), Am-Ar-N (a simple pull-based caching strategy), Am-Rr-ROt (a cache read-only policy), Wm-Wrm-RDt (a more complicated strategy using freezing and a global usage trigger), and many others. Groups or classes of policies are specified with the help of the wildcard specifier x. For example, x-Wrm-x refers to all policies that replicate or migrate warm pages and x-x-xp refers to all policies that use push migration.

By combining the policy pieces, we can create an extremely large number of page placement policies. In effect, there are millions of policies *already implemented* in the DUnX kernel. Of course, it is completely unreasonable to study each of them, so we have concentrated our efforts on combinations that intuitively make sense and combinations that provide insight into the issues at hand.

In table 3 we show several common policy groups, as well as examples of each and a short English description of the group. Most of the more important policies we've considered appear in the table.

# 4 Experiments

## 4.1 Information Gathering

DUnX has been developed to serve as a testbed for experimental evaluation of different NUMA memory management strategies. Because of its role in experimentation, the capability to gather performance information is extremely important. In this subsection, we discuss the set of information gathering tools and procedures we use to study policy behavior.

The primary measure of performance is elapsed time. This "bottom line" measure provides the answer to the simple question, "which policy delivers the best performance for this application?"

The DUnX virtual memory system maintains a large collection of per-node and per-process event counts. Counted events include such things as page faults, separate counts for different types of page faults (e.g. zero-fill, pagein, protection, translation, copy-on-write, coherency, etc.), page migration and replication operations, and frame borrow operations. The DUnX `vm_mapstat` system call is used to extract this information from the kernel and a wide range of user-level tools based on the system call have been developed. Using the data collection tools, we are able to gather summary information on a per-process, per-node, or per-cluster basis as well as dynamically monitor per-process, per-node, or per-cluster behavior. Information of this sort helps to explain the observed completion time performance differences of different memory management strategies.

The DUnX kernel also supports the BBN nX kernel event logging facility. Similar to BBN's user-level `ELOG` (event logging) package [7], the kernel event logging facility provides the ability to log events (a four-tuple consisting of a node number, an event code, a timestamp, and a 32-bit data value) from kernel code. There are over one hundred and forty events currently defined in the DUnX kernel, grouped into several event classes (e.g. page fault events, scheduler events, tlb events, etc.). A user-level program is used to enable and disable logging of a specified set of kernel event classes. Since the granularity of events is fairly small, event logs can grow incredibly large in a very short period of time. Thus, although kernel event logs provide very detailed information, they are not practical as a primary means of data collection simply because of disk space limitations.

There are several ways that kernel event logs can be analyzed. BBN's `gist` program has proven quite valuable. `Gist` is an interactive tool that can graphically display per-node event timelines, making it easy to see when an event occurred relative to other events (possibly on other processor nodes). Another useful feature of `gist` is the ability to define "states". A processor is considered to be in a particular state from the time a specified entry event occurs until the time the corresponding exit event occurs. `Gist` can plot histograms of state duration times (with assorted statistics calculated automatically), making it easy to determine how long certain operations (e.g. page faults, page migrations, etc.) typically take. `Gist` is also able to plot bar charts showing the percentage of the total trace time spent in a particular state. Using this feature, we are able to determine things such as the overhead spent running the page scanners for a particular application under a particular memory management strategy. Several additional tools have been developed at Duke that make it easy to use typical Unix tools (e.g. `awk` and `sort`) to analyze event logs. However, the very large volume of data produced from kernel event logging limits the utility of these logs by requiring a considerable amount of effort for analyzing them.

The flexible policy control mechanism in DUnX has been exploited to provide two ways of collecting information about the memory reference patterns of different applications. The first is a

| Group | Examples | Description |
|---|---|---|
| N-N-N | N-N-N-NSNB<br>N-N-N-NS<br>N-N-N-NB | base-case static policies |
| Rm-x-x | Rm-N-N | recall migrate policies |
| Am-N-x | Am-N-N<br>Am-N-Si<br>Am-N-Ri | pull inactive migrate policies |
| Am-Am-x | Am-Am-N | migrate on reference |
| Am-Tm-x | Am-Tm-St<br>Am-Tm-Rt | triggered migrate policies |
| Am-Bm-x | Am-Bm-St<br>Am-Bm-Rt | balance migrate policies |
| Wm-Wm-x | Wm-Wm-TDi<br>Wm-Wm-TDt<br>Wm-Wm-SDt<br>Wm-Wm-RDt<br>Wm-Wm-LTDt | warm migrate policies |
| Pm-Pm-x | Pm-Pm-SDt<br>Pm-Pm-RDt<br>Pm-Pm-TDt | paced migrate policies |
| x-N-xp | N-N-Sp<br>Am-N-Sp<br>N-N-Rp<br>Am-N-Rp<br>N-N-Cp<br>Am-N-Cp | push migrate policies<br>(some with inactive pull) |
| Am-xr-x | Am-Ar-N<br>Am-Rr-ROt | replicate policies |
| Am-RrBm-x | Am-RrBm-ROtSt<br>Am-RrBm-ROtRt | replicate read-only and balance migrate |
| Wm-Wrm-x | Wm-Wrm-SDt<br>Wm-Wrm-RDt<br>Wm-RrWm-ROtSDt<br>Wm-RrWm-ROtRDt<br>Wm-RrWm-ROtLTDt | warm replicate or migrate policies |
| Pm-Prm-x | Pm-Prm-SDt<br>Pm-Prm-RDt<br>Pm-Prm-TDi<br>Pm-Prm-TDt<br>Pm-RrPm-ROtSDt<br>Pm-RrPm-ROtRDt | paced replicate or migrate policies |
| x-x-x-NSNR | Pm-Prm-N-NS<br>Pm-Prm-TDt-NR<br>Wm-Wrm-N-NS<br>Wm-Wrm-RDt-NR | no scanner and no replacement policies |

Table 3: Common Policy Groups with Examples

"shared access pattern report". Periodically, each node reports the total number of processes (local and remote) that referenced each of its remotely-referenced frames during the current interval. The generated reports also indicate whether or not the page was modified. These reports provide insight into the memory sharing patterns of the application. For example, it is easy to see pages that should be replicated (shared by several processes but not modified) and pages that should be migrated (referenced by only one remote process). The reports also give us a feel for the amount of active sharing (pages being read-write shared by several processes) in the application.

Another special DUnX policy generates "local-remote reference reports". Periodically, each node reports the number of local and remote pages referenced by that node in the previous interval, as well as the number of local and remote pages modified by that node. While not nearly as detailed as a complete memory reference trace, these reports are easy to generate and provide a level of information that has proven to be useful in aiding our understanding of the behavior of different application-policy combinations.

Some of our most powerful information collection techniques are based on performance comparisons. By slightly modifying the memory management policy or the application program and then comparing the results, we can gain valuable insight into the issues affecting performance. For example, we might compare two versions of the same application, where one version has been modified to manually place (and fix) a certain frequently used data structure. If the original version suffers a far greater number of coherency faults than the modified version (as well as longer completion times), we have evidence that the memory management policy in use suffers when presented with the original memory reference behavior.

This same approach can be applied by comparing similar memory management strategies. For example, rather than varying the application program, we might compare two policies that differ only in their criteria for replicating a page. If performance were identical in both cases, there is evidence that for the particular application being run, both replication criteria work equally well. If, on the other hand, one policy performed much better than the other, we have learned something about how this test application reacts to different page replication criteria. The power of this information collecting technique will become more apparent when we discuss the results of our study.

While we have discussed the assortment of information that we are able to collect, we have said little about the information we cannot easily collect. Perhaps the most important piece of information that we have no way of gathering is the average memory access time of a given application under a given memory management strategy. While this would be a very valuable piece of information to have, it is clear that it would be impossible (without special hardware support) to collect such data from an operating system kernel without disturbing the behavior of the application. Even knowing the numbers of remote and local memory references would be useful, but this too, is extremely difficult to measure without special hardware. Similarly, it is difficult to measure the amount of time spent by the kernel in low-level processing (e.g. in the lowest-level trap handler). Kernel event logging might be used to measure these small bits of overhead, but then the event logging itself might become a significant overhead. Additionally, the volume of data logged would grow significantly. In general, there is an inverse relationship between the time it takes for an operation to complete and the difficulty associated with collecting information about that operation.

### 4.2 Workload

In this section, we describe the individual programs that comprise our experimental workload. These programs have been contributed by members of several BBN Butterfly user communities in response to our requests for real application code. One advantage of this collection of programs is that they were all developed independently of our project and without any knowledge of DUnX features or policies being evaluated.

Since the target machine for most of the programs we have received belongs to the BBN Butterfly family, some degree of tuning for a NUMA architecture is generally evident. A NUMA-oriented test suite can answer questions about how various memory management policies interact with programmers' attempts to strengthen locality. However, programs that are not tuned to the memory architecture or assume an UMA environment are useful to answer whether the OS can be an effective substitute for some of the tuning now required. In spite of efforts to solicit UMA programs, only two have been offered. Thus, our approach has been to make very minimal modifications to the submitted workload to eliminate some of the NUMA bias, creating what we refer to as the UMA versions of those original programs. This has the added benefit of allowing us to compare the performance achieved by sophisticated policies for the UMA versions with the base-case performance of the hand-tuned NUMA versions.

Many of the programs are based on BBN's Uniform System library [5]. This programming system provides support for process generation and allocation of shared memory within a parallel program. There are memory allocation functions that give varying degrees of control over the assignment of data to nodes. These include `UsAllocOnUsProc`, which is supposed to allocate memory on the specified processor node of the current cluster, `UsAllocLocal`, which is supposed to allocate memory on the local node of the calling process, `UsAllocScatterMatrix`, which is designed to distribute a matrix over all nodes in the cluster, and the general `UsAlloc`. In the delivered implementation of the Uniform System under nX, each node has an associated memory object bound to it which is used by the Uniform System allocators as the source of dynamically allocated space on that node. One simple transformation applied to NUMA programs written for the Uniform System to produce a more UMA-like version is to replace the node-specific memory allocation calls with calls to the generic `UsAlloc`.

### The `hh3d` Program

The `hh3d` program performs simulations of discontinuous conduction in nerve or cardiac muscle. The formal discontinuous model of conduction used and the simulator that is an instance of that model are discussed in [30].

The input to `hh3d` consists partly of a collection of *computational cells* and their connections to one another. The simulation is parallelized by dividing the collection of computational cells into $n$ sets, each the responsibility of one of the $n$ processors.

At each simulation timestep, the data values associated with a computational cell are updated based on their current values and values of connected cells from the previous iteration. There is considerable locality inherent in the problem, since computational cells tend to be connected to cells that are close by.

This application was written by an experienced NUMA programmer using the Uniform System. As a result, data was carefully placed by hand in order to minimize the average memory access times. Using purely syntactic changes, we created an UMA version of the program by replacing node-specific memory allocation calls as described above.

The shared access pattern reports generated with a special DUnX memory management policy show that both the NUMA and UMA versions of `hh3d` use relatively few shared pages, but that each of those pages is actively read-write shared by several nodes. The shared pages are nicely distributed over the nodes (about two such pages per node) for the NUMA version, but all are placed on a single node under the UMA version. The local-remote reference reports show the corresponding differences in the numbers of local and remote pages referenced by each node.

### The `psolu` Program

The `psolu` application solves the sparse linear system $Ax = b$ using block chaotic relaxation [40].

The input matrix $A$ is broken into blocks, which form the unit of parallelism. A static load balancing scheme uses the block sizes to estimate the processing time required for each block. Blocks are assigned to the processors so that the estimated processing time is minimized. A special convergence-checking process continuously computes $||Ax - b||$ and compares it to the zero vector. When the solution is sufficiently accurate, computation ceases.

Blocks of $A$ are placed (manually) in memory on the node responsible for processing that block, as are the corresponding blocks of $x$. Each processor has its own local copy of $b$. The $x$ vector is the only actively-shared data structure, since neither the input matrix $A$ or the vector $b$ are changed during the computation.

An interesting point to make is that the speed at which the convergence-checking process runs is relatively unimportant. While placing $A$, $x$, and $b$ in its local memory may greatly increase the speed of the convergence checker, this only hampers the performance of the processes actually responsible for solving the system. Of course, without assistance from the programmer it is impossible for the operating system to know that it would be unwise to migrate pages of $A$ or $x$ to that convergence-checking node.

We created an UMA version of `psolu` by replacing Uniform System allocation routines. Although the UMA version is still partially NUMA (e.g. the $b$ vector is still replicated), it does provide us with two versions of the program which we can compare.

Shared access pattern reports for both versions of `psolu` show a fair amount of page sharing (approximately 80 pages total, most shared by three or four processes). With the NUMA version of the program, those pages are perfectly distributed among all but one of the nodes (none are placed on the node running the convergence checker). Under the UMA version of the program, all of the shared pages are placed on a single node (the one running the convergence checker).

### The `fish` Program

The `fish` program [27] implements the "sharks and fishes" problem originally described in [20] using the Uniform System. The program simulates sharks and fishes moving about in a two-dimensional sea. Each point in the sea is occupied by at most one creature. Fishes move about arbitrarily, occasionally giving birth to more fishes. Sharks also move about arbitrarily, but they always eat an adjacent fish if possible. Sharks also give birth to more sharks, and they starve to death if they do not eat enough fish.

The program uses a concurrent pool of elements (sharks and fishes) as the source of parallelism. Worker processes acquire elements from the pool, adding updated elements to a second pool, until the source pool is empty. An empty source pool signals the end of the timestep, at which point the roles of the two pools are reversed and processing of the next timestep begins.

The concurrent pool data structure is designed to maximize node-memory locality. Unfortunately, when the local portion of a pool empties (as it always does towards the end of a timestep), workers attempt to acquire work from other processors. This results in elements being moved from one processor to another. Since each element is small, many fit in a single page of memory. Before long, many virtual pages contain elements in use by several different processors. As a result, a great amount of active memory sharing takes place. Much of this sharing, however, is *false sharing* (entire pages are shared, but the individual data elements that comprise them are not).

Again, we have created an UMA version of `fish` by substituting allocation calls. Both versions of `fish` exhibit a high level of active sharing. The shared access pattern reports show that most of the very large number of shared pages are in active read/write use by a majority of the processors taking part in the computation. Because of this fact, it is difficult to select the best location for each of the virtual pages. Local-remote reference report data show that the majority of pages referenced by each node are remote.

### The gauss Program

The gauss program simulates gaussian elimination on dense matrices without pivoting. The program is a simulation in the sense that integer rather than floating point arithmetic is used in the computation. The program is derived from the same source used by Fowler and Cox in [19]. It has been ported to DUnX by one of the authors of this paper. This port does not represent a significant modification since the system call interfaces of Platinum and DUnX resemble each other (both being descendents, in some sense, of Mach). This program adopts an UMA model, although attention has been paid to page boundaries in the allocation of data.

We use the same 800x800 input matrix used by Fowler and Cox in their study. The matrix is stored in 801 virtual pages; a vector of row pointers is contained in one page, and each row of the matrix is stored in its own virtual page. Other shared data used by gauss include a vector of iteration counts (used for synchronization), a counter used as a barrier to ensure that all the processes begin computation at the same time, and a counter used to keep track of which children have completed processing.

The computation is parallelized statically by assigning row $i$ to processor $i \bmod n$, where $n$ is the number of processors used. The vector of iteration counts is used to ensure that a row has been processed completely before it is used as a pivot row. An important consequence of this implementation approach is that each row (which resides in its own page) is modified only by the main process during initialization and by the processor to which the row is assigned. Furthermore, no other processor references the row until after processing of the row is complete.

It is fairly easy to see that it is best to place each row page on the node which is responsible for processing that row. Also, it is wise to make a local copy of the current pivot row during each iteration, to decrease contention to the pivot elements. There seems to be no "best" place for the pages containing synchronization data, but since the number of references to these pages is relatively small, this isn't likely to pose serious performance problems.

In order to fully understand the behavior of gauss under different page placement strategies, we experimented with several modified versions of the program as well. One variation manually replicates the pivot page during each iteration, one permanently fixes the placement of pages containing synchronization data, and one version is an excellent NUMA implementation of the algorithm (placement is done manually).

The shared access pattern report generated for the UMA gauss clearly shows the potential for page migration to improve performance. Many (over 250) pages referenced only by a single remote process are placed on one node (the node running the main process). In each report interval, there are several pages referenced by all of the processes. These pages likely contain pivot rows. One page is referenced by every process during each interval, but except during initialization, is never modified. We believe that the page contains the vector of row pointers for the input matrix, and is an obvious candidate for page replication. The NUMA gauss shared access report is somewhat different. In each report interval, there are several pages referenced by all the processes in frames on each of the nodes. We suspect that these pages contain pivot rows of the input matrix, which are manually replicated to local memory before being used by remote processes.

### The `msort` Program

The `msort` application is an UMA parallel merge sort, ported to DUnX from the same sources used by Fowler and Cox in their study [19].

The primary shared data structures are the input array, an array half the size of the input array used as a scratch area during merge phases, and an array of synchronization variables used to control the merge phases. The input array (initialized by the main process) is divided into $n$ equally-sized pieces each sorted by one of the $n$ processors using heapsort.

Once the $n$ array pieces have been sorted, the merge phase begins. Merging proceeds up the merge tree, with the left-hand sibling merging the results from the previous level of the tree. Pair-wise synchronization ensures that the merging proceeds correctly. Processing completes when node zero merges the results of the left and right subtrees.

Let us consider the shared memory reference patterns exhibited by `msort`. If the input array is sufficiently large (i.e. $\geq s \times n$ bytes, where $s$ is the pagesize and $n$ is the number of processors) and a multiple of the pagesize, then each processor is initially responsible for sorting a portion of the array stored in a distinct set of virtual pages. If these conditions are not met, some false sharing occurs at the lowest levels of the merge tree.

Assuming that the size of the input array is "nice", the pages containing array elements from the right-hand siblings become the responsibility of the left-hand siblings as the computation progresses up the merge tree. Thus, the primary user of an array page may change up to $\log n$ times. Note, however, that at no one time will more than one processor require access to any single array page.

The shared array used as scratch space during the merge steps is used in a manner similar to that of the array page. As a result, each of its pages may be used by several different processors during the computation, but never by more than one at a time. The page containing synchronization data is actively shared by all of the processors. Ideally, this page should be placed somewhere (almost any frame will do) and left there for the duration of the program.

Most of our experiments use an input size of 819200 elements with sixteen processors. Since the DUnX pagesize is 8192 bytes, the input array fits perfectly in 400 pages. The sixteen processors are each initially responsible for 25 distinct array pages.

The shared access pattern reports for `msort` show that there is great potential for page migration to improve computation performance. Most of the array pages are placed in frames on the node running the main process when the array is initialized, and as a result, the shared access pattern reports show that a large number of pages placed on that node are referenced only by a single remote process (namely, the process responsible for the array portions in those pages) for most of the program's duration. The local-remote reference reports offer further evidence that most pages are incorrectly placed, since an overwhelming majority of the pages referenced by all but one node are remote.

### The `mandelbrot` Program

The `mandelbrot` program generates image data based on mandelbrot set calculations. The application was written by one of the authors of this paper as a simple exercise in parallel programming in the spring of 1988, prior to the design of DUnX.

The program is parallelized by assigning $r/n$ consecutive rows (where $r$ is the total number of rows and n is the number of processors) of the pixelmap to each processor. The calculations required to obtain the value of a pixel do not involve the values of any other pixels, so no sharing of the pixelmap is required. The original version of `mandelbrot` uses the BBN Mach-1000 (also found in DUnX) `vm_allocate_and_bind` system call to ensure the optimal placement of pixelmap data.

Other shared data used by `mandelbrot` consists of a vector of row pointers for the pixelmap and a structure of global variables. The vector of row pointers is never modified once initialized, so is replicated on each processor. The structure of globals contains synchronization variables (a barrier at the start of the computation and a counter of completed processes) and run-time constants (e.g. pixelmap size). The global structure is allocated on node zero of the cluster and left there for the duration of the computation.

We have also considered an UMA version of `mandelbrot`, created by replacing `vm_allocate_and_bind` calls with `vm_allocate` calls. In addition, the vector of row pointers is not manually replicated. There is little difference in the performance of the two versions, however, since the single node that references each pixelmap page causes the page to be paged into a frame on the faulting (local) node.

The shared access pattern reports for `mandelbrot` indicate that `mandelbrot` makes very little use of shared memory. For the UMA version, there are two pages regularly referenced by all of the processes. We suspect that one contains the vector or row pointers to the image matrix, and the other contains the structure of globals. Data collected for the NUMA version of `mandelbrot` shows that only a single page is regularly shared. This is as we expected, since the NUMA version manually replicates the vector of row pointers to each node.

### The `hough` Program

The `hough` program computes the hough transform of an image read from a file [15]. The program computes the same transform several times, simulating the way such a program would be used as a component in a real system.

The primary shared data structures are the image matrix (actually, two copies are maintained), the hough space matrix, several vectors used to store summary information, and several matrices containing trig values precalculated during initialization. The working copy of the image matrix is distributed over the processor nodes by allocating it with the Uniform System `UsAllocScatterMatrix` function. The hough space matrix is replicated to each processor manually, despite the fact that the matrix is logically shared. The manual replication works since each processor references and modifies a distinct portion of the matrix. The matrices containing precalculated trig values are manually replicated to each processor node as well.

We created an UMA version of `hough` by removing all code to implement manual replication and replacing all data allocation calls with `UsAlloc`. The resulting UMA version is much simpler than its NUMA predecessor.

For the UMA version of `hough`, the shared access pattern reports show alot of potential for both migration and replication to improve performance. Most of the shared pages are placed on a single node, and most are either read-write modified by a single process, or read-only shared by nearly all of the processes. The NUMA results, on the other hand, show a fair amount of active read-write sharing of pages placed on all of the nodes (about five such pages per node), as well as several incorrectly placed (referenced only by a single remote process) pages placed on the node running the main process. We suspect that the scatter matrix construct used by the NUMA version of `hough` actually increases the amount of active page sharing, although it does distribute those pages fairly well.

## 4.3 Basic Operation Timings

The performance of any memory management strategy depends partly on the relative costs of the basic operations the system must perform. Important factors include memory access times, page

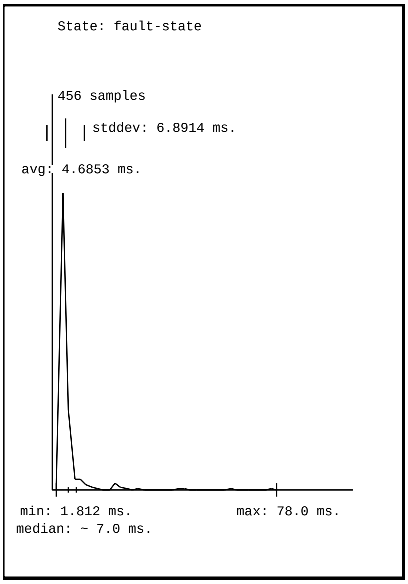


Figure 1: N-N-N-NBNS Page Fault Time

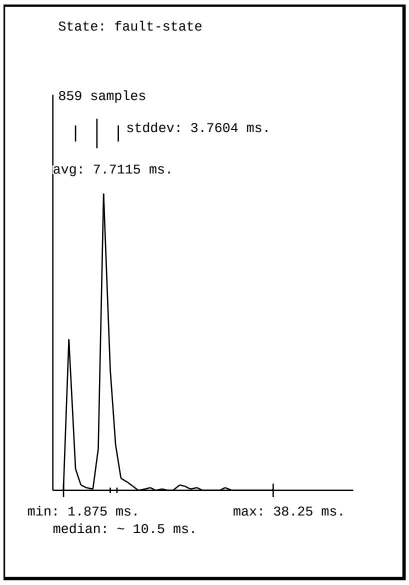


Figure 2: Am-Am-N Page Fault Time

fault time, page migration time, page replication time, coherency fault time, and page scanner overhead time.

Ignoring memory and switch element contention factors, the time to read a local memory location on the Butterfly GP1000 is approximately 0.5 to 0.6 microseconds ($\mu$s.), and the time to read a remote memory location is approximately 7.5 $\mu$s. Thus, a remote memory access costs about 12.5 to 15 times a local memory access.

Cost measures for the other basic operations have been collected using BBN's kernel event logging mechanism and the `gist` application. Since the costs of basic operations may be affected by a wide assortment of contention factors difficult to control when running real user applications, we have implemented a special application just for the purpose of collecting this data. The `test` program consists of $n$ processes, each of which consists of two phases. A large array (200 pages) is shared by each of the processes, and an equal portion of that array (typically an integral number of pages long) is the responsibility of each of the processes. In the first program phase, each process modifies its array elements, and then reads the remaining elements. Each array page ends up resident in a frame on the node to which it belongs (i.e. the node responsible for that page), since that node is the first to reference it. After the first phase, the processes synchronize and then enter the second phase, which is exactly the same as the first. Since the behavior of the `test` program is well understood, it is simpler to understand the kernel timings obtained when running it.

A histogram of page fault times incurred by `test` running on two processors under the N-N-N-NBNS policy is plotted in figure 1. We see that the average time to process a fault in this case is quite low: less than 4.7 milliseconds (ms.). The reason for this low value has alot to do with the *type* of faults encountered. Of the 456 measured faults, we know that 200 are zero-fill faults (the shared array is 200 pages long) and 200 are simple translation faults (faults on resident pages). Only 18 hard faults (pageins from disk or disk buffer cache) are encountered, with the remaining faults being zero-fill, copy-on-write, and copy-on-read data faults.[4] The 18 hard faults explain the

[4]The kernel event logging mechanism records all of the events that occur during the specified time interval, regardless of the process that triggers the event. Because of this, it is possible that a few of the remaining unaccounted for faults are unrelated to the `test` program experiment. The effects of such faults on this analysis is neglible.

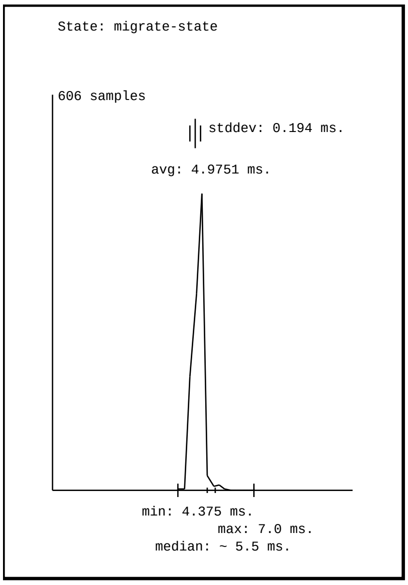


Figure 3: Am-Am-N Page Migration Time

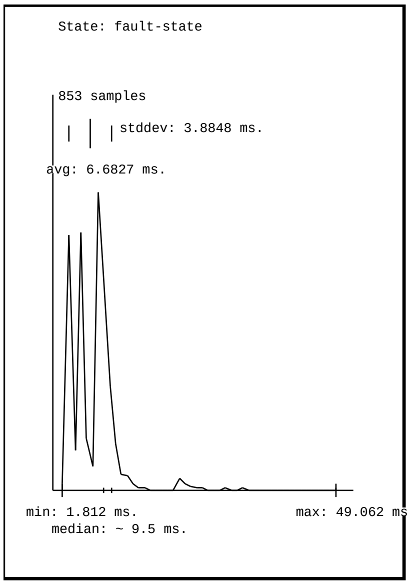


Figure 4: Am-Ar-N Page Fault Time

few slow points in figure 1, and the majority of quick zerofill and translation faults explain the relatively low average time.

To determine the time required to migrate a page, we consider the `test` program running on two nodes under the Am-Am-N policy. Since each array page will be migrated three times (once in phase one and twice in phase two) we know there will be at least 600 page migrations. As it turns out, the page containing the barrier synchronization variable is also migrated six times, so a total of 606 page migrations occur. As with the N-N-N-NBNS case, there are 18 hard faults and 200 zerofill faults. A total of 859 page faults are encountered, as can be seen in figure 2. In the figure, we see that the average fault time is somewhat higher than in the N-N-N-NBNS case, at almost 7.8 ms. The most obvious difference, however, is the shape of the histogram. There are two distinct peaks, each the result of a different type of fault. The first peak is comprised of zerofill and translation faults, whereas the second peak is comprised mostly of faults that result in page migration operations.

A histogram of the measured migration times is given in figure 3. The migration time includes the time required to invalidate the source (remote) copy of the page and to flush the remote TLB entries. Because of the simplicity of the `test` application, we know that a single remote mapping is invalidated as a result of each page migration. The average measured time is a bit less than 5 ms. per migration, which is about 667 times the cost of a remote access.

We used the `test` program running on two nodes under the Am-Ar-N policy to determine page replication and coherency fault costs. A histogram of page fault times is given in figure 4. There are three spikes in the histogram; the first consists of zerofill and translation faults, the second consists of coherency faults, and the third consists of page replication faults. As usual, there are a small number of more costly faults that involve paging in data from disk or the disk buffer cache. The average fault time is about 6.7 ms.

In figure 5, a histogram of replication times is shown. Each array page is replicated twice (once in each phase), for a total of 400 page replications. As the histogram shows, the time to replicate a page is quite consistent, taking an average of 4.7 ms. per replication. Since all of these replications involve changing the original copy of the page to a replica (thus requiring a TLB shoot to lower

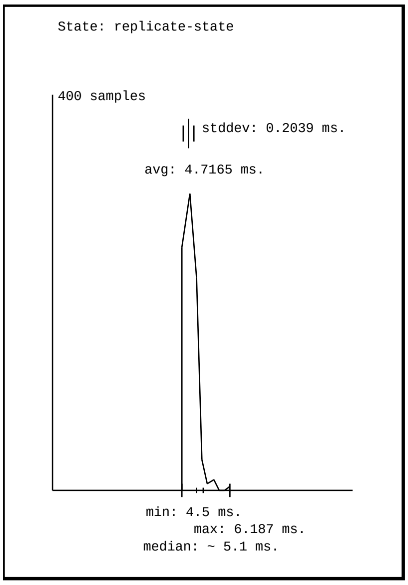


Figure 5: Am-Ar-N Page Replication Time

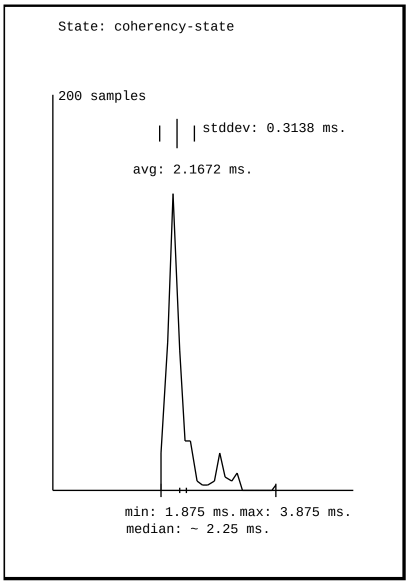


Figure 6: Am-Ar-N Coherency Fault Time

the hardware protections to the page), the measured replications are among the most expensive. Creating additional copies of an already replicated page is quicker, since existing copies of the pages are already protected from write access.

A histogram of coherency fault processing times (the measured time includes only the processing required for the coherence operations, not the entire fault) is shown in figure 6. Exactly 200 coherency faults occurred (one for each array page at the start of phase two), taking an average of less than 2.2 ms each (substantially less than the average fault time of 4.7 ms. measured in the N-N-N-NBNS case). Each of these faults required the invalidation (and TLB shoot) of exactly one remote copy of the page.

To ascertain page scanner overhead, we used a modified version of the `test` program with four nodes running under the Wm-Wrm-RDt policy. The version of `test` that we used for this experiment simply repeats the two program phases twenty-five times each. This change was made so that the page scanner would be able to run several times during the program's execution. The Wm-Wrm-RDt policy was chosen somewhat arbitrarily, but the amount of work it requires of its page scanners is fairly representative of most of our policy choices.

A histogram of page scanner run times is shown in figure 7. The distribution is bimodal for two reasons. First, there are two types of page scanner runs: quicker runs that do not execute the "move pages" phase, and slower runs that do execute the "move pages" phase (which has an average cost of about 55 ms.). The second contributing factor is that the cost of the "count references" phase is also bimodal. The cost of the "count references" phase varies because the number of remote references required by a page scanner varies with the number of remote pages referenced by that processor since the scanner last ran. This in turn varies as pages are migrated and replicated among the processor nodes.

Most of the page scanner time is spent in the "count references" phase, which takes an average of over 53 ms. The "move pages" phase takes a bit longer (55 ms.), but is executed only once for every eight "count references" executions. The "age usage counts" phase takes a negligible (1.7 ms.) amount of time relative to the total time spent in the scanners. All together, the total average scanner time is about 62 ms., as indicated in figure 7.

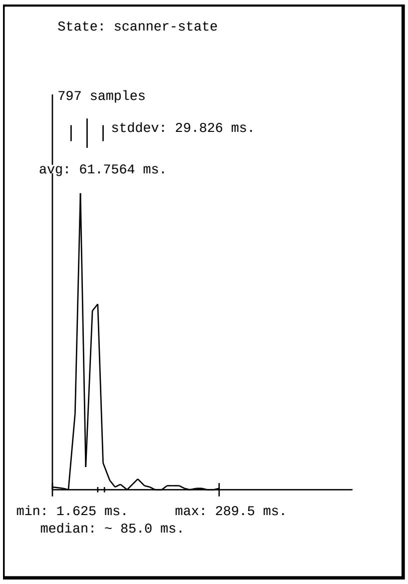


Figure 7: Page Scanner Time Distribution

Since the page scanners run once each second, and each run takes an average of 62 ms., the total overhead is easily calculated to be about 6%. Measurements obtained with `gist` show that the per-node overhead ranged from 4.4% to 6.8%, for an average of 5.6%. The overhead could be reduced by decreasing the frequency of page scanner runs. The effects of the page scanner on application performance are discussed in section 5.9.

It is important to point out that the time required for a page scanner to run is highly dependent on the collection of processes running on the node for which it is responsible. This is because the time required for a page scanner run is related to the size of the page tables on the local node (the "count references" phase requires the scanner to consider each of its page table entries), as well as to the number of remote pages referenced by local processes (each such page requires the "count references" phase to update a remote page usage counter).

Since DUnX has been developed as an experimental research vehicle, an important design goal is flexibity. Because of this goal, we have typically made generality versus performance trade-offs in favor of generality. The performance of some of the operations discussed in this section could likely be improved by lower-level implementations of specific policies.

# 5 Discussion of Experimental Results

The results of our experiments are presented and discussed in this section. Rather than presenting the results on a strictly per-policy or per-application basis, we attempt to answer a series of questions concerning the different page placement strategies. Each subsection addresses a different question and experimental results that help to answer that question are presented.

When analyzing the results of an experiment, there are many contributing factors that must be considered. The following list enumerates some of the more important behavioral aspects that must not be ignored, even though it may not always be easy to quantify their effects:

1. Costs associated with page faults – Every page fault has some non-zero cost associated with it that must be compensated for by savings that result from experiencing the fault. The number of page faults is affected by the page replacement and triggering strategies being used.

Savings take the form of lowered average memory access times resulting from migrations or replications.

2. Cost associated with page scanners – The page scanners consume CPU, memory and switch cycles. They are only effective when the savings that result from running them exceed the associated costs.

3. Appropriateness of page migrations – In general, it is very difficult to assess the quality of an individual migration decision due to the complexity of the interactions that can arise. Migrating a page to the wrong node can be costly. Memory reference times to the page may increase, and more migrations and/or replications may be caused in an attempt to correct the situation. Even when a page is migrated to the node referencing it most often, the average memory access times of other nodes may increase due to differing contention effects (e.g. it is possible to create memory "hot spots" by migrating pages to the wrong places). Intuitively, a *desired* migration is one that serves to decrease the overall computation completion time.

4. Appropriateness of page replications – Pages shouldn't be replicated if coherency faults will result too frequently or too soon. The cost/benefit of a page replication includes the cost of coherency faults, so replications only pay off when memory access savings exceed all the costs associated with resulting coherency faults.

5. Quickness of page migrations and replications – In general, the sooner a desired migration or replication is performed, the better. A migration or replication to a desirable destination that comes too late may be completely useless, and even costly.

6. Relative importance of different page migrations and replications – Not all migrations and replications are created equal. The replication of one page may greatly reduce memory and switch contention for all the processes, while the replication of another may benefit only one process. Thus, the fact that one policy performs more migrations or replications than another does not necessarily imply that the resulting performance will be any better or any worse than under the other policy.

7. Ordering of events – The order in which events occur can be quite important to the overall performance of a policy. For example, if all the right migrations and replications are performed, but they occur in "bursts", performance may suffer due to switch and memory contention, as well as to contention to various kernel data structures in the fault handlers. The ordering of events may also affect the relative speeds of different processes, affecting the reference patterns (and hence the fault, migration, and replication behavior) for the remainder of the program's execution.

In this section, we attempt to gain an understanding of the relative importance of different policy features for each of our test applications. Though the goal is to come to a complete understanding of the observed behavior, that is in many cases an extremely difficult goal to attain due to the complexity of the issues involved. Most of our analysis is based on completion time data and fault, migration, and replication counts, since that information is readily available. As should be evident from the preceding discussion, these data items do not paint a complete picture. They do, however, provide evidence (admittedly circumstantial in some cases) that can help us to understand certain aspects of the observed behavior.

## 5.1 Effectiveness of Dynamic Page Placement

Perhaps the most important question to be answered is quite simple: can dynamic page placement strategies effectively deal with the NUMA problem?

In figure 8, a summary of all our experiments is presented. There are 322 horizontal lines in the figure, separated into seven groups of 46. Each group summarizes the results of experiments done with a particular application (as indicated along the $y$-axis), with each line within the group summarizing the results of a different experiment. The length of a horizontal line represents the computation completion time for that experiment measured relative to the computation performance under our static base-case (N-N-N-NBNS) policy. The first line of each group gives the performance of the NUMA version of that application under the static base case policy (except for `msort`, for which we have no NUMA version). A vertical line drops from the end of the NUMA line, to make comparisons with other results simpler. The second line of a group shows the performance of the base-case policy, which is always 100%, since the results are measured relative to this policy. The remaining lines in a group summarize the results obtained using different page placement strategies. The ordering of the lines within a group is the same as given in table 3. The two policies that perform poorly in nearly every case are Am-Am-N and Am-Ar-N which always migrate or replicate. Obviously, determining how a particular policy performs using only this plot is incredibly tedious. However, that is not the intended purpose of the figure.

Using figure 8, one can immediately answer several questions:

1. Do the NUMA versions of our test applications perform better than the simpler UMA versions under the same static base-case policy?
2. Can a dynamic page placement strategy improve the performance of the UMA versions of our test applications relative to the base-case performance?
3. Can any of the dynamic page placement strategies approach the level of performance obtained with the NUMA versions under the base-case strategy for the UMA versions?

The answer to our first question is rather clear. The NUMA versions perform substantially better than the UMA versions in all but the `mandelbrot` case. Except for the `mandelbrot` data point, this is exactly the behavior we expected.

The `mandelbrot` result is fairly interesting, however. The programmer is known to have spent extra time and effort to ensure that each portion of the image matrix would reside on the best possible node. To create the UMA version, that code has been removed, essentially leaving the (static) initial placement of the image matrix to the operating system. But, since the static placement policies place pages on the node that first faults on them, the image matrix pages end up on exactly the same nodes as they do in the NUMA version. Thus, the extra effort on the programmer's part has been completely fruitless and at the same time makes the program less portable.

The answer to our second question is somewhat more complicated. For five of our seven applications (`hh3d`, `psolu`, `hough`, `msort`, and `gauss`), the answer is affirmative (i.e. there exist horizontal lines that end to the left of the dotted vertical line at $x = 100$, indicating performance improvements). In some cases (e.g. `psolu`, `msort` and `gauss`), the performance improvement is quite substantial. However, all of the dynamic page placement strategies that we investigated are unable to improve the performance of the two remaining applications. For `mandelbrot`, this is somewhat expected since the UMA version performs as well as the NUMA version even using the static base-case policy. Since `mandelbrot` does very little sharing of memory, there seems to be very little any dynamic strategy can do to improve performance.

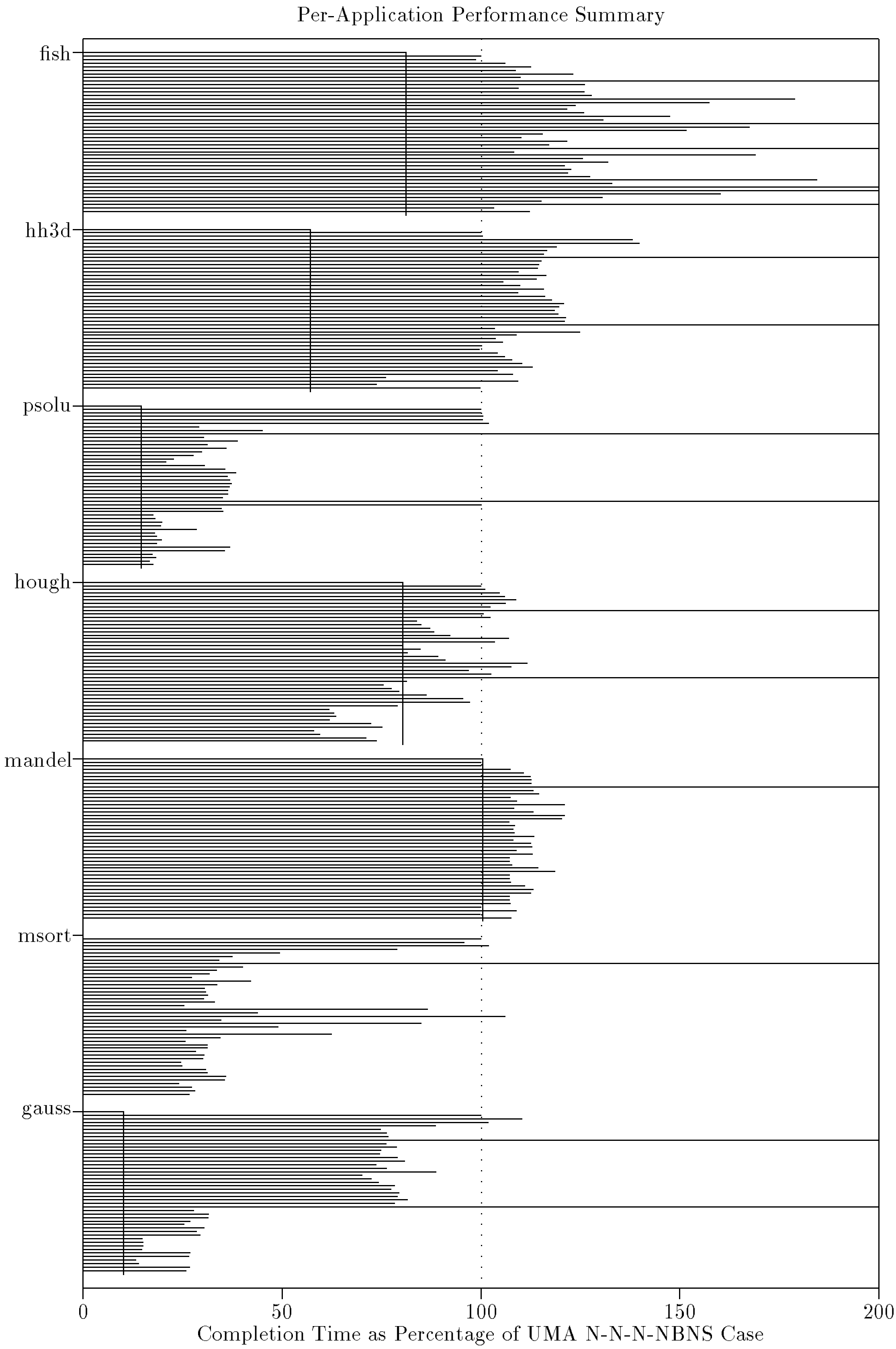


Figure 8: Performance Summary

The results for the fish application tell a very different story. None of the dynamic strategies are able to improve performance, yet the NUMA version of fish does perform about 25% better than the UMA version under that same base-case policy. The large amount of active memory sharing (most of which is false sharing) exhibited by fish makes it very difficult for our operating system level strategies to determine the best location for each of the shared pages.

Our third question is perhaps the most interesting, since it essentially asks whether or not any of the dynamic page placement strategies that we considered can achieve the level of performance obtained by experienced NUMA programmers.

For fish, mandelbrot, and hh3d, the results are not encouraging. As we already indicated, none of the dynamic strategies have been able to improve the performance of the UMA version of the fish and mandelbrot applications. Only two dynamic page placement strategies have been able to improve the performance of the UMA hh3d program, though neither achieves quite the performance of the NUMA version. The two policies that are able to improve performance do not use the page scanners, suggesting that overhead induced by the scanners has hampered the performance of the other dynamic strategies. This is discussed in more detail in a later subsection.

Excellent results were obtained with the psolu and gauss applications, where several dynamic page placement policies were able to perform nearly as well with the UMA versions as the base-case policy did with the NUMA versions. The extra effort required by the programmer to create the NUMA versions of these applications (which was quite extensive for psolu, according to its author) would have to be carefully weighed against the performance improvement gained through that extra effort.

Several of our test policies actually gave performance with the UMA version of hough that is better than the NUMA performance with the base-case policy. However, we cannot be sure that the NUMA version we considered is the best version that could be written. Specifically, the NUMA version makes extensive use of Uniform System scatter matrices, which are designed to reduce memory and switch contention but do little to improve memory reference locality. Perhaps more careful placement of portions of those matrices could be done, producing a NUMA version with still better performance. Of course, the extra programmer effort required cannot be ignored.

The other thing evident from figure 8 is that there is no single policy that can be considered the best over the entire application test suite. However, policies in the x-x-x-NS/NR category (the bottom four lines of the groups) often appear among the possible candidates. Certain of the replicating policies, in particular the Pm-Prm-x family, also show promise.

To summarize, we have found that there are dynamic page placement strategies that are able to improve the performance of some of our test applications. In some cases, that performance improvement approaches (and in one case exceeds) that obtained by the experienced NUMA programmer. It is important to remember however, that we have not exhausted either the policy space or the application space. Hopefully, the examples we have considered are representative of the entire space.

### 5.2 The Effects of Page Replication

In this subsection we address the question of whether or not allowing multiple physical copies of a single virtual page is worthwhile. This is important because if single-copy dynamic page placement strategies perform as well as multiple-copy strategies, the problem of maintaining data coherency can be avoided by using only single-copy policies.

Noting that the multiple-copy page placement policies are the last in the table 3 policy list and that figure 8 uses the same policy order, we can use figure 8 to obtain an initial answer to our question. The last nineteen data points (horizontal lines) in each group are from the multiple-copy

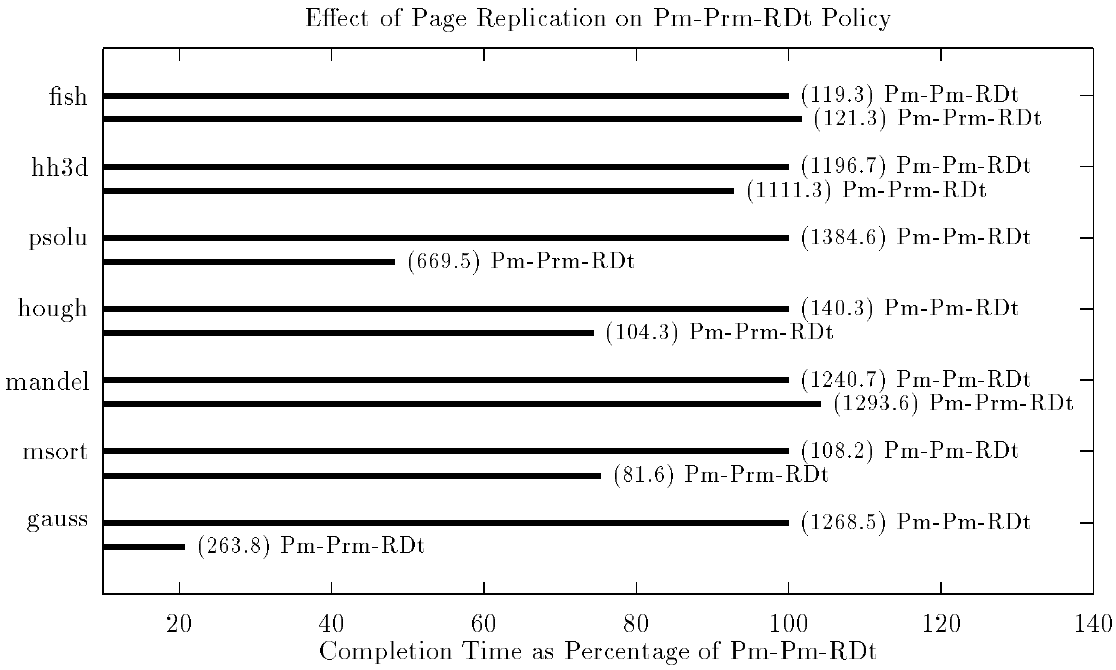


Figure 9: Pm-Pm-RDt and Pm-Prm-RDt Completion Time Comparison

test policies.

Using figure 8, it is fairly easy to see that `hough`, `psolu`, and `gauss` appear to benefit a great deal from page replication. Benefits exist, but to a lesser extent, for the `hh3d` and `msort` programs. On the other hand, the `fish` and `mandelbrot` applications appear to be relatively unaffected by page replication.

To gain more insight into the answer to our question, let us narrow our discussion to two specific policies: Pm-Pm-RDt and Pm-Prm-RDt. These policies are identical except that the former does not allow multiple physical copies of a single virtual page (and thus does not need to worry about data coherency), while the latter does. The performance of these policies is summarized in figure 9, where completion times are presented. The length of the horizontal line represents the time relative to the Pm-Pm-RDt policy. Each line is additionally labeled with the policy name and the actual completion time value in parentheses. All figures in this section use seconds as the unit of time, unless otherwise indicated. The results show that for most of our test applications (all but `fish` and `mandelbrot`), replication appears to be quite worthwhile. For the `gauss` and `psolu` applications, execution time is cut dramatically by allowing page replication.

Of course, completion time results alone do not tell the entire story. Migration and replication each address a different problem arising in the memory access patterns. The results of figure 9 suggest that replication is an appropriate operation for most of our applications. However, migration and replication are, in some cases, interchangeable. The effect of a page migration can also be accomplished by replicating a page and then taking a coherency fault on it (by writing to it). We call this an *RC migration*. The Pm-Prm-RDt policy chooses to replicate a page rather than migrate it whenever the memory reference that triggered the fault is a read, and since it seems common for a program to read a page before writing to it, many RC migrations may occur. Unfortunately, an RC migration is more expensive than an ordinary page migration, since it involves processing two page faults instead of just one.

In table 4, the numbers of page migrations for both policies (labeled "M") and, for the Pm-

| Application | Pm-Pm-RDt Migrations | Pm-Prm-RDt | | | | |
|---|---|---|---|---|---|---|
| | | M | R | C | R-C | M+C |
| fish | 3855 | 602 | 3902 | 3066 | 836 | 3668 |
| hh3d | 86 | 19 | 84 | 49 | 35 | 68 |
| psolu | 784 | 55 | 1740 | 858 | 882 | 913 |
| hough | 585 | 46 | 2335 | 168 | 2167 | 214 |
| mandel | 15 | 0 | 20 | 2 | 18 | 2 |
| msort | 1435 | 718 | 968 | 692 | 276 | 1410 |
| gauss | 2286 | 79 | 6521 | 646 | 5875 | 725 |

Table 4: Pm-Pm-RDt and Pm-Prm-RDt Operation Counts

Prm-RDt policy, the numbers of replications ("R") and coherency faults ("C") are given. The "R-C" and "M+C" columns for the Pm-Prm-RDt policy are the number of replications minus the number of coherency faults and the sum of the migrations and coherency faults, respectively. One immediate observation that can be made when looking at table 4 is that the number of migrations for the Pm-Pm-RDt policy is far greater than for the Pm-Prm-RDt policy for every application. This straightforward comparison can be misinterpreted, however, since it ignores any RC migrations that may have occurred.

Since the number of RC migrations is limited by the number of coherency faults, the number of actual migration operations performed under the Pm-Prm-RDt policy is less than or equal to the sum of the regular migration operations and the number of coherency faults (the "M+C" column). In all but the `psolu` case, the number of Pm-Pm-RDt migrations is greater than this maximal number of migrations for Pm-Prm-RDt ( the "M+C" entry). This suggests that for those applications, the Pm-Prm-RDt policy is able to successfully use replication to eliminate some degree of page bouncing that occurs under the Pm-Pm-RDt policy.

We can get an idea of how much true replication (loosely defined as a replication that isn't an RC migration component) occurs by considering the "R-C" data. The number of true replications lies between the "R-C" entry and the total number of replications (the "R" column). If the number of RC migrations is close to the number of coherency faults (a comparison of the "M+C" data and the number of Pm-Pm-RDt migrations lends some support to this hypothesis), then the number of true replications is likely near the "R-C" value.

If we adopt the "R-C" value as our estimate of the number of true replications, the only applications whose total replications consist primarily of true replications are `hough`, `gauss`, and `mandelbrot`. The `hough` application has several read-only data tables that can easily be replicated, as can the `gauss` pivot rows. Hand analysis of the `mandelbrot` source code shows that all twenty replications likely involved the same page (the one containing the global shared structure). One of the coherency faults is probably the result of the parent process clearing the startup barrier, and the other is most likely the result of the first completing child updating the "finished" counter.

The "R-C" estimate of the number of true replications for the `fish`, `hh3d`, and `msort` applications indicates that the majority of replications performed were RC migration components. Despite this, `msort` and `hh3d` both enjoyed some performance improvement due to the true replications that were performed. The `fish` program wasn't significantly affected by the ability to do page replication.

According to our "R-C" estimate, about half the total `psolu` replications are true replications. The fact that the "M+C" value for the Pm-Prm-RDt policy is greater than the number of Pm-Pm-RDt migrations suggests that the number of RC migrations may be somewhat less than the number of coherency faults, however. This in turn suggests that our estimate of the number of true replications may be too conservative. Based on our hand analysis of the `psolu` source code, we suspect that each process replicates certain portions of the $x$ vector during each iteration of the computation. At the end of the iteration, the $x$ vector is updated by the process responsible for that portion of the solution vector, and remote copies of the the vector are invalidated, only to be replicated again during the next iteration. This analysis supports our hypothesis that the number of RC migrations is somewhat less than the number of coherency faults, since we can easily account for many such faults that are not RC migration components. Clearly, this replication is beneficial, as the `psolu` completion time under the Pm-Prm-RDt policy is less than half of what it is under the the Pm-Pm-RDt strategy. In section 5.6, we present evidence that suggests that some of the Pm-Prm-RDt replications improve performance more by eliminating undesirable Pm-Pm-RDt migrations than by decreasing memory access times.

Nearly all of our test applications have shown improved performance under memory management policies that support page replication. It is interesting to note that none of our test policies are able to improve performance of the two exceptions (`fish` and `mandelbrot`), regardless of whether or not they support page replication. The fact that five of our applications showed improved performance with replication, despite the negative effects of the large number of RC migrations, suggests that even greater performance may be attainable with a policy that can better choose between the two options.

### 5.3 Push vs. Pull Based Page Migration

In this subsection, we compare push-based page migration policies to pull-based page migration policies. The goal is to determine whether one approach is superior to the other.

To narrow our discussion to specific policy examples, we chose to compare the Am-Tm-Rt, N-N-Rp, and Am-N-Rp policies. The Am-Tm-Rt policy provides pull-based migration, but relies heavily on a push-based trigger operation controlled by our remote-wins test. The N-N-Rp policy uses the same remote-wins test to control a push-based migration operation. Since the same test is used to control page migration in both of these policies, a comparison of the two provides insight into the differences between push and pull based migration approaches. We also consider the Am-N-Rp policy as a push based policy that also allows inactive pages to be pull migrated in the same way the Am-Tm-Rt policy does.

Completion time results for these three policies (relative to the Am-Tm-Rt policy) are summarized in figure 10. Except for the `msort` program under the N-N-Rp policy, we see only small (possibly insignificant in some cases) performance differences among the three test policies. For some cases (e.g. `hh3d` and `hough`) pull-based migration performs a bit better, while in other cases (e.g. `psolu` and `mandelbrot`) the push-based approaches have the advantage.

Table 5 gives the number of page migrations that occurred under each of the three policies. Unlike the completion time data, we see significant differences in the numbers of page migrations. The push-based strategies tend to migrate more pages. Initially, this is somewhat surprising since all three policies use the same remote-wins criterion to determine when page migrations should occur. Two possible explanations come to mind. One possibility is that not all triggered page faults result in page migrations in the pull-based policy. The other is that the destination nodes for push migrations are poorly selected, and as a result, pages must be migrated more often in an attempt to find the "right" node.

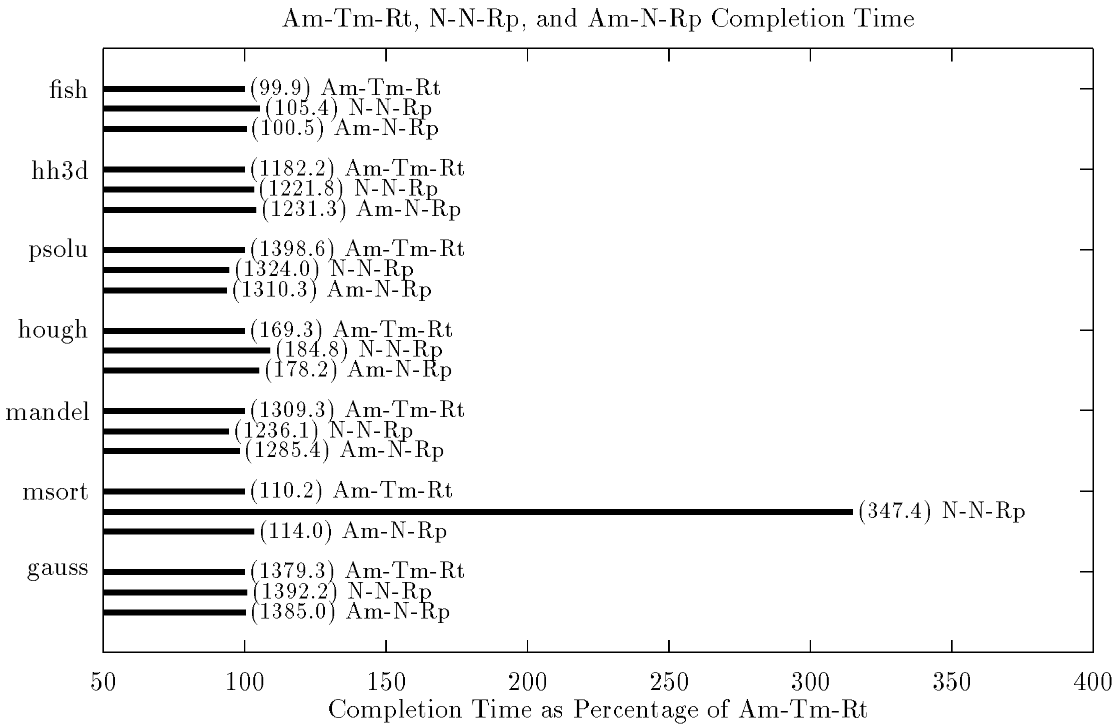


Figure 10: Am-Tm-Rt, N-N-Rp, and Am-N-Rp Completion Time Comparison

Since DUnX does not currently count the number of trigger operations, [5] our first hypothesis cannot be easily verified with experimental data. There are reasons to suspect that the hypothesis is false, however. In order for a trigger not to result in a page migration, the page in question must not be referenced by any remote node until after it has been paged out by the local node. This would imply that many (over 1800 in the `psolu` case) of the push migrations performed by the N-N-Rp and Am-N-Rp policies are wasted, since no possible choice of a destination node would have a process using the page (or else a migration would have occurred under the Am-Tm-Rt policy as well). Based on our knowledge of the application reference patterns, it seems highly unlikely that this could happen.

We suspect that the second hypothesis (i.e. that the reason for the difference is likely due to poor destination choices for the push-based migrations) is the correct explanation. When the pull-based Am-Tm-Rt policy migrates a page, the destination is always a node that has faulted on that page, whereas the destination of a page being push migrated is the remote node that most recently updated the remote usage count for that page. We suspect that the latter method of selecting a destination is more prone to error, thus resulting in subsequent migrations to attempt to improve the situation. The Am-Tm-Rt policy may do less migration because it chooses more appropriate destination nodes.

Most of our test applications (`fish`, `hh3d`, `hough`, `msort` and `gauss`) perform best with the Am-Tm-Rt policy. Of these, the `fish` and `hh3d` programs performed best with the fewest migrations. For the `fish` application, this is because migration seems to be of little use for that application. For `hh3d`, we suspect that Am-Tm-Rt is simply able to migrate pages to appropriate locations in fewer steps. The `hough` and `msort` applications performed best when the number of migrations are highest. These are the only two cases in which Am-Tm-Rt generates more migrations than

[5] This will be added to the DUnX kernel in the near future.

| Application | Am-Tm-Rt | N-N-Rp | Am-N-Rp |
|---|---|---|---|
| fish | 116 | 412 | 349 |
| hh3d | 40 | 588 | 585 |
| psolu | 91 | 1955 | 1915 |
| hough | 44 | 22 | 35 |
| mandel | 13 | 77 | 100 |
| msort | 1084 | 52 | 1061 |
| gauss | 918 | 773 | 955 |

Table 5: Am-Tm-Rt, N-N-Rp, and Am-N-Rp Page Migrations

the other two policies. The interesting point is that the number of migrations of the Am-Tm-Rt policy more closely resembles that of Am-N-Rp than N-N-Rp, suggesting that pulled migrations of inactive pages dominate the activity for these programs. A similar observation can be made for the `gauss` application, although there we see that Am-N-Rp experienced a few more migrations than Am-Tm-Rt. The `psolu` program performs best under the Am-N-Rp policy, when the number of page migrations is quite high. The N-N-Rp policy gives the best performance for the `mandelbrot` program, despite the relatively high (77) number of migrations for that application. In the subsections 5.5 and 5.7, we discuss the relationship between completion time and the number of page migrations for each of our applications.

The page migration data for the `msort` program can be used to explain the N-N-Rp completion time data point for that program. Apparently, the Am-Tm-Rt and Am-N-Rp policies use pull-based migration of inactive pages a great deal, since this is the common feature that distinguishes them from the N-N-Rp policy. The N-N-Rp policy has difficulty migrating pages in use *only* by remote processes, because the page scanners end up inactivating such pages (since the local usage count goes to zero) before they consider the push migration option. Since N-N-Rp doesn't push migrate inactive pages, pages in use only by remote processes end up being re-remactivated as a result of a page fault, only to be inactivated again by the local scanner during the next "move pages" phase. One way to eliminate the problem would be to change the order of the two tests (instead of checking for inactivation and then checking for migration, reverse the order of these tests), but since always pull migrating inactive pages also eliminates the problem (as evidenced by the table 5 data), we chose not to modify the page scanner code at that point in time.

It is important to note that while these results suggest that the Am-N-Rp policy suffers from its inability to successfully select the best destination for each page migration operation, there may be other push-based policies that do not exhibit this difficulty. In the next subsection, we compare three push-based policies.

### 5.4 Constant-Space vs. Linear-Space Push-Migrate Policies

Our large collection of test policies includes only two linear-space dynamic page placement policies: N-N-Cp and Am-N-Cp. Since these policies have much more information available to them when making dynamic page placement decisions, one may expect much better performance from them. On the other hand, such policies do not scale well, since the space overhead per page increases linearly with the number of processors. In this subsection, we compare our linear-space policies to

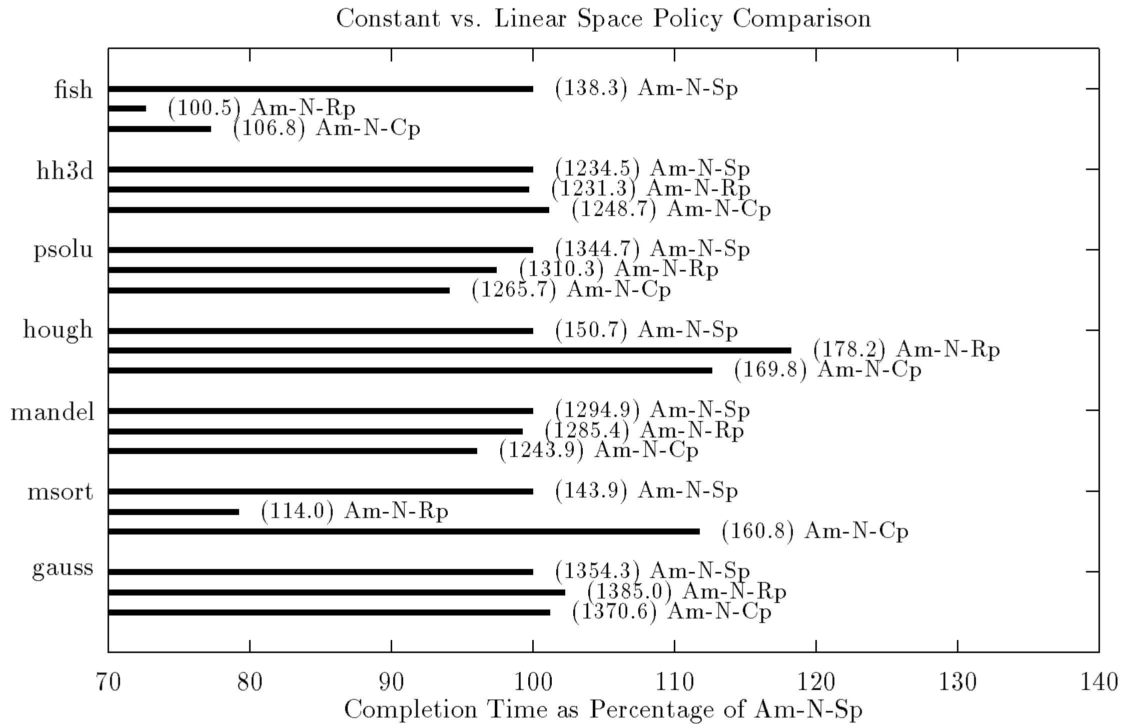


Figure 11: Constant vs. Linear Space Completion Time Comparison

similar constant-space policies.

In figure 11, completion time results (relative to the Am-N-Sp case) for the Am-N-Sp, Am-N-Rp, and Am-N-Cp policies are presented. These policies are identical except for the criteria used by the page scanners to decide when to push migrate shared pages. The Am-N-Sp and Am-N-Rp policies use the simple and remote-wins tests, respectively, to decide when to push migrate pages. The Am-N-Cp policy uses a rough approximation to Black, *et. al.'s* competitive migration algorithm [13].[6]

The results of figure 11 show that while the Am-N-Cp policy does outperform the other push-based migration policies in two cases (`psolu` and `mandelbrot`), it does so by only a small margin (less than 5%). In other cases, at least one of the simpler and more scalable constant-space policies performs better. It is also interesting to note that, for each of our test applications, there is some other constant-space (not push-based) policy that outperforms the two linear-space policies (e.g. the Wm-Wm-SDt policy gives better performance for `hh3d`, `psolu`, `hough`, `msort`, and `gauss`, Am-Bm-St performs better for `mandelbrot`, and Am-N-N gives better performance for `fish`).

The Am-N-Cp policy results show that the volume of data used to make dynamic page placement decisions is not the key to the success of a policy. The Am-N-Sp and Am-N-Rp policies are extremely similar to the Am-N-Cp policy, but use constant-space approximations to the linear-space tests of the Am-N-Cp policy with considerable success. This shows that data summaries can in some cases be used as effectively as the complete data.

Of course, Am-N-Cp is not the only linear-space dynamic page placement policy possible. Other linear-space policies may be developed that perform significantly better than any possible constant-space policy. However, even if such a strategy is developed, the scalability problem must still be

[6] Note, however, that our approximation is definitely *not* competitive. Special hardware is required to implement that algorithm honestly.

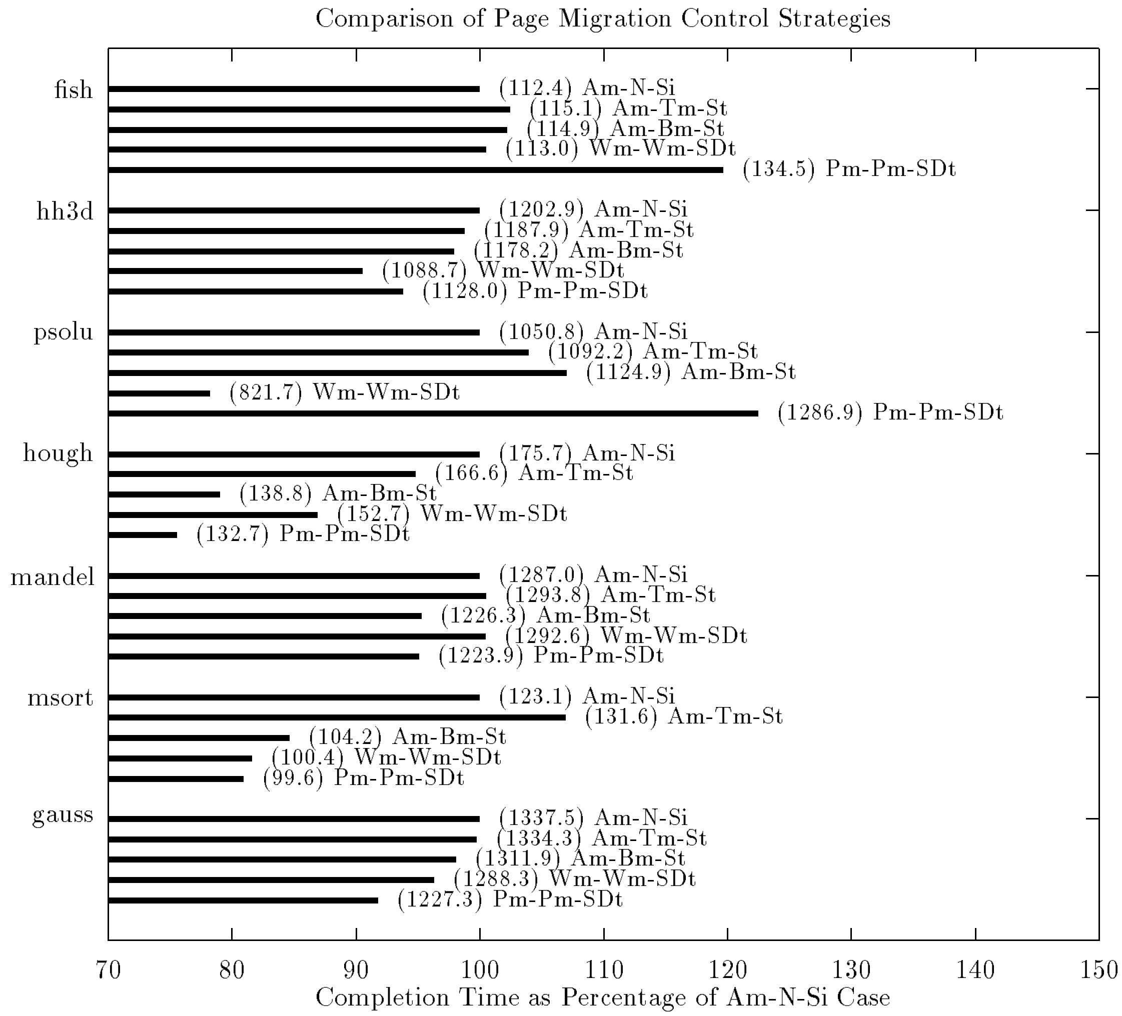


Figure 12: Migration Control Strategy Completion Time Comparison

faced. Since we forsee no immediate solutions to the scalability problem of linear-space policies, we have no plans of further investigating such policies.

### 5.5 Migration Control Methods

Several strategies for controlling page migration are represented in our suite of test policies. Such control may take the form of being selective about which pages are allowed to migrate and/or explicitly addressing the page bouncing problem by limiting migration activity. In this subsection, we compare those strategies.

To motivate the need for control of page migration, we can consider a dynamic page placement strategy that imposes very little control. The Am-Am-N policy ensures that every process references only local memory by always choosing to migrate whenever a fault is taken on a page in a remote frame. The Am-Am-N (migrate on reference) strategy suffers from the page bouncing problem. Our test applications take at least an order of magnitude longer to run under the Am-Am-N strategy (Am-Ar-N suffers from similar problems) than under other more reasonable strategies. This is because some set of pages are continually being migrated from one node to another. Using interactive monitoring tools that we have developed, we can "watch" this page bouncing occur. In some cases, hundreds of pages are migrated each second, leaving the processors very little time to accomplish anything useful.

An obvious question concerning our Am-Am-N results is whether our workload collection just happens to be poorly chosen for this policy. It may seem that applications with well-defined sharing patterns would perform wonderfully under the Am-Am-N strategy. Interestingly enough, `msort` fits this description quite well. Each portion of the primary shared data structure (the array to be sorted) is referenced by at most one node at a time, guaranteed by the synchronization done within the program. The reason `msort` performs poorly under Am-Am-N is that the page containing the synchronization data structure is continually being migrated from one node to another. A quick change to the `msort` source code to prevent movement of that page is enough to correct the problem. Performance of the modified `msort` program under Am-Am-N is quite good (about 108 seconds elapsed time), though not quite as good as some of our other test policies.

Unfortunately, not every page bouncing situation can be eliminated as easily as in the `msort` case. This is the motivation for more sophisticated strategies that limit the amount of page migration from the operating system level.

In figure 12, completion time results (relative to the Am-N-Si policy) for five test policies using different strategies for controlling page migration are summarized. The Am-N-Si and Am-Tm-St policies control migration by allowing only inactive pages and pages for which the page scanner has concluded a migration is appropriate (using the simple test trigger) to be migrated. The Am-Bm-St policy always allows migration of inactive pages, but migrates active or remactive pages only when it appears that pages on the remote source node are being referenced remotely more extensively than pages on the local node. The Wm-Wm-SDt policy controls migration by allowing only warm pages to migrate and freezing pages as they are migrated. Since the page scanners only defrost pages once it appears that another migration may be appropriate (with the SDt defrost trigger), the amount of page migration is limited. The Pm-Pm-SDt policy allows pages to continually migrate (without freezing), as long as the time between migrations is sufficiently long (parameter $t_1$ to the policy). Whenever the migrations become too frequent, the page is frozen.

Missing from this set of policies is Wm-Wm-N, which solves the page bouncing problem by allowing a page to migrate only once (once it migrates it is frozen but never defrosted). Although this policy may perform well (we consider the Wm-Wrm-N-NS variant in section 5.9), it really isn't a dynamic strategy in the same sense as the others being considered in this discussion since placement decisions are reevaluated only a fixed number of times (once, in this case).

In order to interpret the results of figure 12, it is helpful to know the number of migrations that occurred in each case. This is shown in figure 13, where the number of page migrations relative to the Am-N-Si policy is plotted for each case.

The most obvious features of figure 13 are the Pm-Pm-SDt data points. In most cases the number of migrations under the Pm-Pm-SDt policy is far greater than under the other four policies (the exception is `mandelbrot`, which does minimal migrating under any of these strategies). This makes sense, because the Pm-Pm-SDt policy uses the "loosest" control strategy of the five. It is interesting to see how the large numbers of Pm-Pm-SDt policy migrations have affected performance for each of our applications. For the `fish` and `psolu` applications, the Pm-Pm-SDt policy performed poorly as a result of those large numbers of migrations. The `hough`, `msort`, and `gauss` applications, on the other hand, performed best under the Pm-Pm-SDt policy. One question is, "what characteristics of Pm-Pm-SDt cause this behavior?"

There are two parts to the migration control problem. While it is important to limit the amount of migration (to avoid page bouncing situations), it is also important to perform the desirable migrations as soon as possible. Unfortunately, these goals sometimes conflict. One way to avoid page bouncing situations is to slow the rate at which migrations are allowed to occur.

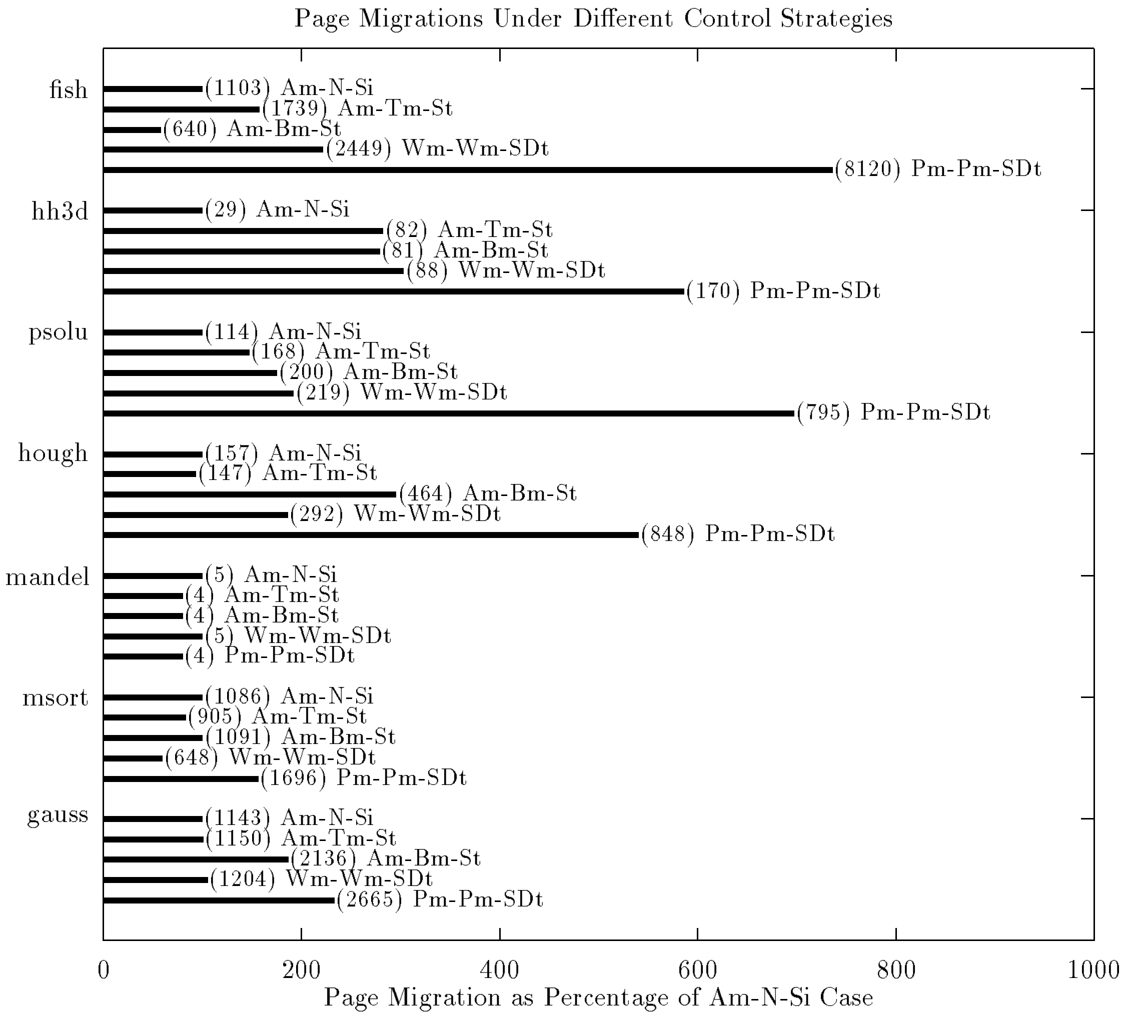


Figure 13: Migration Control Strategy Migration Comparison

Unfortunately, this delays the desirable migrations as well. For example, Wm-Wm-SDt (which is essentially the same as the Pm-Pm-SDt policy with the $t_1$ parameter set to infinity) is better able to prevent excessive page migration than Pm-Pm-SDt, but as a result, it sometimes takes longer to perform desirable page migrations. Another way to avoid page bouncing is to be more selective in choosing pages as candidates for page migrations (i.e. use more conservative page fault trigger tests). In this subsection, we consider only policies based on the simple trigger test. We study the effects of different trigger tests in section 5.7.

It is important to know the abilities of the test policies to handle the two parts of the migration control problem. In order to simplify our discussion, let us consider the following two scenarios:

1. Initially, a shared page $p$ is placed in a frame on node $i$. After a distinct phase change, it becomes desirable to place $p$ in a frame on node $j$ instead (i.e. processes on node $i$ cease referencing $p$ and a process on node $j$ begins heavily referencing it). The desired placement of $p$ changes to node $k$ after a second phase change.

2. Initially, a shared page $p$ is placed in a frame on node $i$. Processes on nodes $i$ and $j$ actively read/write share $p$. Let $\delta$ be the time between references to $p$ from different nodes.

Clearly, in scenario 1 it is desirable to migrate $p$ as quickly as possible at each of the phase change points. Scenario 2 is an example of a reference pattern that would result in page bouncing (migrations of $p$ at the rate $1/\delta$) under the Am-Am-N policy.

The Am-N-Si and Am-Tm-St policies are very good at preventing page bouncing, simply because they allow a page to be migrated no more than once for each time the page scanners run their "move pages" phases. These policies handle scenario 2 quite well, since they lower the rate of page migrations. They are slow to migrate $p$ in scenario 1, however, since they cannot perform the desired migration until after the page scanners detect the phase change and trigger the migration opportunity.

The Wm-Wm-SDt policy can handle the first phase change in scenario 1 quite well, since $p$ is migrated as a result of the first node $j$ reference. It cannot respond as quickly to the second phase change, however, since the first migration results in the freezing of $p$. The second migration is delayed until $p$ is defrosted, which (for the SDt defrost trigger) occurs when the scanner notices that remote processes are referencing $p$ more than local processes.[7] Thus, Wm-Wm-SDt is quick to perform the first migration of a page, but somewhat slower to perform subsequent migrations. The same Wm-Wm-SDt page freezing that delays the second migration of $p$ in scenario 1 prevents the bouncing of $p$ in scenario 2. Like the Am-N-Si and Am-Tm-St policies, Wm-Wm-SDt limits the number of migrations of a page to once for every run of the page scanner "move pages" phase.

If the length of the second phase of scenario 1 is greater than the Pm-Pm-SDt $t_1$ parameter, then Pm-Pm-SDt performs both of the desired migrations of $p$ as soon as possible. Thus, if $t_1$ is set low enough, desired migrations are performed as quickly as possible. Low $t_1$ settings are undesirable for situations like scenario 2, however. If $\delta$ in scenario 2 is greater than $t_1$, page bouncing will result. If $t_1$ is large enough, $p$ ends up frozen and the page bouncing is eliminated. We have not yet systematically studied the effects of the $t_1$ parameter, though we plan to do so in the future.

It is more difficult to understand the Am-Bm-St policy. If the distribution of shared pages is well balanced, Am-Bm-St allows no migration in either of our two scenarios. When the distribution is unbalanced, Am-Bm-St is as quick to migrate page $p$ in scenario 1 as Pm-Pm-SDt. It is unlikely to suffer from page bouncing in scenario 2, however, because migration is allowed only when it improves the distribution of shared pages. It is unlikely that migration of a page will improve the balance in both directions, which is necessary for the page to bounce.

A policy's performance with scenario 1 serves as an indicator of its ability to quickly perform desired migrations, whereas its performance with scenario 2 serves as an indicator of its ability to limit undesirable migrations. By looking at each of the seven applications in turn, we can get a feel for what aspects of the migration control problem are most important for each.

The `fish` and `mandelbrot` results are not terribly surprising, as we have already concluded that dynamic page placement does little to improve the performance of these applications. The `fish` program performs poorly under the Pm-Pm-SDt policy simply because there is too much migration (as a result of the large amount of active sharing in `fish`) that isn't worthwhile. The other four policies also do a fair amount of migration, but each produces about the same level of performance. For the `mandelbrot` program, there simply is no need for page migration and none of the tested policies uses much. They all perform at roughly the same level (within 5% of the Am-N-Si policy), and the migration control strategy appears to be of little importance.

The `hh3d` and `psolu` programs perform best under the Wm-Wm-SDt policy. Since the Am-Tm-St and Am-Bm-St policies use roughly the same amount of migration as Wm-Wm-SDt in both cases but cause the applications to take significantly longer to run, there is some evidence that the speed

[7] Note that if the second phase of scenario 1 lasts longer than the time between runs of the "move pages" page scanner phase, the Wm-Wm-TDt policy (Wm-Wm-SDt with a time-based defrost trigger) is able to perform the second migration of $p$ as quickly as the first. Again, we compare triggering mechanisms in a later subsection.

with which pages are initially migrated is at least somewhat important. This is because none of those policies appear to allow page bouncing for these applications, so Wm-Wm-SDt distinguishes itself from the others primarily by its shorter delay to perform the first desirable migrations. For both programs, the Pm-Pm-SDt policy fails to perform as well as the Wm-Wm-SDt policy. Pm-Pm-SDt is generally quicker to migrate than Wm-Wm-SDt, so speed isn't likely to be the cause for this performance difference. We believe that the likely reason is that the extra page migrations performed by Pm-Pm-SDt are more costly than the benefits they produce.

The `hough`, `msort`, and `gauss` applications perform extremely well under the Pm-Pm-SDt policy, but differ in terms of their performance under the Am-Bm-St and Wm-Wm-SDt policies. Performance of the `hough` program appears to depend primarily on the number of page migrations, since there is a greater correspondence between the numbers of migrations and completion times than between policy quickness and completion times. The opposite appears to be the case for `msort` and `gauss`, since the completion times are more correlated with migration quickness (the Pm-Pm-SDt policy is best, followed by Wm-Wm-SDt, etc.) than the number of migrations performed.

There is another migration control mechanism represented in our collection of test policies that we have yet to mention. Page replication can serve to control page migration in certain situations. If a heavily shared page is not being modified, a policy that chooses to replicate the page is freed from the task of preventing the page from being bounced from node to node. Evidence of the ability of page replication to control migration is presented in the next subsection.

The most important lesson to be learned from the results presented in this subsection is that there is a trade off between the problem of preventing unwanted page migration and the desire to migrate pages as soon as possible. Even in our small workload collection, we have applications for which each is more important than the other, suggesting that it may be difficult to find a single best trade off point. We have not yet studied the effects of varying the $t_1$ parameter of the Pm-Pm-x policies (except for the $t_1 = \infty$ setting implicit in the Wm-Wm-x variations), which serves to adjust the trade off point for this family of policies. Such an investigation may yield insight into this issue.

### 5.6 Replication Control Strategies

Control of page replication is as important as control of page migration. In this subsection, we consider the issues involved in replication control.

The Am-Ar-N policy enforces virtually no control on the amount of page replication. Like the Am-Am-N policy, Am-Ar-N ensures that all memory references are to local memory. Unlike Am-Am-N, the Am-Ar-N policy creates multiple copies of shared pages in preference to moving existing copies. This is essentially the same strategy used by the various distributed virtual memory systems discussed in the literature [32, 33, 36, 10], though the page coherence strategies differ in some cases.

The Am-Ar-N policy performs quite poorly for most of our test applications. The one exception is the `msort` application, which performs quite well under the Am-Ar-N policy (completion time of about 85 seconds). Nearly all of the replications are components of RC migrations, since `msort` does not use much actively-shared memory. This is why the thrashing-like condition experienced by the other applications is not a problem for `msort`.

The `fish`, `hough`, `psolu`, `hh3d`, and `gauss` applications all experience a thrashing-like condition when run under the Am-Ar-N policy. Shared pages are continuously being replicated and then invalidated (as a result of coherency faults), only to be replicated again. Since the policy always chooses to replicate on a read-reference, there is nothing to break the cycle and performance degrades accordingly.

The `mandelbrot` application suffers from a similar thrashing condition that is sufficiently interesting to warrant discussion here. There is only one page involved in the replicate/coherency fault cycle, and there is only one write made to that page. The page in question contains a shared variable used as a spin-lock implementing a barrier at the beginning of the program. As each worker process reaches the barrier, it creates a local replica of the page and begins spinning on the lock until it is released. When the main process attempts to release the lock, it takes a coherency fault on the page (since the children have replicated it) resulting in the invalidation of all the page replicas. Of course, the children immediately fault on the page again (since they are still spinning), and each enters the fault handler. In the fault handler, they are prevented from processing the fault until the main process's coherency fault (on the very same virtual page) is completed and appropriate kernel locks released. Once processing of the main process's coherency fault completes, a race begins between the workers trying to replicate the page (thus removing write access to the page) and the main process trying to release the barrier (by modifying the page). At first, one expects that the main process would easily win the race almost all of the time, since the child processes are just beginning the fault processing. However, the DUnX kernel does some additional processing (possibly even a context switch) before returning from the main process's coherency fault, thus giving the worker processes a fair chance at winning the race. As a result, `mandelbrot` startup is hampered by several iterations of this race, degrading performance considerably in some cases.

Like the migration control problem, there are two aspects to the replication control problem. It is important both to prevent unwanted replications (those that would be soon followed by a coherency fault) and to ensure that desired replications occur as soon as possible. The page migration control mechanisms discussed earlier can also be applied to control page replication. The Wm-Wrm-x and Pm-Prm-x policies are essentially the same as their Wm-Wm-x and Pm-Pm-x counterparts, except that they choose to replicate rather than migrate whenever a page fault is caused by a read reference. Coherency faults are treated as migrations when decisions about freezing a page are made. All of these policies are able to use page freezing to prevent the thrashing-like situations encountered by the Am-Ar-N policy, and generally perform quite well (as evidenced by data shown in figures 8 and 9). We will not present a detailed comparison of these policies here, since such a comparison provides little additional insight into the issues at hand. Not surprisingly, the relative performance of the Wm-Wrm-SDt and Pm-Prm-SDt policies is similar to that of the Wm-Wm-SDt and Pm-Pm-SDt policies (compare figure 14, discussed later in this subsection, to figure 12, discussed in the previous subsection). The only significant difference, which we shall discuss later in this subsection, involves the `psolu` program. First, we consider a technique intended to distinguish between real replication opportunities and RC migration situations.

In an earlier subsection of this paper we described a problem with the Pm-Prm-RDt policy making extensive use of RC migrations (i.e. using replication and coherency fault pairs to effectively migrate pages). This problem is shared with the other Pm-Prm-x policies, as well as the Wm-Wrm-x policies. We have developed the read-only replicate criterion and the corresponding read-only page scanner trigger test to address the RC migration problem. The idea is to dynamically determine when a page is being used in a read-only fashion. The goal is to replicate only pages that aren't being modified, thus eliminating the RC migrations.

The Am-Rr-ROt policy uses the read-only replicate criterion to prevent undesired page replications. Performance of Am-Rr-ROt for our test applications is summarized in table 6, where the numbers of page replications ("R") and coherency faults ("C") is given along with the computation completion time. For reference, the Am-N-N policy (which differs from Am-Rr-ROt only with respect to page replication) completion time is also given. The read-only replicate criterion is clearly successful in preventing most undesired page replications, as evidenced by the small number of coherency faults that occurred for any of our applications. In terms of completion time, the Am-

| Application | Am-N-N | Am-Rr-ROt | | |
|---|---|---|---|---|
| | Time | Time | R | C |
| fish | 99.1 | 98.8 | 11 | 3 |
| hh3d | 1227.3 | 1067.3 | 22 | 2 |
| psolu | 3670.5 | 3605.2 | 8 | 0 |
| hough | 180.1 | 134.7 | 893 | 34 |
| mandel | 1284.9 | 1224.7 | 17 | 2 |
| msort | 162.1 | 204.6 | 7 | 4 |
| gauss | 1310.5 | 489.4 | 288 | 1 |

Table 6: Am-Rr-ROt Performance Summary

| Appl | Wm-Wrm-SDt | | | Wm-RrWm-ROtSDt | | | Pm-Prm-SDt | | | Pm-RrPm-ROtSDt | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | M | R | C | M | R | C | M | R | C | M | R | C |
| fish | 554 | 2103 | 1657 | 2137 | 31 | 30 | 1539 | 10196 | 8084 | 9757 | 230 | 139 |
| hh3d | 16 | 65 | 45 | 62 | 7 | 0 | 18 | 195 | 135 | 144 | 27 | 0 |
| psolu | 71 | 369 | 124 | 193 | 1 | 1 | 98 | 2105 | 959 | 1050 | 29 | 14 |
| hough | 34 | 1224 | 82 | 181 | 270 | 21 | 55 | 2389 | 200 | 508 | 951 | 15 |
| mandel | 0 | 33 | 6 | 2 | 5 | 3 | 0 | 18 | 2 | 4 | 25 | 4 |
| msort | 475 | 165 | 162 | 785 | 57 | 16 | 675 | 1049 | 780 | 1650 | 23 | 3 |
| gauss | 376 | 1098 | 571 | 1019 | 26 | 1 | 56 | 6363 | 631 | 2157 | 130 | 1 |

Table 7: Wm-Wrm-SDt, Wm-RrWm-ROtSDt, Pm-Prm-SDt, and Pm-RrPm-ROtSDt Operations

Rr-ROt policy performs significantly better than Am-N-N for the `hough` and `gauss` applications. Both of these applications experience a relatively large number of true page replications, without suffering from a large number of coherency faults.

The table 6 data show that the read-only replicate criterion can be used to limit the number of coherency faults encountered, but it does not tell us whether or not all the desirable replications are performed. To address this question, let us compare the Wm-Wrm-SDt, Wm-RrWm-ROtSDt, Pm-Prm-SDt, and Pm-RrPm-ROtSDt policies. The relative completion times for our seven applications under these four policies are given in figure 14. The completion time data show that the read-only replicate criterion improves performance for the `fish`, `hh3d`, and `mandelbrot` applications, for both the warm and paced policy variations. It is important to note that these are the same three applications for which the ability to replicate pages did little to improve performance. For the `psolu`, `hough`, and `gauss` applications, the read-only replicate criterion degrades performance for both the warm and paced variations. The `msort` application performs better under Wm-RrWm-ROtSDt than under Wm-Wrm-SDt, but worse under Pm-RrPm-ROtSDt than under Pm-Prm-SDt.

In table 7, the numbers of migrations ("M"), replications ("R"), and coherency faults ("C") for these four policies are presented for each of our test applications. It is readily apparent that the read-only replicate mechanism is able to prevent undesired page replications, since the number

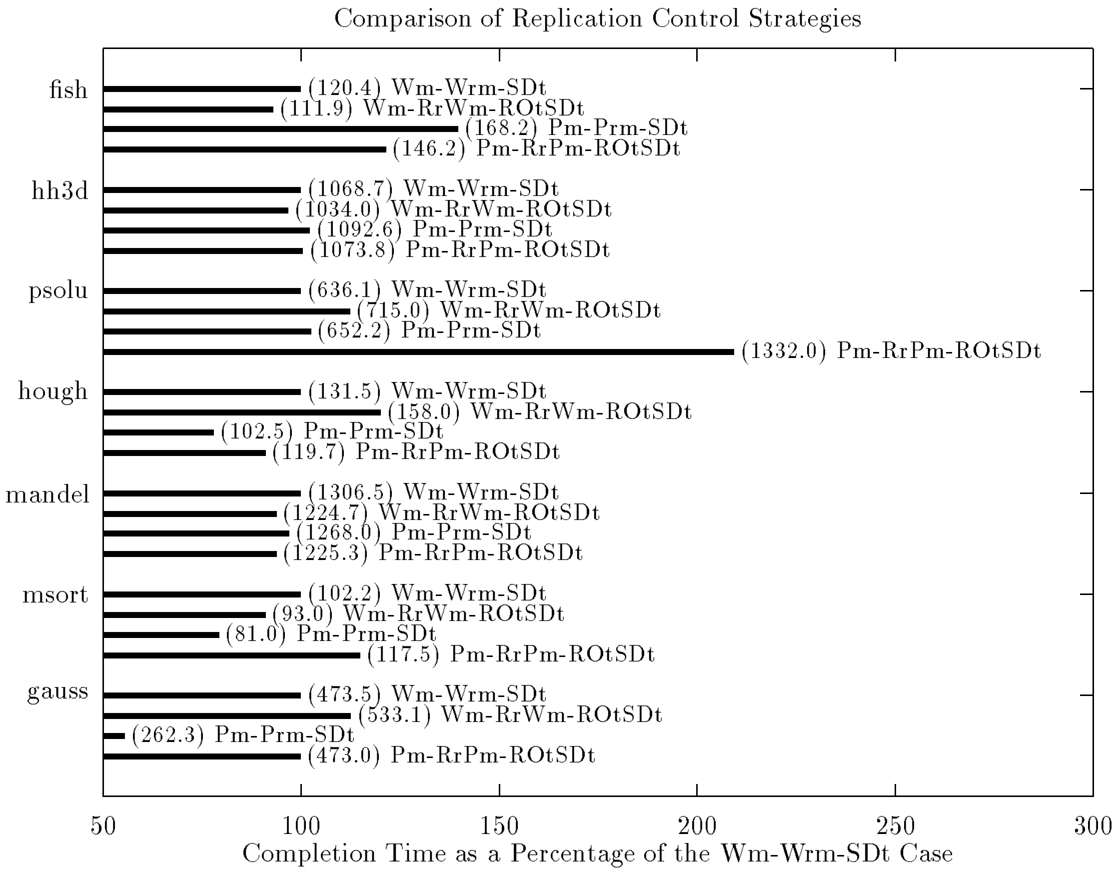


Figure 14: Replication Control Strategy Completion Time Comparison

of coherency faults is much lower for the Wm-RrWm-ROtSDt and Pm-RrPm-ROtSDt policies. Considering the replication and completion time data, however, it appears that the Wm-RrWm-ROtSDt and Pm-RrPm-ROtSDt policies fail to perform some of the desirable replications for the `psolu`, `hough`, and `gauss` applications. This suggests that the read-only replicate policies sometimes conclude that a page has been modified too recently to be replicated when doing so would improve performance. Currently, it takes between 8 and 16 seconds to completely age the modification count on a page (i.e. a page must not be modified for 8 to 16 seconds before it can be declared read-only). Providing quicker aging of the modification count may improve performance, since more pages may be available for replication. We plan to investigate this possibility in the future.

Now we return to the discussion of the `psolu` case mentioned earlier. The data presented in table 7 provide evidence that page replication can serve as a good migration control mechanism. We have already noted that the ability to replicate pages often decreases the amount of page migration activity. In many cases, most of the difference in page migration activity in the presence of a replicating policy can be attributed to RC migrations. In some cases, however, much of the benefit of replication comes from the fact that some degree of page bouncing is eliminated by the replication. This appears to be the case with the `psolu` application (and to a lesser extent, the `gauss` program) under the Pm-Prm-x policies. Table 7 provides the evidence necessary to draw this conclusion. The Wm-RrWm-ROtSDt completion time is only 12% slower than the Wm-Wrm-SDt time, despite the fact that only one page replication occurs under the Wm-RrWm-ROtSDt policy. The Pm-RrPm-ROtSDt policy, on the other hand, gives performance over 100% slower than the

Pm-Prm-SDt policy even though it experienced 29 page replications. Since Pm-RrPm-ROtSDt does more replication than Wm-RrWm-ROtSDt, but performs much slower, the replications themselves are not likely the only reason for the success of the Pm-Prm-SDt. We believe that a primary reason for the success of Pm-Prm-SDt is the decreased page bouncing activity relative to that encountered under the Pm-RrPm-ROtSDt policy (98 vs. 1050 page migrations). The Pm-Prm-SDt policy is using replication to control migration (as suggested in the previous subsection). Since the Pm-RrPm-ROtSDt policy fails to perform some of those migration-controlling page replications, a page bouncing situation develops and performance degrades. In section 5.2, we presented data that show the success of the Pm-Prm-RDt policy for the `psolu` program, relative to the Pm-Pm-RDt policy. We believe that much of that success can be attributed to the migration-controlling effect of the Pm-Prm-RDt page replications.

One page replication control strategy that we have not yet mentioned simply prohibits replicating pages that may be modified. While this ensures that code pages are replicated, it prohibits the system from replicating data pages even when they are rarely modified. Since DUnX always replicates read-only text pages (under any of our test policies), this replication control strategy is of little use in our environment.

The issues that arise when considering migration control strategies also apply to the page replication control strategies. In addition, it is desirable to minimize the number of unnecessary replications (any replication that is soon followed by a coherency fault on that page) and RC migrations, while still ensuring that the desirable page replications occur. We have also found that page replication can itself serve as a migration control mechanism. Based on the results presented in this subsection, we suspect that a good mechanism for choosing between migrating and replicating a page is key to a successful dynamic multiple-copy page placement policy. We plan to pursue this problem further, perhaps by considering alternative parameters to the read-only replicate policies, in future work.

### 5.7 Comparison of Triggering Tests

In section 5.5 we considered different strategies for controlling page migration, but limited ourselves to just one triggering test (the simple test defined in section 3.1). The triggering test, however, plays an important role and in this subsection, we consider several such tests in an effort to understand the advantages and disadvantages of each.

The purpose of a triggering test is to determine when the page scanner should create an opportunity for the reevaluation of an earlier page placement decision. For some policies the test is applied to all pages, while for others (the Wm-Wrm-x and Pm-Prm-x policy groups) the test is applied only to frozen pages. In either case, the purpose of the test is to determine when the page scanner should cause the potential for faults on a page by invalidating virtual to physical mappings to that page.

There are several issues and trade offs to consider. Clearly, we want to provide opportunities to reevaluate decisions as often as necessary, but we don't want the operating system to spend too much of its time making unnecessary decisions. Ideally, we want the page scanner to trigger a decision reevaluation if and only if the placement of the page (possibly multiple copies) should change.

For purposes of this discussion, let us consider the following four policies: Wm-Wm-SDt, Wm-Wm-RDt, Wm-Wm-TDt, and Wm-Wm-LTDt. These policies differ from one another only in the triggering test used by the page scanners. The Wm-Wm-SDt and Wm-Wm-RDt policies attempt to use reference information to determine when best to defrost a page, while the Wm-Wm-TDt and Wm-Wm-LTDt policies base their decisions on the amount of time a page has been frozen. We

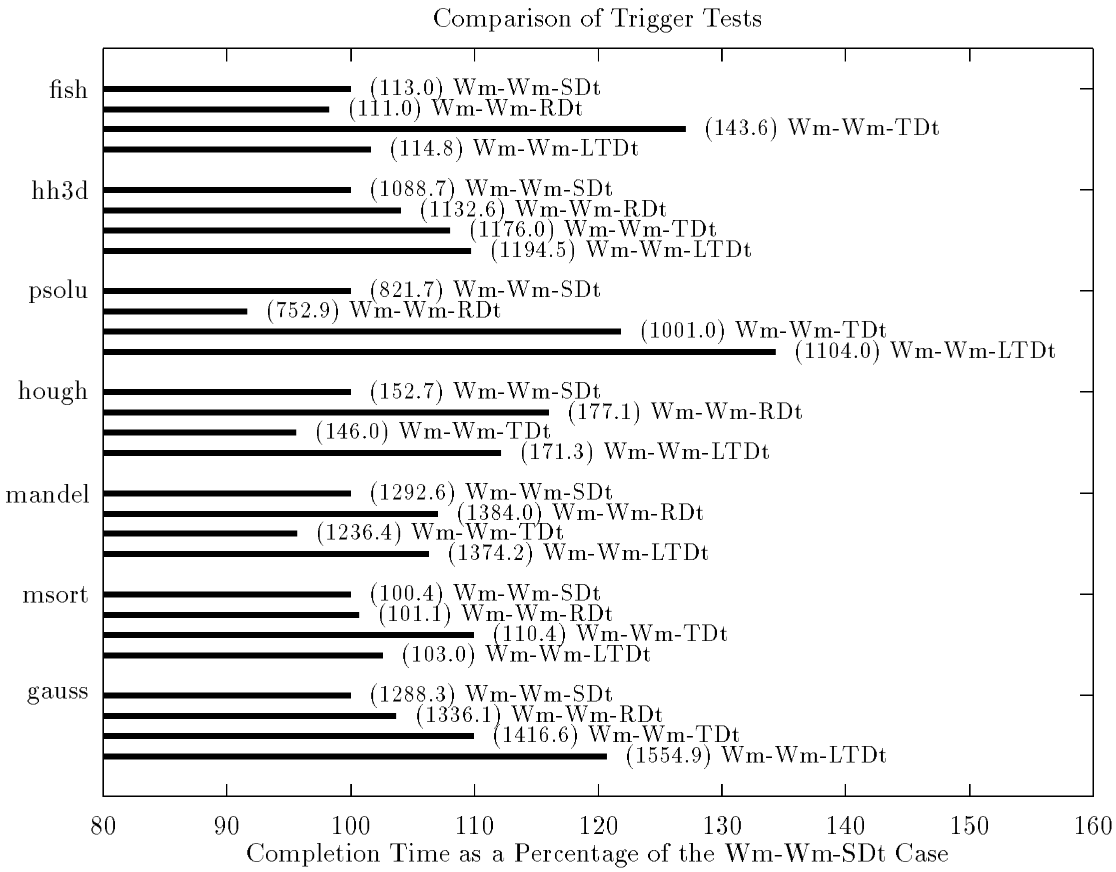


Figure 15: Triggering Test Completion Time Comparison

call these two types *reference-based* and *time-based* triggers, respectively. The Wm-Wm-RDt and Wm-Wm-LTDt policies are more *conservative* than their *liberal* Wm-Wm-SDt and Wm-Wm-TDt counterparts. The conservative triggers use stricter trigger tests than the liberal triggers.

In figure 15, the completion times for our test applications under these four policies is plotted relative to the Wm-Wm-SDt policy. For three of our applications (`hh3d`, `mandelbrot`, and `msort`), performance is within 10% of the Wm-Wm-SDt case for each of the policies. For the other applications, there are cases where different triggering tests degrade performance considerably.

In order to understand the results, it is helpful to look at the numbers of page faults and page migrations for each case. These data are presented in table 8 (the number of page faults is in the "F" column and the number of page migrations is in the "M" column). The table shows that the number of migrations for the more conservative version of each type of trigger (either reference-based or time-based) is usually lower than the more liberal version, as expected. The anomalous case is the `mandelbrot` program, which experiences fewer page migrations under the Wm-Wm-SDt policy than under the more conservative Wm-Wm-RDt policy.

The time-based triggers appear more liberal (in the sense that they allow more page migration) than the reference-based triggers. The Wm-Wm-TDt policy experiences large numbers of page faults as well as the highest number of migrations for all seven applications. For our applications with longer execution times (`hh3d`, `psolu`, `mandelbrot`, and `gauss`), the Wm-Wm-LTDt policy performs more migrations than both the Wm-Wm-RDt and Wm-Wm-SDt policies. Our applications with shorter run times behave differently under the Wm-Wm-LTDt simply because they don't run

| Application | Wm-Wm-SDt | | Wm-Wm-RDt | | Wm-Wm-TDt | | Wm-Wm-LTDt | |
|---|---|---|---|---|---|---|---|---|
| | F | M | F | M | F | M | F | M |
| fish | 23472 | 2449 | 13495 | 1058 | 38845 | 5512 | 13273 | 944 |
| hh3d | 1449 | 88 | 794 | 48 | 31821 | 2493 | 3325 | 315 |
| psolu | 3084 | 219 | 4052 | 159 | 4902 | 608 | 6688 | 404 |
| hough | 3722 | 292 | 3940 | 154 | 5197 | 648 | 3361 | 168 |
| mandel | 776 | 5 | 846 | 12 | 3697 | 372 | 986 | 25 |
| msort | 4506 | 648 | 4676 | 581 | 5120 | 1648 | 4784 | 578 |
| gauss | 9604 | 1204 | 9279 | 809 | 12014 | 2231 | 10967 | 1479 |

Table 8: Wm-Wm-x Page Faults and Migrations

long enough (`fish`, `hough`, and `msort` all have execution times of about two minutes) to experience very many defrost operations.

Five of our application programs (`fish`, `psolu`, `hh3d`, `msort`, and `gauss`) performed best with the reference-based triggers. There isn't a direct correspondence between the relative performance and numbers of migrations for any of these applications, suggesting that the reason for the success of the reference-based triggers has more to do with the choice of pages triggered (which pages and when they are triggered) than simply the number of pages migrated.

When we considered migration control strategies in an earlier subsection, we found that `hough` and `msort` seem to depend on migrations being done quickly to obtain the best performance. The behavior of the `hough` application under the different trigger tests is certainly consistent with that result, since the more conservative tests (which are generally slower to trigger) fail to perform as well as the others. The TDt trigger is at least as quick to defrost a page as any of our other strategies, since it defrosts every frozen page each time it runs. We also know that the SDt trigger is at least as quick to defrost as RDt, since the simple-test must succeed several times for the remote-wins test to succeed. Thus it appears that the two liberal tests perform well in this case because they are quicker to trigger and provide migration opportunities.

The fact that the `msort` program performs poorly under the Wm-Wm-TDt policy does not mean that there is an inconsistency between these results and those of the earlier subsection. Though the Wm-Wm-TDt policy is the quickest to trigger when it is desirable, it is also the quickest to trigger when it is not. The shared page containing synchronization data is the cause of the poor performance of `msort` under Wm-Wm-TDt, since that policy is unable to limit the migration of that page, which migrates from node to node throughout the computation. This extra overhead forced upon `msort` is more costly than the savings achieved by being able to perform the desired migrations quickly.

The results for the `mandelbrot` program are initially confusing, since despite the fact that `mandelbrot` has little need for page migration, it performs best with Wm-Wm-TDt. Actually, it turns out that these differences are relatively insignificant, since the times from multiple runs of `mandelbrot` also vary by as much as 5%. The number of migrations encountered is somewhat interesting, however. The `mandelbrot` migrations involve two shared pages that are modified during initialization and shutdown only (one page contains a vector of pointers to the rows of the image matrix and the other contains program globals such as the image matrix size). Ideally, these pages would be replicated to each node, but none of these policies allows multiple copies of a single page.

Since there is no ideal location for either page, one may expect that it is best to pick a node and leave the pages there for the duration of the program. As these results show, however, the policy that does the most migrations (i.e. Wm-Wm-TDt) performs at least as well as the others. We suspect (but have not yet confirmed) that this is because each process is given its fair chance to have those pages local, and each has to put up with its fair share of having the other processes referencing its memory. Since each process has approximately the same amount of work to do and they all share the costs and benefits associated with the placement of those two pages, the completion times for the processes are approximately equal, thus minimizing the completion time of the longest process. The maximum process average access time to data in those pages is decreased, thus improving performance. This result is particularly interesting because it suggests that page migration may be able to improve performance even when there is no ideal place for each page.

To summarize, for five of our test applications, it appears that the reference-based trigger tests perform somewhat better than the simpler time-based tests. We also see that there are cases in which a more conservative test performs better than its liberal counterpart, as well as cases in which the more liberal version is better.

### 5.8 Comparison of Triggering Methods

In the previous subsection, we compared different tests that the DUnX page scanners can use to decide when to trigger the reevaluation of an earlier page placement decision. In this subsection, we consider two different ways to trigger those reevaluations.

The simplest way to trigger a page placement reevaluation is to inactivate the page in question. All virtual to physical mappings to the page are invalidated, so it is certain that all processes still using the page will fault on it once more, providing the opportunity for a page migration or replication. Instead of inactivating the page, our second triggering method only invalidates the remote virtual to physical mappings. This method avoids causing faults on the page for local processes (which already have local copies of the page anyway), but still ensures that remote processes using the page must fault on it.

The completion time results for two comparisons (Am-N-Ri vs. Am-Tm-Rt and Pm-Prm-TDi vs. Pm-Prm-TDt) are shown in figures 16 and 17. For the most part, we see that in both comparisons neither method demonstrates an obvious superiority over the other. The invalidate remote method differs from the inactivate method by more than 5% in only three cases (`psolu` with Am-Tm-Rt and Pm-Prm-TDt, and `fish` with Pm-Prm-TDt), and it performs better in two cases and worse in the other.

Though the bottom-line performance of both methods is similar, there are other interesting issues to consider. Page fault overhead is an important consideration. At first thought, the invalidate-remote triggering method seems as though it would generate fewer page faults than the inactivate method. This seems especially likely for the time-based triggering tests (as in the Pm-Prm-TDx policies), since the triggering of reevaluations is generally more frequent when using those tests.

In table 9, the number of page faults ("F") and migrations ("M") for all four policies are presented, as well as the number of replications ("R") for the Pm-Prm-TDx policies. We immediately see from the data in this table that our hypothesis about the numbers of page faults under the Am-N-Ri being greater than under the Am-Tm-Rt policy is, for the most part, false. The reason for this is that the Am-N-Ri policy makes better migration choices, whereas the Am-Tm-Rt policy makes some poor choices that must then be corrected by triggering additional faults and migrations (thus explaining the larger number of migrations for the Am-Tm-Rt policy). Under the Am-Tm-Rt policy, the first process to fault on a recently invalidated page is always remote (unless some local process just happens to *begin* using the page), so every time a page scanner triggers a reevaluation

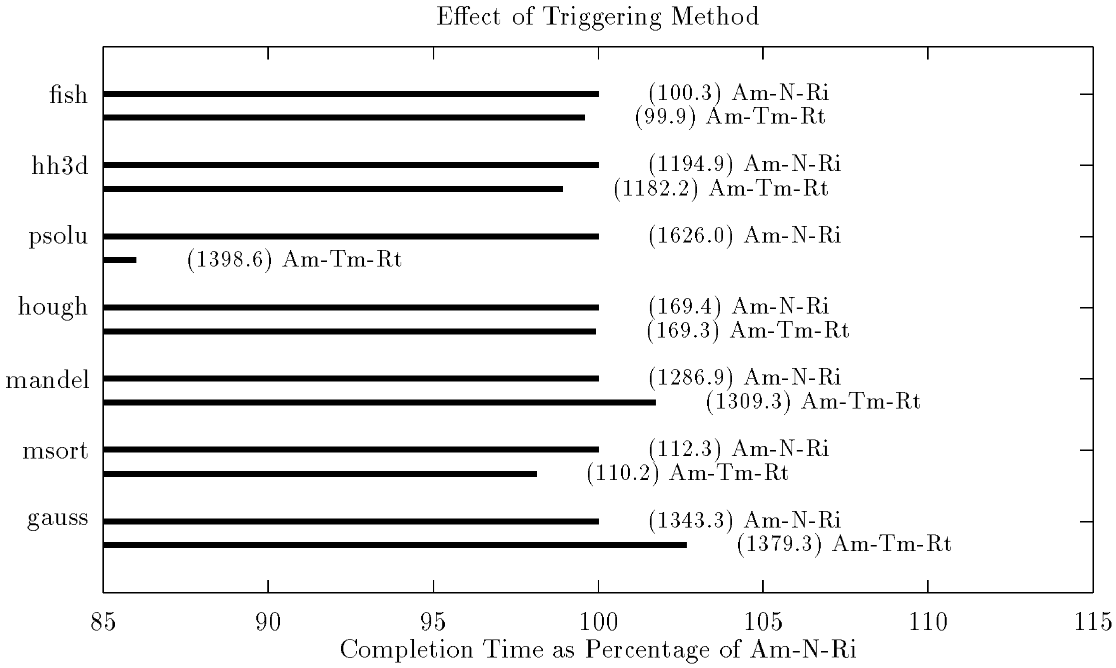


Figure 16: Am-N-Ri and Am-Tm-Rt Completion Time Comparison

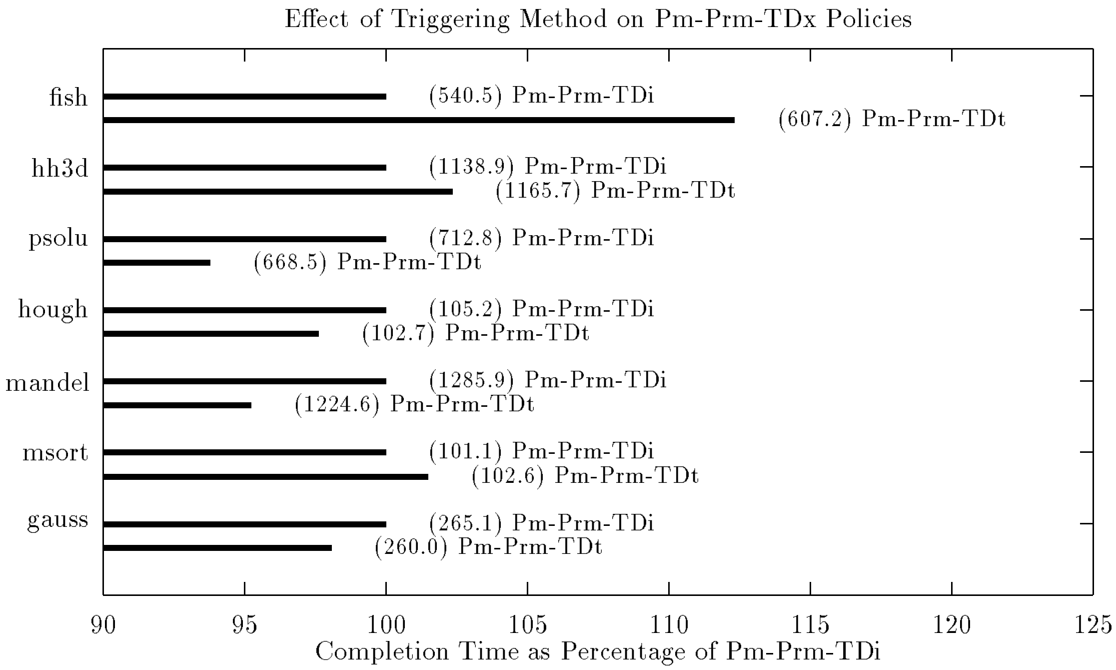


Figure 17: Pm-Prm-TDi and Pm-Prm-TDt Completion Time Comparison

| Appl | Am-N-Ri | | Am-Tm-Rt | | Pm-Prm-TDi | | | Pm-Prm-TDt | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | F | M | F | M | F | M | R | F | M | R |
| fish | 9779 | 66 | 10801 | 116 | 124308 | 12472 | 37032 | 134724 | 14297 | 42048 |
| hh3d | 632 | 10 | 1055 | 40 | 30467 | 2083 | 720 | 24436 | 210 | 3140 |
| psolu | 2879 | 96 | 2768 | 91 | 12216 | 297 | 4149 | 10743 | 98 | 3780 |
| hough | 3105 | 42 | 3090 | 44 | 4237 | 123 | 2353 | 4280 | 72 | 2441 |
| mandel | 837 | 12 | 845 | 13 | 1308 | 68 | 32 | 785 | 0 | 25 |
| msort | 4801 | 1064 | 5238 | 1084 | 4621 | 689 | 1334 | 4669 | 796 | 1150 |
| gauss | 10406 | 905 | 10469 | 918 | 9868 | 52 | 6686 | 9869 | 126 | 6658 |

Table 9: Triggering Method Operation Count Comparison

and a remote process references the page, the page is migrated. Under the Am-N-Ri policy, the local process may fault on the page first, thus reactivating it and preventing a migration. Since the process using the page the most is more likely to reference the page first, a migration will be prevented in most cases where the current node is the best node. Under the Am-Tm-Rt policy, the local node isn't given a chance to prevent the impending page migration, so some unwise choices may be made.

Things are more complex for the Pm-Prm-TDx policies, since these policies allow page replication as well as migration. When a page is defrosted under the Pm-Prm-TDi policy, a remote process faulting on the page migrates it to a local frame. When a page is defrosted under the Pm-Prm-TDt policy, a remote process faulting on the page creates a local copy of that page (unless a write access caused the fault, in which case the page is migrated). The data of table 9 demonstrate this predicted behavior, since we see that in many cases, the number of migrations are lower and the number of replications higher for the Pm-Prm-TDt policy.

The key to which triggering method performs best for the Pm-Prm-TDx policies depends on which operation (migration or replication) is most appropriate for a given fault. If a migration would be the best approach then the inactivate trigger method is best, since the Pm-Prm-TDt policy ends up using an RC migration (thus experiencing an additional fault) instead of the simple migration the Pm-Prm-TDi policy performs. If, on the other hand, a replication is the best choice, the invalidate remote triggering method is likely to be the better choice, since the Pm-Prm-TDi policy uses a migrate-replicate pair (migrate from local to remote and then replicate back to the original source) rather than the single replicate operation the Pm-Prm-TDt policy performs. Obviously, the performance of an entire application under either of these strategies depends on the relative frequency and importance of each type of operation.

The results of figures 16 and 17 show that in general, there is very little difference between the triggering approaches in terms of computation completion time. We have seen, however, that there are differences in the numbers of page faults, migrations, and replications. We suspect that a mechanism using our read-only trigger test could be used to select the trigger method dynamically (on an instance-by-instance basis) to derive some performance advantage.

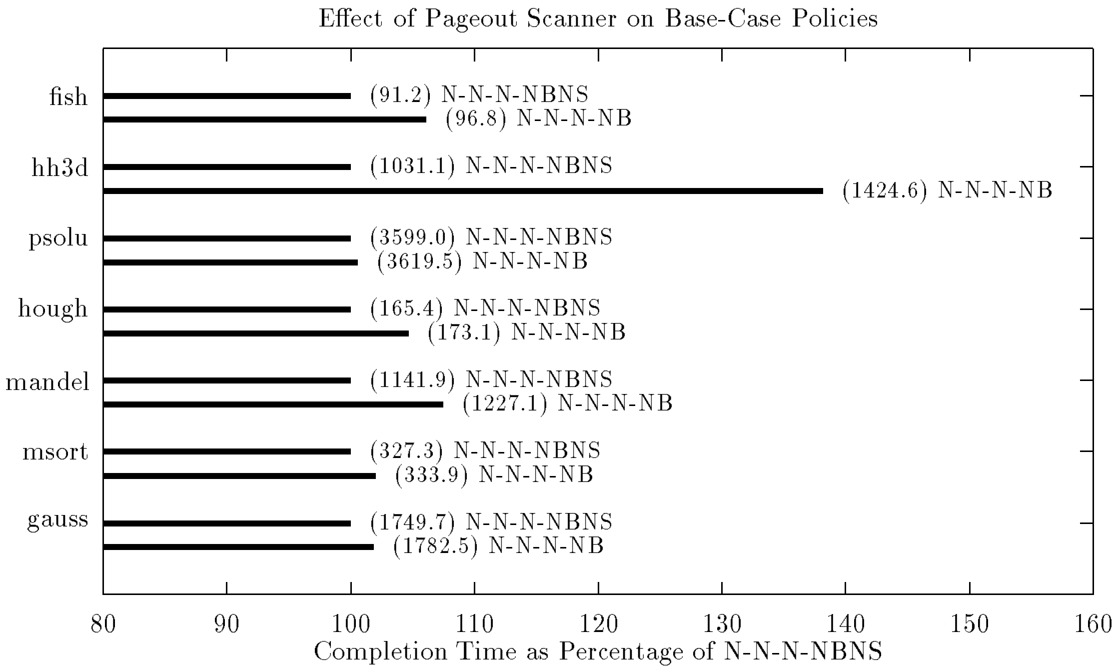


Figure 18: N-N-N-NBNS and N-N-N-NB Completion Time Comparison

### 5.9 Page Scanner Effects

The DUnX page scanners are potentially large sources of overhead, both directly (they require CPU and memory cycles) and indirectly (they often cause page faults). A DUnX page scanner has two primary responsibilities: supporting the page replacement algorithm and handling policy-specific dynamic page placement operations (e.g. triggering page placement decision reevaluations). It is important to understand how each of these page scanner duties affects performance.

The summary data of figure 8 provides immediate evidence of the costs associated with running the page scanners. For each of our test applications, the policy which gave the best performance is one which does not run the pageout scanners (i.e. one of N-N-N-NBNS, N-N-N-NS, Wm-Wrm-N-NS, or Pm-Prm-N-NS). The N-N-N-NBNS and N-N-N-NS policies are the second and third data points of each group in the figure, and the Wm-Wrm-N-NS and Pm-Prm-N-NS policies are the second and fourth data points from the bottom, respectively. In this subsection, we consider the effects the page scanners have on performance.

We begin with a comparison of our static base-case (N-N-N-NB) policy with and without the page scanners running. The completion times for each of our test applications under these two policy options are shown in figure 18. The no scanner (N-N-N-NBNS) policy outperforms the version with the page scanner in every case, though the difference is fairly small in most cases.

The system resources directly used by the page scanners contribute to the noted performance differences. Measurements taken using the kernel event logging facility indicate that this effect is fairly low. For example, the average time for a scanner run with `msort` under the N-N-N-NB policy is about $35ms.$, and since the scanners run only once each second, this amounts to 3.5% of the total CPU time. As mentioned in section 4.3, page scanner overhead varies with the memory reference patterns of the application(s) running. Since the memory reference patterns vary with the memory management policies in use, the page scanner overhead may also vary with the memory management policy selection.

| Application | Policy | Time | Faults | Pageins |
|---|---|---|---|---|
| fish | N-N-N-NBNS | 91.2 | 8995 | 143 |
| | N-N-N-NB | 96.8 | 9294 | 134 |
| hh3d | N-N-N-NBNS | 1031.1 | 552 | 101 |
| | N-N-N-NB | 1424.6 | 3665 | 136 |
| psolu | N-N-N-NBNS | 3599.0 | 2080 | 188 |
| | N-N-N-NB | 3619.5 | 2583 | 302 |
| hough | N-N-N-NBNS | 165.4 | 1888 | 106 |
| | N-N-N-NB | 173.1 | 2808 | 126 |
| mandel | N-N-N-NBNS | 1141.9 | 689 | 76 |
| | N-N-N-NB | 1227.1 | 751 | 119 |
| msort | N-N-N-NBNS | 327.3 | 3749 | 182 |
| | N-N-N-NB | 333.9 | 6023 | 214 |
| gauss | N-N-N-NBNS | 1749.7 | 9098 | 902 |
| | N-N-N-NB | 1782.5 | 35945 | 742 |

Table 10: Page Faults and Pageins for N-N-N-NBNS and N-N-N-NB

Another important reason for the performance degradation caused by the page scanners becomes evident when we consider the number of page faults and pageins that occur under both policies (table 10). Each of the applications encounters more page faults under the N-N-N-NB policy than under the N-N-N-NBNS policy, and in all but two cases (`fish` and `gauss` are the exceptions), they also perform more pageins from backing store. One important cause of the increased faulting behavior is related to the page replacement policy implemented by the page scanners. The decision to inactivate a page is based only on the local usage count (maintained by the scanners by software aging the hardware reference bits). Pages that are "incorrectly" placed are not referenced locally, and end up being inactivated by the local page scanner. The remote processes that fault on those pages simply reactivate (actually, they re-remactivate) them, only to fault on the same pages again a few seconds later (since the local node still isn't referencing them). This effect is especially noticeable with the `hh3d` and `gauss` applications, which suffer from far more page faults when the page scanners are running. Many data pages initialized by the main processes of these programs are all initially placed in frames on that node. The processes that really use the pages execute on remote nodes, however, so there are many "incorrectly" placed pages that cause frequent faults as described above.

Together, these two sources of overhead act against potential performance improvements attained by running the scanners. It is hoped, of course, that the overhead can be kept lower than the performance gains. For the N-N-N-NB policy, performance improvements are expected to come from better page replacement decisions, which could lower the number of page faults and pageins experienced by the applications. For the `fish` and `gauss` applications, it does appear that the scanners are able to better select pages to be replaced, since the number of pageins is lower when the scanners are run. Unfortunately, the extra reactivation page faults caused by the scanners are more costly than those savings.

Earlier, we indicated the the page scanner overhead for `msort` under the N-N-N-NB strategy is about 3.5%. The overall difference in performance between N-N-N-NBNS and N-N-N-NB, however, amounts to just over 1%. This is evidence that there is potential for a performance improvement of about 2.5%, but the costs associated with running the scanners is too great. We suspect that the performance gains would have come in the form of better overall page placements. The early page replacements caused by the scanners provide opportunities for more appropriate placements later on, resulting in improved memory reference locality.

Given that the scanner seems to be of little benefit in these cases, it is reasonable to question this approach to providing page replacement. The replacement policy used when the page scanners aren't run is essentially FIFO replacement, whereas the policy implemented in conjunction with the page scanners is a fair approximation of per-node LRU. One expects the latter policy to make better replacement choices than the former, however, this is not reflected in the performance results because the actual page replacement decisions are relatively unimportant for our suite of applications. The total amount of physical memory available to the applications exceeds the total amount of virtual memory they use. Since much of the virtual address space for each of these applications is shared data space, a page selected for replacement on one node is likely to be faulted into a free frame on another. Once the virtual pages are distributed in this way, no more free frames are required. The page scanners implement a *free pool* replacement policy, however, so they continue to inactivate pages not in use locally even when there is no immediate need to do so. But as we discussed earlier, many shared data pages are poorly placed and are not being referenced by local processes. As a result, they are continually inactivated and faulted upon by remote processes. We suspect that an LRU approximation that also considers the remote usage counts may eliminate this problem (at least partially), though we have not investigated this possibility yet. Otherwise, FIFO replacement at fault time may be adequate.

The page scanners also perform policy-specific operations to help support dynamic page placement. In order to study the effects of these operations, we have compared three paced (Pm-Prm-x) and three warm (Wm-Wrm-x) migration/replication policies. In figure 19 we show the completion time results for the Pm-Prm-TDt, Pm-Prm-TDt-NR, and Pm-Prm-N-NS policies (comparing the timed defrost trigger, the same trigger without page replacement, and without a scanner), and in figure 20 we show the completion time results for the Wm-Wrm-RDt, Wm-Wrm-RDt-NR, and Wm-Wrm-N-NS policies (comparing remote-wins defrost trigger with and without replacement, and without a scanner).

The data in figure 19 indicates that the cost of running the TDt defrost mechanism is fairly high for some of our test applications (compare Pm-Prm-N-NS with Pm-Prm-TDt-NR), as is the cost of running the page replacement mechanism (compare Pm-Prm-TDt and Pm-Prm-TDt-NR). As in the N-N-N-NBNS/N-N-N-NB comparison, additional page faults caused by the page scanners are an important cause of this performance degradation. This is made evident in table 11, where the numbers of page faults ("F"), pageins ("P"), migrations ("M"), replications ("R"), and coherency faults ("C") are tabulated. The column labeled "M+R-C" is the sum of migrations and replications minus the number of coherency faults, and it serves as a measure of the dynamic page placement encountered by the application.

The results for the `fish` and `mandelbrot` applications were not unexpected. We've already found that dynamic page placement does little to improve performance of `fish`, and as expected, additional dynamic page placement activity only degrades performance. Dynamic page placement is of little use to `mandelbrot`, as well, and any gains resulting from the dynamic placement are offset by the costs associated with the extra page faults encountered.

For the `hh3d`, `psolu`, and `msort` applications, it is evident that the page scanners enable a greater amount of dynamic placement to occur (see the table 11 "M+R-C" column). Unfortunately, they also cause a great number of page faults and some additional pageins, whose cost exceeds the

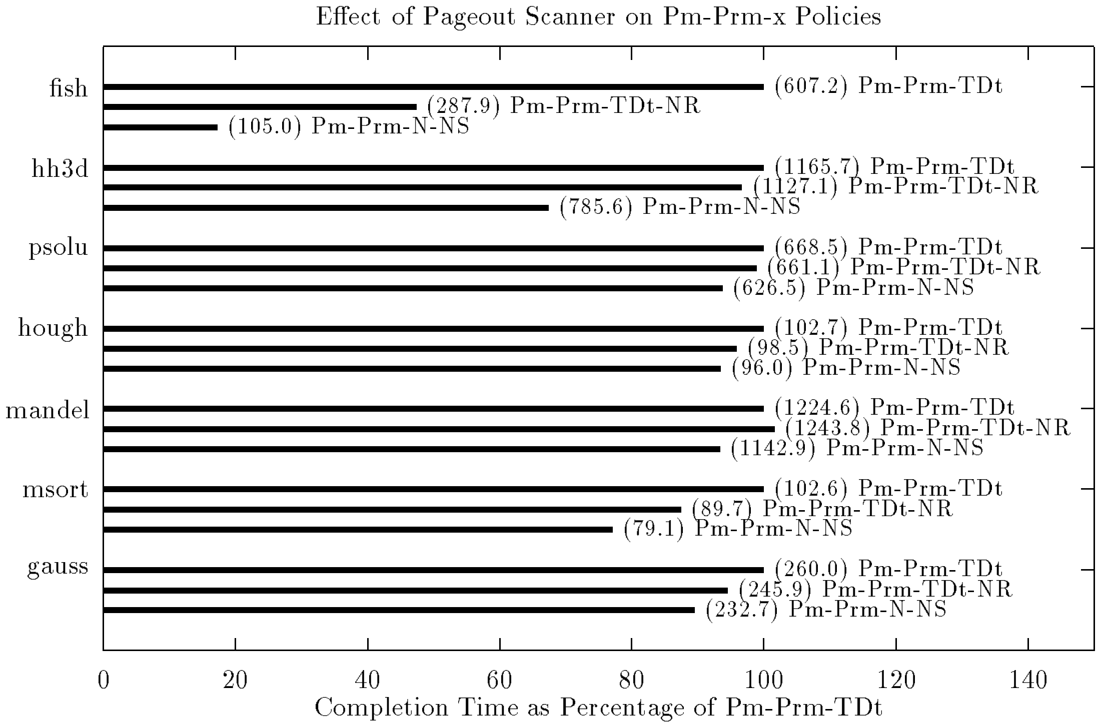


Figure 19: Pm-Prm-TDt-x Policy Completion Time Comparison

| Application | Policy | Time | F | P | M | R | C | M+R-C |
|---|---|---|---|---|---|---|---|---|
| fish | Pm-Prm-TDt | 607.2 | 134724 | 122 | 14297 | 42048 | 35395 | 20950 |
| | Pm-Prm-TDt-NR | 287.9 | 76034 | 100 | 3386 | 22220 | 18847 | 6759 |
| | Pm-Prm-N-NS | 105.0 | 13449 | 101 | 424 | 2824 | 2383 | 865 |
| hh3d | Pm-Prm-TDt | 1165.7 | 24436 | 136 | 210 | 3140 | 1523 | 1827 |
| | Pm-Prm-TDt-NR | 1127.1 | 25689 | 114 | 137 | 3276 | 1804 | 1609 |
| | Pm-Prm-N-NS | 785.6 | 618 | 114 | 17 | 48 | 27 | 38 |
| psolu | Pm-Prm-TDt | 668.5 | 10743 | 266 | 98 | 3780 | 2012 | 1866 |
| | Pm-Prm-TDt-NR | 661.1 | 11500 | 188 | 78 | 3995 | 2152 | 1921 |
| | Pm-Prm-N-NS | 626.5 | 5231 | 176 | 48 | 1671 | 839 | 880 |
| hough | Pm-Prm-TDt | 102.7 | 4280 | 140 | 72 | 2441 | 229 | 2284 |
| | Pm-Prm-TDt-NR | 98.5 | 4098 | 106 | 54 | 2417 | 216 | 2255 |
| | Pm-Prm-N-NS | 96.0 | 3564 | 95 | 41 | 2315 | 165 | 2191 |
| mandel | Pm-Prm-TDt | 1224.6 | 785 | 106 | 0 | 25 | 4 | 21 |
| | Pm-Prm-TDt-NR | 1243.8 | 723 | 77 | 0 | 25 | 4 | 21 |
| | Pm-Prm-N-NS | 1142.9 | 696 | 77 | 0 | 11 | 1 | 10 |
| msort | Pm-Prm-TDt | 102.6 | 4669 | 491 | 796 | 1150 | 690 | 1256 |
| | Pm-Prm-TDt-NR | 89.7 | 4342 | 268 | 358 | 1527 | 978 | 907 |
| | Pm-Prm-N-NS | 79.1 | 4231 | 192 | 304 | 1391 | 978 | 717 |
| gauss | Pm-Prm-TDt | 260.0 | 9869 | 344 | 126 | 6658 | 595 | 6189 |
| | Pm-Prm-TDt-NR | 245.9 | 9815 | 104 | 2 | 7029 | 681 | 6350 |
| | Pm-Prm-N-NS | 232.7 | 9809 | 103 | 2 | 7069 | 720 | 6351 |

Table 11: Pm-Prm-TDt-x Policy Operation Count Comparison

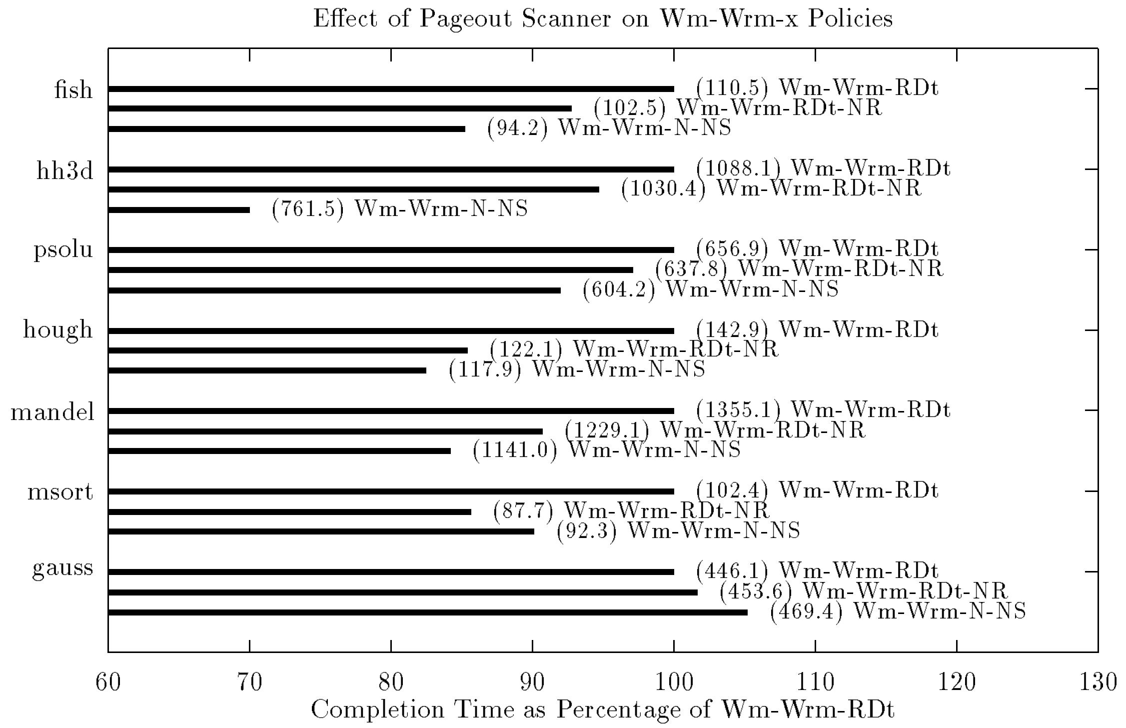


Figure 20: Wm-Wrm-RDt-x Policy Completion Time Comparison

potential savings of the extra dynamic page placement activity. For these three programs, the extra dynamic placement might have paid off if the costs associated with running the page scanners were lower.

The page scanners did not increase the amount of dynamic page placement activity for the `hough` and `gauss` applications as much as it did for our other test applications. The amount of page fault activity did increase, however, thus resulting in decreased performance. The fact that dynamic placement activity did not increase with the page scanners under the Pm-Prm-x policies implies that there would be little need for running them for these two applications, even if the costs associated with running them were completely eliminated.

The Wm-Wrm-RDt, Wm-Wrm-RDt-NR, and Wm-Wrm-N-NS comparison in figure 20 and table 12 gives slightly different results. For the `fish` and `mandelbrot` programs, the results are similar to the Pm-Prm-TDt results. Extra dynamic placement activity serves little purpose, and the costs are detrimental to the overall application performance.

For the `hh3d` and `hough` applications, we see that the page scanners enable a bit more dynamic placement activity, but that the extra page fault costs exceed the resulting savings. The scanners appear to have little effect on the amount of dynamic activity for the `psolu` program, but they still have significant costs that come in the form of additional page fault activity.

The `msort` and `gauss` programs provide our first positive results. The Wm-Wrm-RDt-NR policy outperforms Wm-Wrm-N-NS for both applications, and Wm-Wrm-RDt also outperforms Wm-Wrm-N-NS for `gauss`. The page scanners provide increased dynamic placement activity with benefits that exceed the additional costs associated with running the scanners. This result is significant, since it shows that there are cases which can benefit from the presence of a page scanner.

The delay between page scanner runs is an important parameter. For all of the experiments

| Application | Policy | Time | F | P | M | R | C | M+R-C |
|---|---|---|---|---|---|---|---|---|
| fish | Wm-Wrm-RDt | 110.5 | 14243 | 134 | 339 | 969 | 756 | 552 |
| | Wm-Wrm-RDt-NR | 102.5 | 12769 | 100 | 323 | 1102 | 886 | 539 |
| | Wm-Wrm-N-NS | 94.2 | 10207 | 113 | 318 | 792 | 623 | 487 |
| hh3d | Wm-Wrm-RDt | 1088.1 | 1090 | 125 | 25 | 71 | 33 | 63 |
| | Wm-Wrm-RDt-NR | 1030.4 | 904 | 101 | 18 | 69 | 31 | 56 |
| | Wm-Wrm-N-NS | 761.5 | 584 | 114 | 14 | 37 | 14 | 37 |
| psolu | Wm-Wrm-RDt | 656.9 | 3934 | 281 | 40 | 290 | 112 | 218 |
| | Wm-Wrm-RDt-NR | 637.8 | 2535 | 175 | 39 | 337 | 126 | 250 |
| | Wm-Wrm-N-NS | 604.2 | 2411 | 188 | 35 | 296 | 111 | 220 |
| hough | Wm-Wrm-RDt | 142.9 | 2779 | 129 | 19 | 900 | 38 | 881 |
| | Wm-Wrm-RDt-NR | 122.1 | 2462 | 93 | 22 | 941 | 43 | 920 |
| | Wm-Wrm-N-NS | 117.9 | 2289 | 106 | 19 | 808 | 37 | 790 |
| mandel | Wm-Wrm-RDt | 1355.1 | 795 | 122 | 0 | 15 | 2 | 13 |
| | Wm-Wrm-RDt-NR | 1229.1 | 715 | 76 | 0 | 18 | 3 | 15 |
| | Wm-Wrm-N-NS | 1141.0 | 693 | 89 | 0 | 7 | 1 | 6 |
| msort | Wm-Wrm-RDt | 102.4 | 4729 | 135 | 335 | 244 | 242 | 337 |
| | Wm-Wrm-RDt-NR | 87.7 | 4091 | 101 | 197 | 722 | 440 | 479 |
| | Wm-Wrm-N-NS | 92.3 | 4092 | 113 | 176 | 401 | 401 | 176 |
| gauss | Wm-Wrm-RDt | 446.1 | 9279 | 129 | 99 | 1310 | 625 | 784 |
| | Wm-Wrm-RDt-NR | 453.6 | 9199 | 103 | 1 | 1551 | 685 | 867 |
| | Wm-Wrm-N-NS | 469.4 | 9204 | 106 | 0 | 1299 | 679 | 620 |

Table 12: Wm-Wrm-RDt-x Policy Operation Count Comparison

discussed in this section, the page scanners are run once each second. To determine the effects of this parameter setting, we also considered delays ranging from 0.1 to 20 seconds. Our results indicate that shorter delays degrade performance. As the delays grow larger (above 3 or 4 seconds for most applications), performance is essentially the same as it is without the page scanners.

The results presented in this subsection seem to indicate that the page scanners can be a burden on the DUnX virtual memory system. While the extra faults caused by the scanners can be beneficial (as in the Wm-Wrm-RDt `gauss` and `msort` results), for most of our experiments they serve only to degrade performance. We do not believe that it is appropriate to completely write off the potential usefulness of page scanners, however, since the potential they offer for performance improvements under large, multi-phased workloads seems quite significant. Unfortunately, we have no experimental data to support this intuition at this time.

# 6 Conclusions

We have taken an experimental approach to evaluating the effectiveness of OS-based memory management policies in addressing the NUMA problem. The DUnX kernel has been designed to support a wide range of dynamic multiple-copy page placement policies with user-level control on a per-cluster basis. Policies are composed of individual policy pieces isolated in distinct policy modules. Special "policies" are provided for gathering statistics about the behavior of the system. The programming interface of BBN's delivered systems has been maintained in DUnX so that it can host an established user community and run existing programs. These aspects of the DUnX design have proved valuable in making it a productive environment for experimental research. The status of DUnX is that it has been running as our production OS since November 1989, supporting the entire Butterfly user community at Duke University while also serving as the vehicle for our study.

Using a workload consisting of real application programs contributed by various users of BBN multiprocessors from across the country, we have tested 45 different policy combinations. These include (at least approximations of) most policies proposed in the NUMA memory management literature. The migration algorithm presented by Black, Gupta, and Weber [13] is approximated by Am-N-Cp. Cox and Fowler's policies in [19] are essentially Pm-Prm-TDi and Pm-Prm-N-NS. The simple strategy described by Bolosky, Scott, and Fitzgerald [14] (with threshold of 1) is Wm-Wrm-N-NS. The distributed shared memory form of migration and replication policies (e.g. [32]) are Am-Am-N and Am-Ar-N.

Our results show that there are memory management policies implemented in our system that can improve the performance of programs written using the simpler uniform memory access (UMA) programming model. While achieving the level of performance of a highly tuned NUMA program is still a difficult problem, some of our examples come close. There appears to be no single policy that can be considered the best over our set of test applications. Among the most promising candidates are policies in the x-x-x-NSNR and Pm-Prm-x classes.

Investigations into the contributions made by individual policy features toward overall behavior of our workload provide some insight into the design of a set of effective policies. Some key observations include:

1. Replication is an effective technique for improving application performance, even when it is essentially doing migration through more expensive RC migrations.

2. The read-only replication mechanism can be used to limit the number of unnecessary replications, coherency faults, and RC migrations. However, with the current implementation, it may be too slow in detecting a change to read-only and, thus, fail to perform some of the desired replications.

3. There is generally little completion time performance difference between push and pull-based migration (with one exception), although push-based policies tend to migrate more pages. There appears to be little benefit to the extra information available from linear-space policies over constant-space policies.

4. Migration and replication control methods vary in their quickness to perform desired operations and their ability to prevent undesired migrations and page bouncing. The best strategy depends on the access patterns of the particular program. Replication itself can be viewed as a migration control strategy.

5. Triggering tests can be categorized by the type of information used to make the decision (referency history vs. time) and by the strictness (conservative vs. liberal) of the test. The reference-based triggers tend to be somewhat more successful than the time-based triggers. More attention needs to be paid to distinquishing between migration and replication situations since that also affects the best method to be used for triggering.

6. Finally, the page scanner can be extremely costly. Although necessary to the reevaluation of placement decisions, its overhead often negates the benefits of any improved locality achieved.

There is still considerable work to be done in this area. This study alone has raised a number of additional questions about OS policies that deserve attention. We are currently investigating several of them. A complete solution of the NUMA problem requires research on architectural issues, compiler assistance, and language constructs, as well.


### Acknowledgements

The authors wish to thank Rich Schaaf, David Barach, and Robert Wells for their assistance in providing the nX virtual memory subsystem as a basis for our work. We also thank the BBN Science Office for facilitating the close working relationship between the BBN Operating System Development group and the NUMAtic Project at Duke. Finally, we thank everyone who contributed application programs for our workload. This group includes Anselmo Lastra, James Wilkes, and David Kotz (members of the Duke Butterfly users community), Bill Bolosky, Alan Cox, and Rob Fowler (University of Rochester) and Kevin Bowyer (University of South Florida).


## References


[1] A. Agarwal and A. Gupta. Memory-reference characteristics of multiprocessor applications under Mach. In *Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 215–225, Santa Fe, NM, May 1988.

[2] A. Agarwal, R. Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 119–127, June 1986.

[3] D. Barach, R. Wells, T. Uban, and J. Gibson. Highly parallel virtual memory management in large-scale shared memory multiprocessors. To Appear ICPP 1990, August 1990.

[4] BBN. *Butterfly Parallel Processor Overview.* Cambridge, MA, June 1985.

[5] BBN. The Uniform System approach to programming the butterfly parallel processor. Technical Report number 6149, Bolt Beranek and Newman Adv. Computers Inc., October 1985.

[6] BBN. *Inside the Butterfly GP1000.* Cambridge, MA, October 1988.

[7] BBN. Software tools for Mach 1000. Technical report, Bolt Beranek and Newman Adv. Computers Inc., March 1988.

[8] BBN. *Inside the BBN TC2000.* Cambridge, MA, February 1990.

[9] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. Technical Report COMP TR89-99 and ELEC TR 8928, Rice University Departments of Computer Science and Electrical and Computer Engineering, November 1989.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. Technical Report COMP TR89-98 and ELEC TR 8927, Rice University Departments of Computer Science and Electrical and Computer Engineering, November 1989.

[11] R. Bisiani and M Ravishankar. PLUS: A distributed shared-memory system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 1990. To Appear.

[12] R. Bisiani and M Ravishankar. Programming the PLUS distributed-memory system. In *Proceedings of the 5th Annual Distributed Memory Computing Conference*, April 1990.

[13] D. Black, A. Gupta, and W-D Weber. Competitive management of distributed shared memory. In *Spring COMPCON 89 Digest of Papers*, pages 184–190, 1989.

[14] W. Bolosky, M. Scott, and R. Fitzgerald. Simple but effective techniques for NUMA memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[15] K. Bowyer. private communication. 1990.

[16] M. Brorsson. Performance impact of code and data placement on the IBM RP3. Technical Report Research Report RC 14651 (#65734), IBM Research Division, June 1989.

[17] D. Cheriton, H. Goosen, and P. Boyle. Multi-level shared chaching techniques for scalability in VMP-MC. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 16–24, June 1989.

[18] D. Cheriton, A. Gupta, P. Boyle, and H. Goosen. The VMP multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 410–421, June 1988.

[19] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with Platinum. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32–43, December 1989.

[20] A. K. Dewdney. Computer recreations. *Scientific American*, pages 18–24, July 1984.

[21] S. Eggers and R. Katz. A characterization of sharing in parallel programs and its applicability to coherency protocol evaluation. Technical Report UCB/CSD 87/387, UC Berkeley, December 1987.

[22] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. Cedar—a large scale multiprocessor. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 524–529, August 1983.

[23] J.R. Goodman and P.J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, Honolulu, HI, 1988.

[24] M. Holliday. Page table management in local/remote architectures. In *ACM SIGARCH Int. Conf. on Supercomputing*, pages 1–8, July 1988.

[25] M. Holliday. Reference history, page size, and migration daemons in local/remote architectures. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pages 104–112, April 1989.

[26] A. K. Jones, R. J. Chansler Jr., I. Durham, P Feiler, and K. Schwans. Software management of Cm* - a distributed multiprocessor. In *1977 National Computer Conference*, volume 46, pages 657–663, 1977.

[27] D. Kotz. private communication. 1989.

[28] R. P. LaRowe Jr. and C. S. Ellis. Dynamic page placement in a NUMA multiprocessor virtual memory system. Technical Report CS-1989-21, Duke University, October 1989.

[29] R. P. LaRowe Jr. and C. S. Ellis. Page placement policies for NUMA multiprocessors. *Journal of Parallel and Distributed Computing*, to appear.

[30] A. A. Lastra. *Parallel Numerical Solutions of Discontinuous Models: Capturing the Inherent Parallelism of Biological Processes.* PhD thesis, Duke University, 1988.

[31] T. LeBlanc, M. Scott, and B. Marsh. Memory management for large-scale multiprocessors. Technical Report 311, Dept. of Computer Science, Univ. of Rochester, March 1989.

[32] Kai Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 2, pages 94–101, 1988.

[33] Kai Li and Richard Schaefer. A hypercube shared virtual memory system. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I–125–132, August 1989.

[34] R. McGrath and P. Emrath. Using memory in the Cedar system. Technical Report CSRD rpt. no. 655, Center for Supercomputing Research and Development, U of Illinois, June 1987. Also in Proceedings of Int'l Conf. on Supercomputing, Athens, Greece, June 1987.

[35] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[36] U. Ramachandran, M. Ahamad, and M. Y. A. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II–160–II169, August 1989.

[37] C. Scheurich and M. Dubois. Dynamic page migration in multiprocessors with distributed global memory. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 162–169, June 1988.

[38] A. J. Smith. Cache evaluation and the impact of workload choice. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 64–73, June 1985.

[39] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. Cm* - a modular, multi-microprocessor. In *1977 National Computer Conference*, volume 46, pages 637–644, 1977.

[40] J. T. Wilkes. BLITPAK II: A parallel implementation of block iterative methods for sparse systems of linear algebraic equations. Master's thesis, Duke University, December 1989.

[41] A. W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987.

[42] Pen-Chung Yew. Architecture of the Cedar parallel supercomputer. Technical Report CSRD 609, Center for Supercomputing Research and Development, Univ. of Illinois, August 1986.