Memory Page Placement Based on Predicted Cache Behaviour in CC-NUMA Multiprocessors

by

Robert Ho

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Memory Page Placement Based on Predicted Cache Behaviour in CC-NUMA Multiprocessors

Robert Ho

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2004

An important characteristic of CC-NUMA multiprocessors is the relative difference in latency between local and remote memory accesses. For many applications running on these systems, the amount of time spent stalled on remote memory accesses can make up a significant fraction of the total execution time. Previous work has shown that proper placement of pages in memory can reduce much of this time by changing remote memory accesses to local memory accesses. This work has also shown that such placement decisions are most effective when they are based on the caching behaviour of those pages. In this thesis, we present a new method of predicting such caching behaviour at allocation time, and making appropriate placement decisions based on these predictions. This method required minimal additions to the memory subsystem of the University of Toronto Tornado operating system, and no special hardware for monitoring the memory hierarchy. We also show that this method can result in improvements of up to 35% in total execution time over traditional placement policies such as first-touch placement when the data sets of the applications being run exceeds the size of a local memory node. These results hold for both single application and multiprogrammed workloads.

Contents

Chapter 1

Introduction

In recent years, there has been a growing acceptance of using multiprocessor systems as general purpose compute servers. This use of a powerful central compute server shared by multiple users represents a shift from the distributed networks of workstations that have dominated offices and labs for over a decade. The network of workstations arrangement itself supplanted an earlier centralized model of computing which organized inexpensive terminals around powerful mainframe systems. The decline of the mainframe based arrangement and subsequent adoption of the distributed model was largely driven by significant increases in the cost-to-performance ratio of microprocessors, making it possible to place powerful workstations on the desks of individual users at relatively competitive costs. Such an environment gave users greater control over their own computing environments, rather than relying on a heavily contended central resource.

The willingness to explore the use of multiprocessor compute servers and a return to a more centralized model of computing has risen due to a variety of factors. Distributed networks of personal workstations have large administration costs, requiring considerable expenditure for such things as maintenance, user support, and configuration. While multiprocessor servers are not simple to administer and maintain, an administrator of such a system must deal with only a single hardware system, making system upgrades, software troubleshooting, and other administrative tasks easier. Additionally, the amalgamation of computing resources that a central server represents can be more efficient than a network of individual workstations. Reliance on personal workstations often means that data and resources are duplicated in the system. For example, each workstation in a distributed environment must have enough memory to run the largest application it can potentially be foreseen to run, even if its typical workload requires less memory. A centralized compute server can make do with less memory and other resources than the aggregate resources of a distributed network because these resources are shared amongst all users.

Faced with these issues, it can make sense in some environments to move toward using central compute servers to run large jobs by multiple users. Multiprocessors make good choices for compute servers because they aggregate a large number of resources in a tightly coupled system. The tight coupling of the resources allows fine grained sharing of resources and more flexibility for the operating system in scheduling. However, multiprocessor architectures introduce new considerations for resource management that previously did not exist in uniprocessor servers. In the following sections, we describe one of these considerations that we will be examining in this dissertation: non uniform memory access and data locality.

1.1 CC-NUMA Shared Memory Multiprocessors

While the evolution of the multiprocessor has spawned a variety of research directions, the most popular architectures for medium- and large-scale systems often fall into the class of multiprocessors known as Cache Coherent Non Uniform Memory Access, or CC-NUMA. CC-NUMA is a subclass of the shared memory type of multiprocessor, describing any shared memory multiprocessor that: (1) employs a hardware based cache coherence protocol, and (2) exhibits significant variance in latencies for memory accesses. A cache coherence protocol is a mechanism which ensures that all writes to shared cache lines are seen by processors in the same order. The NUMA characteristic arises because scaling shared memory systems to larger sizes generally requires that main memory be physically distributed throughout the system. The consequence of this is that memory latencies vary depending on the location of a processor in relation to the data it accesses.

The shared memory CC-NUMA multiprocessor has become a popular choice for a central compute server for several reasons. First and foremost, the shared memory environment enables users to execute existing uniprocessor applications without modification. This is extremely important given the tremendous base of existing uniprocessor code. A single address space view also provides a familiar environment for programmers used to a uniprocessor environment. This can facilitate the writing of new uniprocessor or parallel code, or the parallelization of existing uniprocessor code. Finally, the aggregation of memory, I/O, and processor resources in a tightly coupled environment allows for greater flexibility than in a network of workstations. For example, an application now has access to a larger pool of memory than if it ran on a workstation, or an application can be parallelized to take advantage of the greater processor resources in the multiprocessor environment.

1.2 The Impact of NUMA Latencies on Memory Allocation

Since memory stall time can often make up a significant portion of an application's overall execution time [59], the way that data is distributed in a CC-NUMA multiprocessor can have a large effect on the performance of an application. A policy as simple as placing a page in the local memory of the first processor that accesses it, known as *first-touch placement*, has been shown to improve performance by as much as 40% over round-robin placement[1] in some multi threaded scientific applications [39]. Such a policy is based on the assumption that the first processor to access a page tends to be the source of the majority of accesses to that page over the lifetime of an application.

Although first-touch placement can be a simple and effective policy for page placement, more recent work has shown that using cache miss rates to influence page placement can further significantly improve performance. In particular, Verghese has shown that the placement of data in memory not local to the processor making the majority of accesses can have a negative impact on performance if the processor cache miss rate for that data is high [56]. In such cases, it is preferable for the data to be located in local memory rather than remote memory, so that the cache miss latency can be minimized.

Since first-touch placement does not explicitly consider cache miss rates, the memory distribution resulting from the application of this policy can be suboptimal when a process accesses more pages than are available in the local memory node. Under first-touch placement, page frames from a memory node are allocated as a result of page faults from locally running processes until there are no more unallocated frames. Subsequent local page faults are then satisfied using memory frames from remote memory nodes, if available. In other words, first-touch placement implicitly allocates local memory frames to page faults on a first come first served basis. If the total memory demand by the local processes does not exceed the available local memory, then such a policy has no negative impact on the performance of these applications. However, if the demand is greater than the amount of available local memory, the OS must either decide to replace pages in local memory, or begin satisfying subsequent page faults with remote memory frames. Previous work has shown that allocating pages in remote memory is preferable to initiating page replacement on the local memory node [31]. However, if these subsequent page faults are to pages with high processor cache miss rates, their allocation in remote

[1]This type of placement allocates each consecutive page to a different memory node in round-robin order. This is done to evenly distribute allocated pages and avoid *hot spotting*, a condition where many accesses over a short period of time are sent to the same node.

memory can have a significantly negative effect on performance.

1.3 Results and Contributions

In this dissertation, we propose a new page placement policy for CC-NUMA operating systems called *cache aware placement*. Cache aware placement improves on first-touch placement by explicitly considering cache miss rates when placing pages. These miss rates are not directly observed in the cache hierarchy, but inferred by observing page level accesses in the operating system. We begin by examining a variety of scientific applications running on the University of Toronto NUMAchine multiprocessor [21], as well as a simulator based on this environment, to determine how these applications are affected when there is not enough local memory to satisfy their demands, forcing some of their data to be placed on remote nodes. While some work has been done on improving data locality in CC-NUMA multiprocessors [40, 55], most of this research has not dealt with situations where the demand for local memory exceeds the amount available. Under conditions where there are limited local memory frames and some pages must be placed in remote memory, we have found that first-touch placement can result in as much as a 35% increase in execution time compared to a placement policy that minimizes the number of remote memory accesses.

In our proposed placement policy, we make predictions regarding the processor cache miss rates to pages by examining the ordering of an application's page faults. For the applications in our test suite, we have found that if we divide user allocated memory into contiguous regions of pages, there exists a strong correlation between sequential fault patterns in a region and low processor cache miss rates for the pages of that region. Cache aware placement utilizes this correlation by observing the first N page faults to a region, where N is a preset threshold much smaller than the size of the region in pages, and classifying its access pattern as either sequential or non-sequential. If a sequential fault order is observed (indicating a low cache miss rate for that region), we satisfy the remaining page faults to that region with remote memory frames, leaving room in local memory to allocate those pages of other regions with expected higher cache miss rates. We show that under this policy, we can significantly improve on first-touch placement in many applications under both single application and multiprogrammed environments when there is insufficient local memory to satisfy the locally running processes.

The main contributions outlined in this dissertation are:

1. The design of a method for predicting whether a region of memory will have a low processor cache miss rate based on its page fault patterns.

2. The design and implementation of a page placement policy for CC-NUMA multiprocessors that incorporates this prediction method.

3. Validation that this placement policy can significantly improve on first-touch placement in an existing CC-NUMA hardware environment.

1.4 Overview of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 presents background and work related to our investigations. We briefly examine the main characteristics of the CC-NUMA class of multiprocessors, as well as describe memory placement optimizations such as first-touch placement, and page replication and migration. Chapter 3 outlines the shortcomings of first-touch placement that we are attempting to address with our research. Chapter 4 describes a novel method of predicting processor cache miss rates in memory regions, and its incorporation into a new page placement policy, called *cache aware placement*. Chapter 5 gives our experimental methodology and compares the results of cache aware placement against first-touch placement on a variety of benchmark applications in both single application and multiprogrammed workloads. Finally, we present our conclusions and items for future research in Chapter 6.

Chapter 2

Background and Related Work

The recent history of mainstream multiprocessor research and design can be divided into two paradigms: *message passing*, and *shared memory*. We begin this chapter with a brief discussion of these two design philosophies, with an emphasis on the commercially popular *cache coherent non-uniform memory access*, or CC-NUMA, subclass of the shared memory class. Following this discussion, we summarize previous research in reducing the impact of non-uniform memory latencies on performance in CC-NUMA multiprocessors. We conclude this chapter by discussing how our own work relates to ongoing research in the area of memory management not specifically related to the tolerance of memory latency.

2.1 Message Passing and Shared Memory Multiprocessor Architectures

In recent years, the two most widely used architectures for medium- and large-scale multiprocessor systems have been the *message passing* and *shared memory* architectures. Message passing systems employ several processing nodes, each of which typically resemble a traditional uniprocessor computer, joined together by a dedicated network.

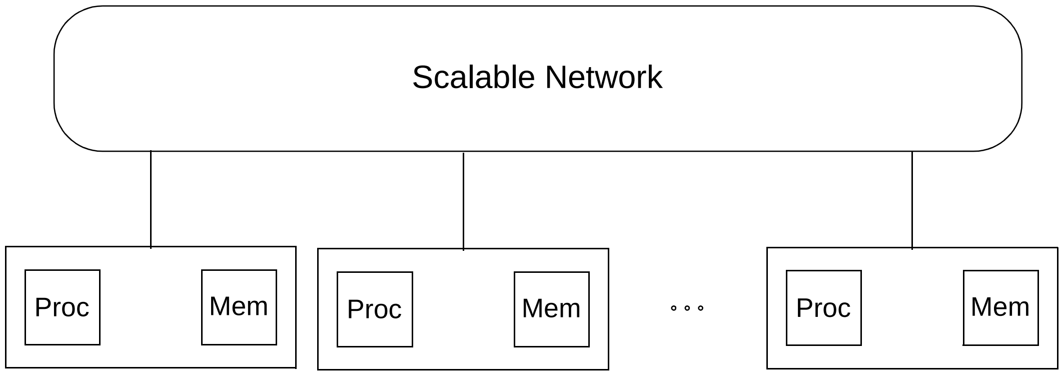


Figure 2.1: Message Passing Multiprocessor architecture.

Communication between nodes is explicit through user coded message sending. Shared memory multiprocessors can be organized in a variety of different physical configurations. However, in all shared memory multiprocessors, communication between processing elements or nodes is implicit by accessing shared memory through memory addressing. In the following sections, we discuss the distinguishing characteristics of each architecture, as well as their respective strengths and weaknesses.

2.1.1 Message Passing

The message passing (MP) class of multiprocessor architectures involves several processing nodes connected together by a dedicated high speed network. Each of these processing nodes typically resembles a complete uniprocessor computer (i.e., a single processor with associated cache hierarchy, memory, and I/O devices)[1][15]. The topology of the interconnect joining these nodes can be one of several forms, such as a mesh, hypercube, or some form of hierarchical arrangement. Figure 2.1 shows a typical arrangement of processors and memory in a message passing multiprocessor.

The fundamental characteristic of the MP architecture is the implementation of the

[1]Strictly speaking, not all nodes need complete I/O capabilities. In most cases, the number of I/O devices attached to each node will vary, and it is often the case that some nodes are used for I/O exclusively.

interface between processing nodes and the network, and the programming paradigm that this imposes on the user. Multiprocessors employing the MP architecture do not give the user a global view of memory; rather, each processing node can only directly access its local memory, and not the memories located at other nodes. Communication between processing nodes is accomplished by explicit messages sent from one node to another. This message passing is usually accomplished by the issuing of *send* and *receive* calls by the communicating processes.

At its simplest, a send call specifies a local buffer containing data to be transmitted and the identifier of the intended recipient process. A matching receive call specifies a local buffer for transmitted data to be copied into, and possibly the identifier of the expected sending process. Send and receive calls can also be synchronous or asynchronous. The former implies that a process will block on a send call until its message is consumed by a receive call at the recipient process. Similarly, a process will block on a receive call until it has received a message communicated by a send call by another process. Asynchronous sends and receives are nonblocking. The use of synchronous sends and receives requires that the programmer be careful to ensure that a process does not block indefinitely waiting for a message to arrive or be received. Asynchronous messages allow for greater concurrency, but may require additional synchronization to ensure correctness.

2.1.2 Shared Memory

In contrast to message passing architectures, shared memory architectures present a view of memory to the programmer that is very similar to that seen in a uniprocessor. In shared memory systems, physical memory is viewed as a single physical global address space accessible by all processors. Interprocessor communication is not explicit, as in the message passing paradigm, but implicit through the accessing of shared memory, e.g., when two processors access data at the same location in the global address space. Additionally, unlike MP systems, each processor in a shared memory system can access

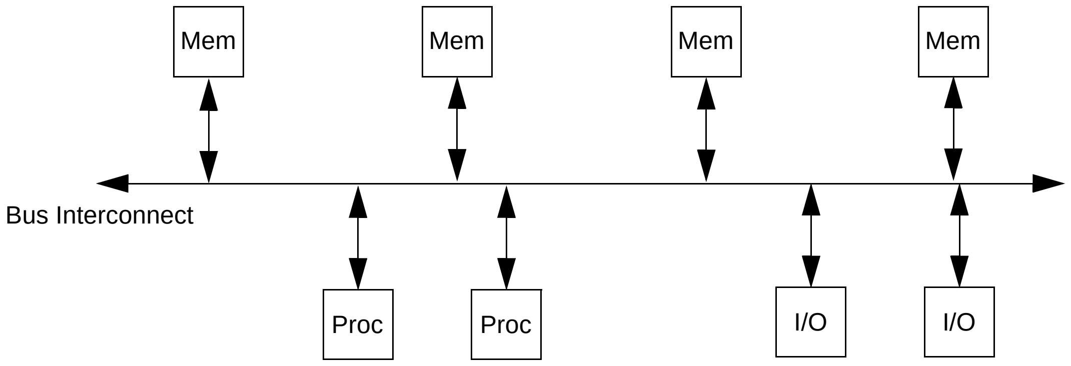


Figure 2.2: Symmetric Multiprocessor architecture.

any memory module through traditional *load* and *store* operations. The specific module that is accessed by these operations depends solely on the target physical addresses of these instructions.

Shared memory multiprocessors can be physically organized in several ways. For small-scale systems of up to 4 processors, the most popular arrangement is the bus-based *symmetric multiprocessor*, or SMP (Figure 2.2). SMP systems are organized around a central memory bus adapted to support multiple processors. Each processor is equidistant from all memory locations, and experiences the same latency for each memory request (hence the term *symmetric*).

Although the SMP architecture is a popular choice for small-scale systems due to its relatively simple design and ability to provide high speed access to memory, bus bandwidth becomes a limiting factor as these systems add more processors [15]. As shared memory systems move to medium- and large-scale designs, a more scalable network is needed to allow efficient memory access. Typically, these systems consist of several processing nodes made up of one or more processors and associated local memory, connected together by a scalable network. Shared memory multiprocessors of this type with a single processor per processing node tend to resemble the MP architecture shown in Figure 2.1; processing nodes with more than one processor tend to resemble the SMP architecture. One of the distinguishing characteristics of these larger scale systems is that the latency for accessing memory that is located in another processing node is greater than that for accessing memory in the local processing node, a dichotomy that is often described as *remote* and *local* memory. For this reason, this architecture is often referred to as the *Non Uniform Memory Access*, or NUMA, architecture.

Cache Coherence

As in the case of uniprocessors, the presence of a cache hierarchy in a shared memory multiprocessor helps to alleviate processor stall time due to memory accesses. In almost all modern computer systems, the latency of a main memory access is two orders of magnitude greater than the processor cycle time, meaning that processor stalls due to these accesses can have a significant impact on performance. Since the rate of increase in processor speeds continues to outstrip the rate of improvement in memory speeds, this relative difference will only increase, making the efficient use of the cache hierarchy a paramount concern.

The reliance on cache hierarchies to hide memory latencies is of even greater importance in the case of shared memory multiprocessors than in uniprocessors due to their typically higher memory access latencies, as well as their reliance on high frequency synchronization primitives that require low latency to provide good performance. However, the presence of caches in a shared memory multiprocessor can lead to situations where processors have inconsistent views of the data in memory [15]. This problem arises because a processor may write data to a cache block that is not immediately updated in main memory or reflected to other caches that have a valid copy of that block. Cache coherence protocols provide a mechanism which ensures that all processors have an identical view of shared memory. Two common mechanisms for this are *invalidation* and *updating* [50]. In an invalidation protocol, all shared instances of a cache block are invalidated prior to allowing a write to proceed to that block. In an update protocol, a write to a shared block is propagated to all other shared instances of that block. These two mechanisms can also be combined into a *hybrid* protocol, which switches between these two mechanisms depending on the type of sharing occurring [48].

The implementation of a cache coherence protocol typically involves tracking the sharing state of all cache blocks in the system, and possibly maintaining a directory that tracks the location of all blocks. For SMP systems, maintaining this state information is facilitated by the presence of the shared bus which allows processors to observe all memory accesses in the system. However, hardware implementations of cache coherence for larger non-bus-based NUMA systems can be more challenging, and the earliest NUMA systems either did not include coherent caches or relied on software to enforce cache coherence. As research in this area has progressed, modern NUMA systems have added hardware support for cache coherence, leading to their designation as *Cache Coherent Non Uniform Memory Access*, or CC-NUMA, multiprocessors.

2.1.3 Message Passing vs. Shared Memory

The message passing paradigm's explicit use of sends and receives for interprocessor communication makes all communication overhead observable by the programmer. The advantage to such an approach is that the programmer has complete control over when these overheads are incurred, increasing his or her ability to tune an application. However, a potential drawback to the message passing paradigm is that programmers experienced with sequential programming for uniprocessors may find the need for explicit communication of data between processing nodes to be foreign. Additionally, while the most straightforward way of adding interprocess communication is to send messages as soon as data is ready to be sent, the relatively high cost of send and receive calls[2] makes it important to minimize the number of these calls. Thus, combining several messages into a single send is preferable to sending many small messages. This can make the adjustment

[2]These are generally expensive system calls that can require copying between address spaces, as well as to and from the hardware network interface.

to the message passing paradigm even more difficult for the programmer.

Conversely, the main benefit of the shared memory architecture is the familiarity of the shared memory paradigm. Since this view of memory closely resembles that of a uniprocessor, programmers familiar with the latter environment tend to have less difficulty adapting to a shared memory multiprocessor environment. On the other hand, shared memory multiprocessors introduce their own complications in the transition from the more common uniprocessor experience. As discussed earlier, the problem of *cache coherence* is a non-trivial complication that can require complex hardware and/or software to solve [20, 52]. Additionally, since the shared memory paradigm relies on the implicit communication of data rather than explicit communication as in message passing, programmers can unknowingly incur communication overheads due to coherence actions on shared locations. This can be illustrated by the example of a shared counter variable that is modified by several sharing processors. If these writes are mostly interleaved between the processors, then each write will cause large numbers of either invalidations or updates to be sent to each instance of the shared counter. Splitting the counter into a local variable for each processor may not alleviate these overheads if these local counters are co-located in the same cache block. Since the granularity that coherence actions occur on is a cache block, updates to each counter still result in invalidations or updates to each instance of the cache block. This problem is known as the *false sharing* problem, as the variables are separate memory locations and are shared only by virtue of being on the same cache block. In either case, the coherence overhead of sharing in this manner can have a severe impact on performance that increases exponentially as the number of sharing processors increases [6].

For the most part, recent trends suggest that the shared memory design has overtaken the message passing architecture in popularity, and the CC-NUMA class of systems has become especially popular amongst commercial medium- and large-scale designs. Some of the commercial CC-NUMA systems available today include the SGI Origin 3000 [28], Sun Fire 15k [11], Alpha GS320 [18], IBM NUMA Q [38], and the HP Superdome [12]. In the past, the efficient implementation of cache coherence has been an overriding concern for researchers working on improving performance in CC-NUMA multiprocessors. A poorly chosen cache coherence protocol can result in network congestion, high memory contention, and low cache hit rates, all of which can significantly affect overall system performance. However, even given an efficient cache coherence protocol, the relative cost of accessing remote memory versus local memory can also have a profound impact on application performance. For example, the latency of remote memory reads compared to local memory reads can range from approximately 3 times greater (SGI Origin 3000 [28], Sun Fire 15k [11]) to 7 to 10 times greater (Alpha GS320 [18], IBM NUMA Q [38]) in modern CC-NUMA systems. The difference between remote and local write latencies can be even greater on misses to shared cache lines due to the need for coherence actions that are typically more costly when the line comes from a remote memory page. While there have been substantial efforts made in investigating coherence protocols, the minimization of memory stall time due to remote memory latencies has drawn less attention, particularly from operating systems researchers. In the following section, we discuss the issue of remote memory latencies in CC-NUMA multiprocessors, as well as some of the recent research into reducing or eliminating its impact.

2.2 Dealing with Remote Memory Access Latencies

The relative cost of accessing remote memory in a CC-NUMA multiprocessor compared to that of accessing local memory can have a significant impact on overall application performance. For example, early work done on investigating different page placement strategies on CC-NUMA multiprocessors showed that for some scientific applications, the choice of a static placement scheme could result in as much as a 40% difference in overall execution time due to changes in the number of remote memory accesses [39].

Intuitively, a reasonably good placement for a page should be in the local memory of the processor that accesses it most frequently.[3] This will mean that all cache misses from that processor can be satisfied by local, rather than remote, memory accesses. However, in practice, placing all of the pages accessed by a given processor in its local memory may not be possible, e.g., there may not be enough local memory to hold all of the memory pages needed by an application. Even if it is possible to place all these pages in local memory, process migration may move accessing processes to a different node than the one that the process originated on, resulting in memory pages that were once local to that process becoming remote pages. Finally, two or more processors with different local memories may access the same page during the application's lifetime.

In the following sections, we outline a number of strategies that were devised to reduce the number of remote memory accesses. We begin by examining the concept of software-based page placement, including both static methods where the placement scheme is fixed for the application's duration, as well as dynamically adaptive placement schemes that involve migration and replication of pages during an application's execution. Following this, we examine hardware-based approaches to improving memory locality, ranging from simple additions to the memory hierarchy such as network caches, to complete reorganization of the memory architecture, as is the case for the *Cache-Only Memory Architecture* [53]. Finally, we briefly examine similarities between page placement and page replacement.

2.2.1 Memory Placement

One straightforward approach to reducing remote memory accesses and the increased latencies associated with them is to attempt to place data in a memory node that is local to the processor or processors that will access it most frequently. To facilitate this,

[3]This assumes that the system provides no mechanism for dynamic page migration or replication, which will be discussed in Section 2.2.2.

some languages, such as High Performance Fortran (HPF) [37], allow arrays to be placed within the system according to a user defined distribution. In HPF, users can specify *decompositions* which are distribution templates upon which the dimensions of an array can be aligned. These decompositions can be used to distribute the various dimensions of an arrays amongst the processors of a system in a block or cyclic distribution. For example, a user could specify that a two-dimensional array be distributed in block fashion in the first dimension, and cyclically in the second dimension. Block distribution of the first dimension would split each column of the array into even blocks, with each block assigned to a different processor. Cyclic distribution of the second dimension would assign each successive element in a row to a different processor in round-robin fashion.

Relying on programmers to specify the distribution of their data arrays presupposes that their knowledge of the data access patterns of their code will allow them to determine the best distribution. Although programmers are likely to have more intimate knowledge of their applications than can be properly extracted by a parallelizing compiler today, it is often the case that programmers will not sufficiently understand the complex nuances or consequences of their algorithms as it relates to the specific system that they are programming. Even under the most optimistic scenario, we would expect that programmers would need several attempts at tuning their code to achieve good performance on a particular system with respect to locality decisions. Additionally, this tuned code would not necessarily run effectively on a differently configured multiprocessor system. In any case, even given a programmer's ability to correctly tune their code to make good decisions on memory placement, there is value to providing automated tools that relieve them of this burden.

Memory placement decisions can also be made at the operating system level. Previous work has shown that relatively simple placement policies that choose where each individual page should be allocated can have a substantial effect on performance [39]. Such policies are known as *static* placement policies, since pages do not move once they are allocated in a specific location until they are paged out. In the past, many multiprocessor systems have used a static placement policy that allocates pages to memory nodes on a *round-robin* basis [31, 35, 57]. The round-robin policy has the advantage of extreme simplicity, and was designed in part to alleviate the problem of memory hot spotting; spreading pages throughout the system theoretically spreads out the requests to memory so that no single memory module will receive a disproportionate number of accesses. However, this policy can have a detrimental effect on performance with respect to the locality of pages. Since a process will have its pages evenly scattered throughout the system, it will often be the case that most of the pages it accesses will be held in memory that is remote to the processor it is running on.

The most popular static placement alternative to the round-robin policy is to base page placement on the first processor to access a page. This policy is known as *first-touch* placement, meaning that the first processor to touch a page allocates it in its local memory. First-touch placement is the default placement used in commercially-available multiprocessor operating systems like IRIX [44], and has been used in multiprocessor ports of Linux [8]. The basis for this policy is the tendency in many parallel scientific applications for the first processor referencing a page to generate most or all of the subsequent references to that page. A study comparing first-touch placement to round-robin placement for a set of parallel applications with statically scheduled threads and memory data sets that did not exceed physical memory size found that first-touch placement could improve performance over round-robin placement by as much as 40% [39].[4]

A major drawback to first-touch placement is that in many parallel applications, a single thread allocates and initializes all data structures for the entire application before each of the parallel threads starts working on their own portion of the data. In these applications, first-touch placement can be fooled into allocating all pages to the

[4]This study also found that in this environment, the addition of dynamic placement techniques such as migration and replication, which will be discussed in Section 2.2.2, had no positive impact on performance.

local memory of the processor running this initialization thread, causing the data to be remotely located for all threads running on other processors. First-touch placement must be able to differentiate between this initialization phase and the later computation phases (this is typically accomplished by user-added directives that signal when the initialization phase is over [39]), as well as migrate pages from the initializing processor to the other processors, to effectively distribute data in the system.

2.2.2 Page Migration and Replication

Page placement techniques such as those described in Section 2.2.1 make placement decisions when pages are first referenced, at which point they are allocated a physical memory frame in a particular memory module. Because these decisions are made before the actual reference patterns to these pages are known, these placement techniques can be thought of as predictive; i.e., the placement decision for a page is based on the predicted reference behaviour for that page. For example, first-touch placement predicts that the processor that issues the first reference to a page will be the processor that issues the majority of references to that page over the application's lifetime.

In contrast to these predictive techniques, page migration is a reactive technique, altering the placement of pages in the system in response to observations of the memory reference traffic. During the execution of an application, a page migration system will monitor the references made to each page in memory by each processor (typically by means of hardware monitoring tools). If the migration system determines that a given page is being frequently accessed by a particular processor, this page can be migrated to that processor's local memory with the expectation that subsequent accesses by that processor will incur local access latencies rather than remote access latencies. Similarly, pages being frequently accessed by multiple processors can possibly be replicated so that each sharing processor has a local copy. Some of the challenges involved in implementing a migration scheme of this type include minimizing the overhead of memory access monitoring, accurately assessing the net gain of migrating a page, and preventing pages from being constantly passed back and forth between processors that are write-sharing those pages.

Much of the early work in the area of page migration was based on software implementations of shared memory in loosely coupled distributed systems. Software shared memory systems such as Treadmarks [1] and IVY [36] migrate a page whenever a remotely located page or object is referenced, effectively using page migration to enforce the shared memory abstraction. Later work done on the BBN Butterfly [42] and the IBM ACE [4, 14, 25, 32] multiprocessors focused on page migration as a potential method for reducing the impact of remote memory latencies in non-cache coherent NUMA multiprocessor systems. Finally, page migration was incorporated into the Stanford DASH and FLASH multiprocessors[10, 35, 55, 56], extending migration research into the CC-NUMA domain. The following sections summarize this later research.

Identifying Candidate Pages for Migration

To implement a successful page migration system, it is necessary to have information regarding the memory accesses generated by each processor for each page in the system. Ideally, this information would take the form of a priori knowledge of all memory accesses that occur over the lifetime of an application.[5] Given such knowledge, we could determine the best initial placement of each page, as well as determine if and when each page should be migrated such that the number of remote accesses made is minimized.[6]

Unfortunately, obtaining this kind of a priori knowledge is impractical, likely requiring a preprocessing run with the same input data to gather information. Instead, almost all page migration research thus far has relied on heuristics to dynamically identify pages

[5]This assumes that processes do not migrate.

[6]This relies on the simplifying assumption that load and store cache misses have equal latencies (an assumption also made by the migration studies referenced in this section). In general, a write miss to a location on a shared page may cost significantly more than a read miss due to the need for coherence actions, such as invalidates or updates, to occur.

that are likely to incur remote memory accesses in the future. These heuristics typically assume that a page that has been frequently accessed in the recent past will continue to be frequently accessed in the near future. These pages, called *hot pages*, are candidates for migration to the processing node making the majority of these accesses.

Gathering these types of statistics can introduce significant overhead in the absence of special monitoring hardware, as the operating system would likely have to unmap a page and service an in core page fault to gather a single data point. These costly operations would have to be repeated several times to gather the multiple data points required to reliably identify hot pages. As described below, some experimental systems (and at least one commercial system, the SGI Origin series [28]), have introduced hardware-based monitoring to reduce these costs.

A migration system must also consider the overhead of migrating or replicating a page in comparison to the expected benefits of increased locality. This overhead includes not only the cost of physically copying a page from one location to the next, but such costs as the associated kernel overhead required for allocating a new page, changing page mappings, and flushing TLBs to maintain coherence. Such consideration typically involves adjusting the threshold of how frequently a page must be accessed before it is considered a hot page.

Given a page that is being accessed frequently enough to exceed this threshold, the decision as to whether to migrate or replicate depends on the pattern in which it is being accessed, and whether there is room at the target node or nodes to accommodate such action. Access patterns to a page can be broadly classified into three categories: (1) those that are mainly accessed by a single processor,[7] (2) those that are accessed by multiple processors in a mostly read-only fashion, and (3) those that are accessed by multiple processors in a mostly read-write fashion. Pages exhibiting the first type of sharing are

[7]This may include pages with private data owned by a single-threaded process, or pages that receive a block of accesses from a single processor, then a block of accesses from another processor etc.

the best candidates for migration.

A page that is accessed by multiple processors in a mostly read-only manner is not a good candidate for migration unless a majority of the accessing processors share the same local memory. If this is not the case, then no matter which node the page is migrated to, a majority of the processors will be accessing it remotely. Continuously migrating the page between the local memory of each accessing processor is not a viable solution because the overhead of each migration would outweigh the benefits gained. In this case, replication of the page is a better solution, as it allows each sharing processor to have its own local copy.

Read-write pages that are accessed by multiple processors at the same time, and where the reads and writes are mostly interleaved, are not good candidates for either migration or replication. Replication is not viable because of the write accesses, which incur the overhead of eliminating replicated copies to maintain consistency. Migration provides no benefits for the same reason as in the read-only case: there is no single location to migrate the page to that will eliminate remote accesses. Hence, remote memory accesses cannot be easily avoided using migration or replication for pages that are actively read-write shared.

Migration using Hardware-based Monitoring

Early page migration systems such as those developed for the BBN Butterfly [5, 31] and the IBM ACE [4, 14, 25, 32] were tied to the page fault mechanism, with the frequency of in-core page faults to a page being used to determine whether or not a page should be classified as a hot page. One key difference between these early multiprocessor systems and the CC-NUMA environments that are prevalent today is that the former systems either did not employ processor caches or did not provide hardware to maintain cache coherence.[8] This is an important distinction in CC-NUMA multiprocessors since the intention of a page migration system is to alleviate the relatively high cost of remote memory latencies. If a page is heavily accessed, but most or all of these accesses hit in the local processor cache, then its placement in local or remote memory will have little impact on performance. [9] For this reason, migration using software-based techniques, such as counting in-core page faults, to identify hot pages in CC-NUMA systems typically do not improve performance [55].

Work done on the Stanford DASH and FLASH multiprocessors examined the utility of page migration and replication based on hardware monitoring in a modern CC-NUMA multiprocessor [55, 56]. The Stanford approach used hardware monitoring of cache misses to identify hot pages, initially relying on the counting of all cache misses by all processors for each page in memory. This full monitoring approach allowed exact identification of those pages that incurred the most remote memory traffic. The results of their studies show that the use of migration and replication could result in up to a 30% improvement in execution time over a static first-touch placement for some multiprogrammed workloads consisting of both sequential and parallel programs using UNIX priority scheduling. This improvement was based on up to a 50% savings in memory stall time, i.e., time spent waiting for loads and stores to return data. Additional experiments showed that full cache miss information was not necessary to achieve these performance gains. They found instead that sampling as little as 10% of all cache misses could reduce the hardware overhead with no appreciable impact on the identification of hot pages.

[8]In fact, migration on these latter systems was considered a substitute for providing hardware-based cache coherence [55].

[9]The exception to this is a write access to a valid shared cache line, which may require coherence actions. Such write accesses will typically require a message to be sent to the memory node where the home page is located, as well as possibly other coherence messages sent from that node to other nodes. As such, the latency of write accesses to valid shared lines can depend on the placement of the backing memory page.

Migration using User Level Information

The majority of research into page migration has focused on the implementation of a migration framework at the operating system level. The advantage to a system level implementation is that it will have better access to lower level information such as page fault data. It also allows easy access to hardware monitoring subsystems, if available, that can inform the page migration policy.

However, making migration decisions at the system level does not include some of the higher level context for which these decisions are being made. Recently, it has been shown that making use of user level information can result in performance improvements of over 200% for some workloads [40].[10] In particular, they show that a page migration system can take advantage of iterative control flow structures that are easily identifiable at the user level to drive migration decisions. By taking page reference count snapshots using hardware monitoring tools at the ends of loop iterations, a migration system can develop a memory reference profile for each loop structure in an application by extrapolating the memory reference behaviour of early iterations to the entire lifetime of each loop. The decision to migrate or replicate pages can then be made based on this extrapolated behaviour. Migration decisions informed in this manner become more timely; rather than wait for a threshold number of references to trigger a migration decision, user level migration takes advantage of the iterative nature of parallel programs (and their subsequent predictability with regards to memory reference patterns) to identify hot pages earlier.

In general, while dynamic migration techniques correctly focus on the volume of cache misses to inform placement decisions, they do so by relying on hardware monitoring to gather this information. Our own work also focuses on identifying pages that suffer

[10]The experimental environment for these studies included extremely frequent process migration, making it somewhat difficult to compare these large performance gains to the more modest gains reported by the hardware monitoring migration policy discussed in the previous section.

high numbers of cache misses, but avoids the need for specialized hardware by using information that is easily gathered through the operating system.

2.3 Architectural Approaches to Reducing Remote Memory Latencies

Another way of dealing with the effects of remote memory latencies in CC-NUMA multiprocessors is to introduce architectural features to improve data locality. Two such approaches to eliminating remote memory latencies are *network caches*, and the *Cache Only Memory Architecture*.

2.3.1 Network Caches

A *network cache*[11] is a large cache that is shared by the processors of a processing node [38, 41]. This cache, often implemented with DRAM, is used to cache data that has been allocated in remote memory and accessed by a local processor. Each processor cache miss by a local processor whose address indicates a location in a remote memory is first sent to the network cache. If the requested data is present in the network cache, it is provided to the secondary cache, alleviating the need for a remote memory request. Otherwise, the memory request is sent to the proper remote node indicated by the address of the request. When the data is returned, the network cache stores a copy and passes the fetched data to the processor.

The benefit of adding network caches is in the reduction of the number of remote memory accesses being made. This is accomplished by the migration of a processor's working set to the local network cache. This effect is most evident when the working set is too large to fit in the processor's cache hierarchy. Although there is no general

[11]Sometimes referred to as a *remote cache*.

consensus regarding the usefulness of network caches, there have been some studies that have reported sizable performance gains in some applications. For example, some studies have found the addition of network caches produce an average of 20% improvement in the execution time of the SPLASH-2 benchmarks on some CC-NUMA multiprocessors [22, 60].

The cost of adding network caches to a CC-NUMA system can be measured in the extra hardware and complexity they require, as well as an increase in the average latency to satisfy a remote memory request that is not satisfied by the network cache. The hardware overhead consists of the memory each network cache is composed of, which might otherwise have been added to the size of each processing node's local memory. Additional hardware is also required to integrate the network caches into the cache coherence protocol of the overall system. The average remote memory latency on a network cache miss is increased (typically on the order of 10% [22, 60]) due to the added cost of the failed network cache lookup.

2.3.2 Cache Only Memory Architecture

We have previously stated that one approach to reducing or eliminating the impact of remote memory latencies in NUMA multiprocessors is to migrate pages from remote locations to the local memories of the processors that are accessing them. In Section 2.2.2 we explained how this could be done in the operating system, with appropriate assistance from hardware monitoring systems. In contrast to this, the *Cache Only Memory Architecture* (COMA), is a migration and replication system implemented entirely in hardware.

COMA multiprocessors [16, 30, 43, 49, 53] have the same basic physical structure as more traditional MP or CC-NUMA systems, where several processing nodes are connected together by a dedicated high speed network. However, COMA systems differ from these more traditional multiprocessor architectures in the way that their memories are organized. In a COMA system, each memory module is organized as a cache, called an *attraction memory*, consisting of a collection of data blocks (typically smaller than a page) and associated tags. Cache misses are initially sent to the local attraction memory. If the address requested matches one of the tags in the attraction memory, then the request can be satisfied as a local memory transaction. However, if the local attraction memory does not hold the requested block, then a remote request is required. Once the remote block is found, it is copied into the local attraction memory in addition to satisfying the processor's request.[12] In this way, the processor's working set is migrated to the local attraction memory, reducing or eliminating the need for further remote requests until the working set changes.

The organization of memories as caches implies certain consequences that are unique to COMA systems. One of these consequences is the need for an infrastructure that organizes block location information in the system so that remote data can be located. This need arises due to the fact that, unlike non-COMA shared memory multiprocessors, the address of a request does not specify a home memory location in the system, but is rather a unique identifier for data in the system. Without some kind of directory structure, the only way to find a remotely held block would be to query the tags of every attraction memory in the system.

Early COMA multiprocessors such as the Kendell Square Research KSR-1 [26, 33] or the Swedish Institute of Computer Science Data Diffusion Machine [24] used a hierarchically organized directory to track all blocks in the system. In this type of directory structure, each processor is connected to the leaf nodes of a directory tree. Each node in the directory tree holds block information for all nodes in the subtree for which it is the root. The location of any block can be found by sending a request up the directory tree until the root of the subtree containing the requested block is reached. At this point, the request can be sent down the tree until the specific leaf node containing the block is

[12]Because the attraction memory is a cache, this can result in the ejection of a valid block.

found.

Some subsequent COMA multiprocessors have typically employed an organization known as Flat-COMA [29, 49]. In this type of directory structure, each attraction memory also includes a directory containing location information for blocks in the system. Each block is assigned a unique home location in one of these directories, whose contents are a pointer to the attraction memory that holds the block. In this type of architecture, remote requests first go to the home location in the correct directory, and then are redirected to the proper attraction memory. Such redirection is not necessary if the attraction memory that holds the directory listing also holds the requested block.

Another consequence of organizing memories as caches is that there must exist a mechanism to prevent a block from being lost because it has been overwritten in all the attraction memories that it is held in. In a typical cache hierarchy, ejecting a block from a cache does not cause that data to be lost because it is also stored in main memory. However, in a COMA system, since the attraction memories are themselves caches, there is no master location that prevents an ejected block from being lost. One solution is to simply designate one of the copies of each block as the master copy in the system. The master copy is never allowed to be overwritten should the need arise to eject it from an attraction memory. Instead, if a master copy is to be ejected, it must be migrated to another attraction memory.

2.3.3 CC-NUMA with Network Caches vs. COMA

Both network caches and COMA attempt to reduce the number of remote memory accesses in multiprocessor systems through hardware migration of the working set of a processor to its local memory hierarchy. Despite this commonality, both are very disparate solutions. Although the network cache approach appears to be more prevalent in current systems (in fact, to the best of our knowledge, there are no currently active COMA projects in academia, and no commercially available COMA systems), there have been COMA proponents who have claimed that their approach can be superior to the network cache solution [60].

One of the major advantages of a CC-NUMA multiprocessor using network caches over the COMA approach is that the former is a simpler design. Additionally, a CC-NUMA multiprocessor will typically exhibit lower memory latencies than a comparable COMA system. This is because of the way each type of system resolves certain remote memory requests. A cache miss to a remote address in a CC-NUMA system that is not due to coherence effects will be satisfied in two network hops: a request to the home location of the address, and the reply. The same cache miss in a COMA processor may take either two or three network hops. The extra hop is required when the master copy of the block being accessed has been displaced from its home attraction memory, and the original request must be redirected to the new location of the master copy.

The advantage that a COMA multiprocessor has over an equivalent CC-NUMA multiprocessor using network caches[13] is that the COMA design should have greater success at reducing the number of remote memory requests for applications whose working set is larger than the network cache size of the equivalent CC-NUMA system. Since a COMA system has up to the full size of an attraction memory with which to fit migrated blocks, it can better accommodate larger working sets than a CC-NUMA system with its relatively smaller network cache. In practice, this advantage is only a factor for shared pages that are accessed by a single processor at a time (as in traditional page migration systems, shared pages whose accesses are interleaved among several processors do not benefit from migration).

The few direct comparisons between the two designs appear to show that the CC-NUMA with network cache approach outperforms COMA by a small amount (around 5%)

[13]Two systems are considered equivalent if: (i) they share the same network topology and number of processing nodes, (ii) processing nodes in each system have the same number and type of processors, and (iii) the size of each attraction memory in the COMA system is equal to the size of a local memory node plus the network cache size in the CC-NUMA system.

[46, 60], while also being a much simpler design. Some COMA proponents have argued that this slight advantage is due in large part to the choice of benchmarks studied, i.e., SPLASH-2, which they claim to be heavily optimized for CC-NUMA systems. To date, there have been no comprehensive studies that have compared each design with more general purpose applications.

2.4 Page Replacement

An area of memory management that bears some similarities to the problem of locality optimizations for page placement in multiprocessors is page replacement. The need for page replacement mechanisms arises due to the finite amount of physical memory in any system, and the discrepancy in size between virtual and physical memory. Since physical memory is typically much smaller than the size of the virtual memory address space, the memory management subsystem transfers pages from secondary storage to physical memory as pages are accessed. This type of memory management is often referred to as *demand paging* [45]. In demand paging systems, an access to a page that does not currently reside in physical memory causes that page to be *paged in* to a physical memory frame. The page replacement mechanism is invoked when a page must be paged in and there are no free memory frames available.[14]

Page replacement algorithms in a demand paging system can be distinguished by how they choose pages to evict from physical memory. The choice must be made when a page that does not currently reside in physical memory is accessed, i.e., it must be *paged in*, and there are no free memory frames available. The choice of which page to evict can have a substantial impact on performance, because the cost of faulting in a page from secondary storage to main memory is orders of magnitude higher than the cost of accessing a page already in memory. If we evict a page that will be accessed again in the

[14]Some policies may use other thresholds of memory usage to invoke replacement before all physical memory is exhausted.

near future, we will be forced to page it back in, incurring this expensive overhead.

The dichotomy between low and high latency storage makes the page replacement problem similar to the NUMA locality problem we have discussed above. In both cases, we wish to maximize the number of cache misses that are satisfied in low latency storage, while minimizing those accesses that must go to high latency storage. In the case of memory locality in multiprocessors, the division between low and high latency occurs between local and remote memory respectively; in page replacement, between physical memory and secondary storage. Of course, these two situations are not completely analogous. In particular, the cost differential between local and remote memory is much lower than that of memory and disk. In the former case, it generally costs between 2 and 10 times more to access remote memory as it does local memory; in the latter case, it costs several orders of magnitude more to access the disk than to access memory. Nevertheless, there are concepts that have been introduced in the area of page replacement that are relevant to NUMA locality and page placement.

2.4.1 The Second Chance Algorithm

In theory, one of the simplest and best heuristics for choosing a victim page for replacement is *least recently used*, or LRU [45]. This heuristic selects the page in memory whose last access occurred furthest in the past. In practice, LRU is difficult to implement, and many different heuristics have been proposed to mimic LRU behaviour. One of the most popular approximations of the LRU replacement algorithm is known as the *second chance* algorithm [2]. Conceptually, this algorithm begins by building a list of page descriptors corresponding to each page in memory. This list is ordered as a queue, with the descriptor for the most recently allocated page located at the end of the list. Each descriptor includes a *reference bit*, which is set when the corresponding page is accessed, and can be implemented in hardware, or simulated by software.

Since the order in which descriptors are stored in the list is the order in which their corresponding pages have been allocated, the oldest pages are those at the head of the list. When it is necessary to choose a victim page for replacement, the algorithm begins inspecting candidate pages starting with the head of the page descriptor list. If the candidate page's reference bit has been set (indicating it has been accessed since the last time it was inspected), the page is given a second chance: the reference bit is cleared, the descriptor is moved to the end of the list, and the algorithm moves on to the next page in the list. Otherwise, the page is selected for replacement. In this way, pages that have been recently used are allowed to stay in memory, and a page that is accessed with sufficient frequency will never be victimized.

A variation on this algorithm is to introduce a reference counter composed of multiple bits for each page descriptor, and a clock that initiates regular sweeps of the descriptor list. An access to a page sets the leftmost bit in its reference counter, while each sweep of the list *ages* the reference counter (e.g., right shifting the bits). If a page is not accessed in the time between n clock sweeps (where n depends on the type of aging and the size of the reference counter), the value of the reference counter will be zero. During each sweep, all descriptors that have a zero counter are placed on the *free list*, which contains descriptors of pages that can be used to satisfy page faults.

2.4.2 Adaptive Page Replacement Based on Memory Reference Behaviour

Although LRU-type heuristics generally show good performance on a wide range of applications, some applications that exhibit certain memory reference patterns perform very poorly under LRU-type replacement relative to the optimal offline replacement algorithm. For example, a commonly occurring memory reference pattern that is not handled well by LRU-type replacement algorithms is a streaming pattern that repeatedly loops through a group of N pages in the same order, 0 to $N-1$, with N greater than M, the number of physical memory frames. In this streaming pattern, each page will be re-accessed every

N page accesses. Under LRU-type replacement, since the number of physical memory frames is less than N, a page will become the least recently used page in memory after M page accesses. Since $M < N$, a page will be chosen for eviction under LRU-type replacement before it is re-accessed.

The unsuitability of LRU-type replacement for streaming access patterns is further highlighted by the fact that increasing the number of free physical pages M does not improve performance as long as $M < N$. For most applications that do not exhibit streaming access patterns, gradually increasing the number of free physical pages results in a corresponding gradual reduction in page replacement activity under LRU-type replacement [19]. However, for the streaming access pattern that we have described above, increasing M has no effect on the number of page replacements until $M = N$, at which point there is enough physical memory to hold every page until it is re-accessed.

In applications that exhibit streaming access behaviour, using LRU replacement can result in as many as 5 to 10 times more page faults than the optimal offline algorithm [19]. One solution to this problem is to adapt the type of page replacement being used to fit the memory reference patterns being observed. For example, the SEQ replacement algorithm [19] applies LRU replacement until it detects long patterns of sequentially addressed page faults, at which point it switches to an approximation of most recently used (MRU) replacement. The choice of MRU replacement is appropriate for this type of access pattern since the most recently used page in memory is the one that will be re-accessed farthest in the future; i.e., the same criteria for choosing a victim in Belady's optimal offline algorithm [45]. Under SEQ replacement, many applications that perform poorly under LRU-type replacement have been shown to achieve near optimal replacement performance.

Another approach to the problem of streaming access patterns is *Early Eviction LRU*, or EELRU [47]. Rather than considering the addresses of page faults to detect streaming access patterns, EELRU collects information on how many other pages have been touched since a page has been last accessed, called *recency information.* EELRU switches from LRU-type replacement to an early eviction replacement scheme (choosing the e-th most recently used page rather than the least recently used page for eviction)[15] when it detects that recently evicted pages are being re-fetched into memory. The advantage to relying on recency information lies in the fact that some streaming access patterns can be detected by EELRU that are ignored by the SEQ algorithm, e.g., looping through a dynamically allocated linked list.

Adaptive page replacement has influenced our work by suggesting that observing the page fault ordering of an application can provide useful data that can inform resource management policies. As we will show in Chapter 4, we use a similar distinction between sequential and non-sequential fault patterns to inform page placement decisions.

2.5 Summary

Of the many classes of multiprocessor architectures, a popular type of architecture is the CC-NUMA shared memory architecture. One of the prominent characteristics of this type of multiprocessor architecture is that it exhibits non-uniform memory access times, which arise due to the physical distribution of memory modules or nodes throughout the system. This distribution generally leads to a division within the memory hierarchy, with the memory nodes closest to a processor making up local memory, and the more distant nodes making up remote memory. The dichotomy between local and remote memory and the non-uniform access times that arise can make the location of pages in memory an important consideration for application performance.

There have been a number of projects that have addressed this issue. They include software solutions, such as page migration, and architectural solutions, such as the Cache

[15]Early eviction is similar to, though less aggressive than, MRU replacement (both attempt to evict more recently used pages over less recently used ones). By choosing the e-th most recently used page, early eviction responds better to phase changes in the working set of an application (which can cause MRU to carry pages that will never be used again indefinitely).

Only Memory Architecture (COMA) and network caches. Most of these approaches consist of attempts to dynamically relocate heavily referenced memory units (e.g., cache blocks or memory pages) closer to the referencing processor.

Another area of research that bears similarity to the optimization of memory locality is the issue of how to implement page replacement. Page replacement algorithms also deal with the optimization of memory usage between two levels of hierarchy that have significantly different latencies: main memory and secondary storage. Although the latency difference between these two levels of the memory hierarchy is much greater than the typical difference in latency between local and remote memory, page replacement algorithms employ some concepts that are relevant to the discussion of memory locality and page placement.

Chapter 3

Page Placement in CC-NUMA Multiprocessors

In recent years, there has been a significant change in the way that large multiprocessors with distributed memory subsystems are used. Previously, these multiprocessor systems were often seen as highly specialized compute platforms running specially written multithreaded applications. These parallel applications would typically run in isolation, without having to compete with other applications for the system's resources. Additionally, these applications often explicitly specified where and how their data structures were to be placed in memory (such as when using HP-Fortran or Fortran D [37]). In some cases, simplistic automatic page placement policies were applied, such as round-robin or first-touch placement [39], and were successful at achieving good performance for highly parallelizable and well behaved applications whose threads did not use more memory than was available at each processing node.

More recently, there has been an increasing tendency to use large multiprocessors as centrally managed, general purpose compute servers with multiple applications running concurrently. In the extreme, these compute servers run UNIX workloads involving many independent processes running in parallel, with the vast majority of the applications being single threaded. These applications do not specify where and how their data should be distributed, and in fact are unaware that they are running on a system with physically distributed memory, having been designed and written for uniprocessor systems. This lack of awareness of the physical distribution of memory can lead to situations where there is not enough local memory to satisfy all of the processes running on a processing node. Under such conditions, simplistic memory management policies such as first-touch placement can lead to poor performance.

In this chapter, we discuss how a traditional static page placement policy like first-touch placement can result in poor memory locality for multiprogrammed workloads, leading to poor performance. First-touch placement has been used in the popular commercially-available IRIX multiprocessor operating system [44], as well as in multiprocessor ports of uniprocessor operating systems like Linux [8]. As such, it is the standard against which most page placement techniques are measured. We begin by describing the limitations of first-touch placement, and demonstrate how its use on a single threaded application whose data does not fit in local memory can lead to performance that is up to 30% slower than a best case allocation policy that has a priori knowledge of all memory accesses. Although workloads consisting of one single threaded application are not common in multiprocessor systems, we go on to show how the shortcomings of first-touch placement for a single application directly translate to the more common multiprogrammed scenario where several processes running on a node cannot fit entirely into the local memory of that node.[1] Finally, we show that a placement policy that knows whether a group of pages will be heavily accessed or lightly accessed can use this information to place pages so that the number of remote memory accesses will be reduced compared to first-touch placement.

[1]The case of a multiprogrammed workload where each process is running on a separate node is similar to the single program workload case.

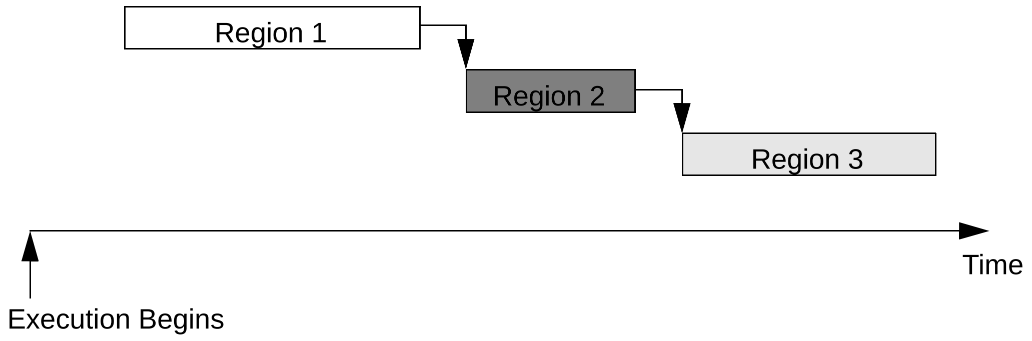
3.1 Limitations of First-Touch Placement for Single threaded Applications

One of the goals of a static page placement policy, such as first-touch placement, is to locate data in memory so that the majority of accesses to this data are made from a processor on the same node as the memory module containing the data. First-touch placement attempts to achieve this goal by relying on the common observation that for a multithreaded application, the thread that first accesses a memory page is also the one that tends to access it the most in the future. Based on this observation, first-touch placement allocates each page in the local memory of the processor running the thread that first accessed it.

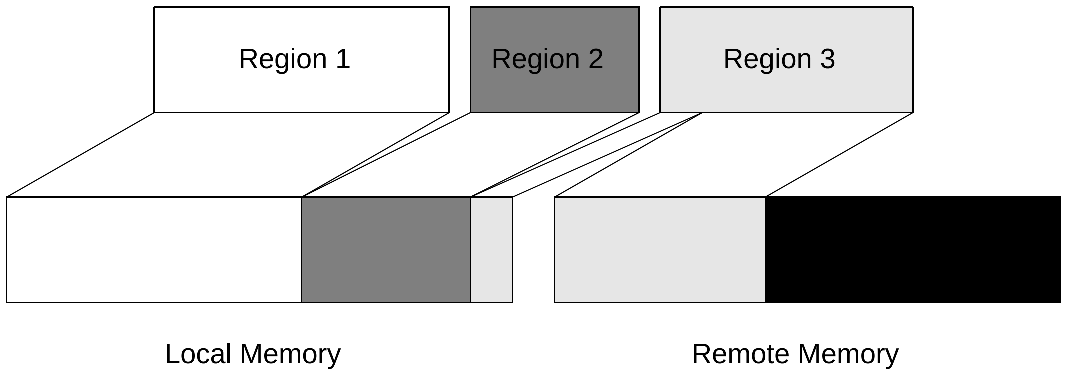
First-touch placement has been found to be effective at improving the performance of well behaved multithreaded applications where the combined memory needs of the threads on each processing node do not exceed the amount of memory available at each node [39]. However, as we will show in this section, first-touch can be a poor choice for applications or workloads where the combined data of the threads or processes on a given node do not entirely fit in the local memory of that node. First-touch placement can perform poorly in these situations because the implicit criteria it uses to decide which pages are allocated to local memory frames is first come first served: the pages that are allocated in local memory are those pages that are accessed first during execution. After local memory has been filled, subsequent page faults are satisfied with remote memory frames.[2] Because of this, the performance of an application can be highly dependent on the order in which it first accesses and allocates its memory pages.

Figure 3.1 illustrates this dependence on the ordering of page accesses using a hypothetical single threaded application. In this example, the application allocates more

[2]Typically, the remote node that is the next closest to the accessing processor in terms of network hops is chosen to provide these frames.



(a) Page Fault Ordering



(b) First-touch placement of Region Pages

Figure 3.1: This figure depicts (a) the page fault ordering, and (b) the resulting first-touch page placement of a hypothetical application. Figure (a) shows the pages of region 1 being paged in first, followed by the pages of region 2, and then the pages of region 3. Figure (b) shows the pages of regions 1 and 2 placed in local memory (by virtue of having been accessed first). Part of region 3 is placed in local memory until there are no local frames left, with the remaining pages being placed in remote memory.

pages than can be held in the local memory node, but there is enough aggregate space in the entire system to hold these pages. The initial page fault ordering for this application is given in Figure 3.1(a), where three separate *regions* of memory are being allocated. We use the term *region* to denote a group of contiguous virtual pages where each page in the region has similar memory access characteristics, i.e., they are all accessed approximately the same number of times.[3] In this application, the initial page fault ordering consists of the pages of region 1 being accessed first, followed by the pages of region 2, and finally the pages of region 3. Under first-touch placement, the pages of region 1 and region 2, being the first pages accessed, are allocated in local memory. When the pages of region 3 are accessed, the first few page faults are allocated in local memory until no more local memory frames are available, after which the remaining page faults are satisfied by remote memory frames.

Given an application like the one shown in Figure 3.1, a more efficient use of local memory would be to allocate those pages that will incur the most accesses in local memory, while placing those pages with the least number of accesses in remote memory. However, there is no guarantee that the pages of region 3 that are allocated in remote memory will incur fewer memory requests than the pages of regions 1 and 2. The pages of region 3 are placed remotely only by virtue of being accessed last. Since we cannot assume that the pages that will be accessed the most will be the first ones allocated for all applications, first-touch placement can sometimes cause a substantial number of remote memory accesses that could be avoided.

3.1.1 First-Touch Placement vs. A Priori Placement

To quantitatively illustrate the shortcomings of first-touch placement with respect to single threaded applications, Figure 3.2 shows a comparison of the simulated performance

[3]We will present a more formal definition of the term *region* as it relates to our base operating system, Tornado, as well as a method for identifying regions, in Chapter 4.

of several benchmark programs executing under first-touch placement versus a placement policy that minimizes the number of references to remote memory. This comparison was generated for applications taken from the Spec95fp and Perfect benchmark suites [3, 13], using a MINT-based simulator [22, 54] configured to reproduce the NUMAchine multiprocessor environment [23]. NUMAchine is a CC-NUMA multiprocessor with 16 MIPS R4400 processors divided into stations of 4 processors each. Each station has a 96 MB local memory module, and the remote to local memory latency ratio is approximately 4:1. The application suite, simulator environment, and the NUMAchine architecture, are described in greater detail in Chapter 5 where we present the majority of our other results. As in the example of the previous section, each application is by itself and allocates more memory than can be held in the local node, but not more than can be held in the entire system. Additionally, each application is statically scheduled and no process migration occurs. To minimize the number of remote memory references, we assumed the availability of a priori knowledge regarding all future memory accesses[4] before any pages were allocated in memory. Given a local memory of size M pages, we used this knowledge to place the top M most referenced pages in local memory, with the remaining pages placed in remote memory.

Figure 3.2 shows that a placement policy that minimizes the number of remote memory accesses can reduce execution time over first-touch placement for 5 of these applications. This reduction is possible despite the fact that NUMAchine is a CC-NUMA system with network caches that help reduce remote accesses. In the other 5 applications, no improvement is seen because the order of page allocations results in the most heavily accessed pages already being placed in local memory under first-touch placement. Best case placement is not feasible for a real system, since a priori knowledge of this type

[4]We make no distinction between read and write accesses here. Although the latency for read and write misses can differ in CC-NUMA systems due to the possible need for coherence actions in the latter case, the use of single-threaded applications with no shared pages implies that no coherence actions are necessary on a write miss.

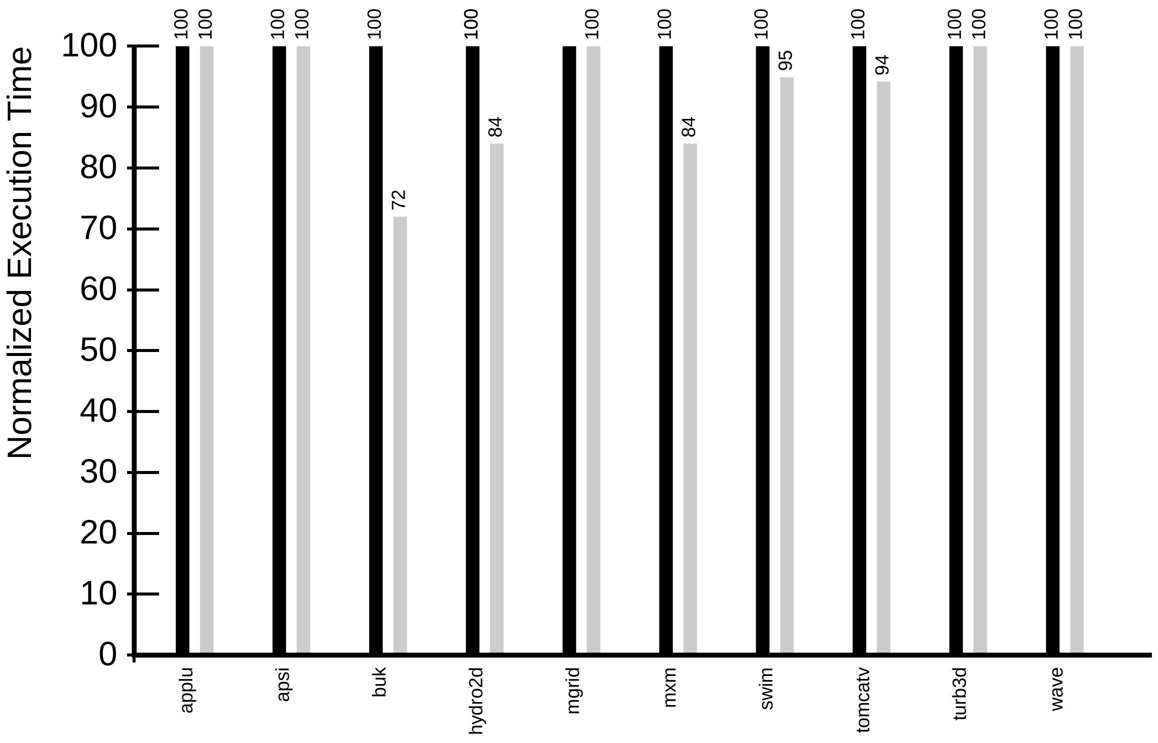


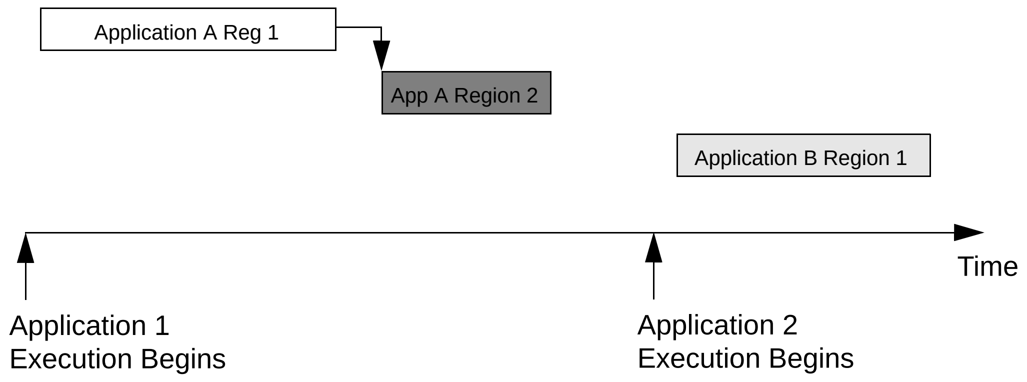
Figure 3.2: A comparison of execution times for each application in our benchmark suite. The gray bar for each application shows the execution time using a best case placement. This time is normalized to the application's execution time using first-touch placement (black bars).

requires a full run of the application with the same input data to generate the reference data, as well as hardware monitoring capable of tracking every memory reference before the knowledge can be applied. However, while not realistic, this policy gives an indication of the possible improvement that can be achieved for these applications running in isolation using a static placement policy in the NUMAchine environment.

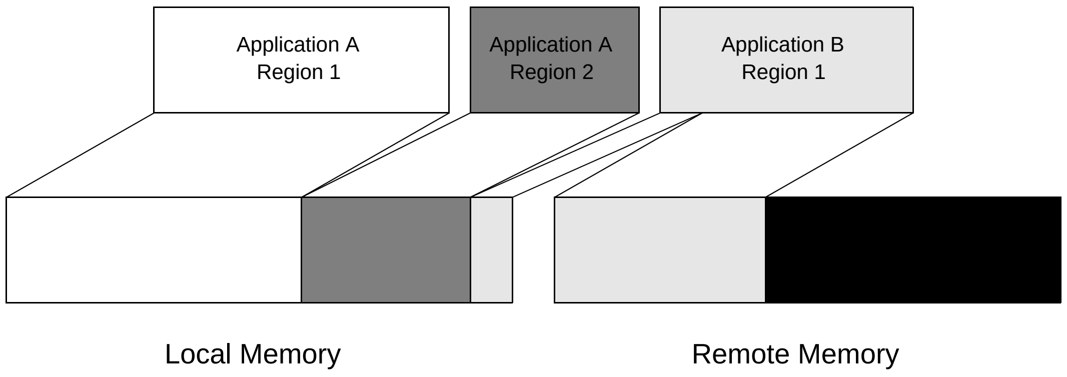
3.2 Limitations of First-Touch Placement for Multiprogrammed Workloads

Although it is not common for multiprocessor workloads to consist of a single application, the shortcomings of first-touch placement outlined above still apply in a multiprogrammed environment. For example, the previous analysis can also be applied to those processing nodes in a multiprogrammed environment where only a single process is being executed i.e., the other processes are executing on different nodes, and none of that node's local memory has been allocated to these other processes. Figure 3.3 gives an example of a hypothetical multiprogrammed workload where two applications are executed on two processors from the same processing node, where the data of both applications do not simultaneously fit in the local memory module. Figure 3.3(a) shows one possible ordering of the execution of these two applications. To make this example clear, we have chosen a very simple ordering, where all of the pages in application A are allocated before all of the pages in application B. As in the single application example shown in Figure 3.1, first-touch placement allocates local memory pages on a first come first served basis, which results in application A receiving the majority of local pages for its two regions $A1$ and $A2$, and application B receiving mostly remote pages for its region $B1$. However, if region $B1$ is accessed more frequently than both $A1$ and $A2$, this may not be the most efficient use of local memory.

One crucial difference between this multiprogrammed case and the single application case is that each of the individual applications in the multiprogrammed case can fit into local memory if they are executed on their own. Running a multiprogrammed workload on a multiprocessor where memory nodes are shared between processors can result in greater competition for local memory than a single application workload. A consequence of this, which we demonstrate later, is that applications that are unaffected by the inefficiencies of first-touch placement in the single program case might still be



(a) Application Execution and Page Fault Ordering



(b) First-touch placement of Region Pages

Figure 3.3: This figure depicts (a) the page fault ordering, and (b) resulting first-touch page placement of a hypothetical two application workload. Figure (a) shows the regions from application A being paged in first, followed by a single region from application B. Figure (b) shows the regions of application A placed in local memory (by virtue of having been accessed first). Part of region 1 from application B is placed in local memory until there is no more left, with the remaining pages being placed in remote memory.

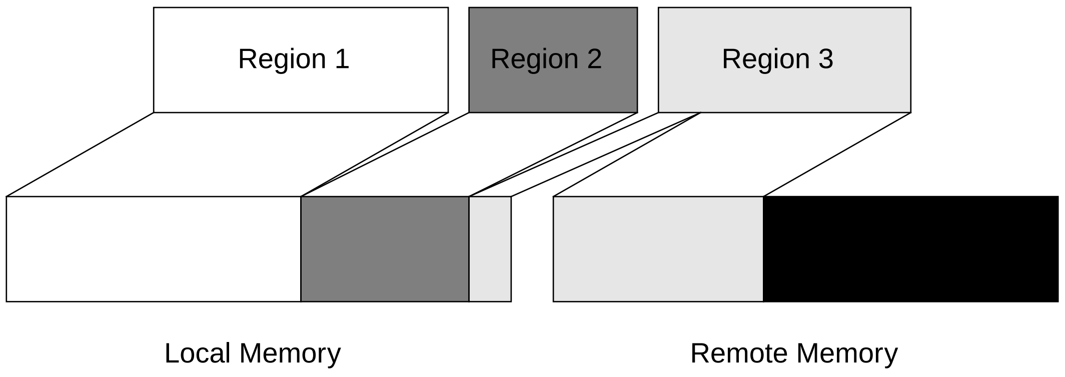
negatively affected in the multiprogrammed case. In either case, a placement system that prioritizes local memory allocation based on how often regions are accessed may be more effective in reducing remote memory accesses than first-touch placement.

3.2.1 Reducing Remote Memory Accesses in Single Program Workloads

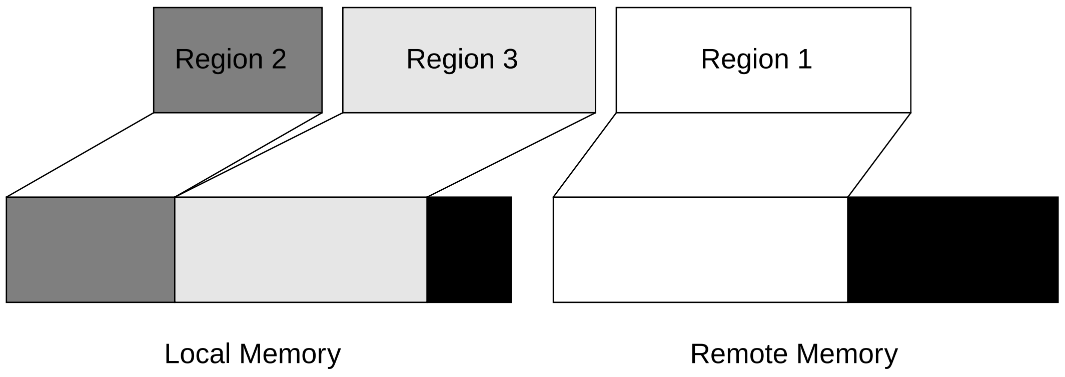
We now revisit the hypothetical application described in Section 3.1 to show how a placement policy that considers how often memory regions are accessed can lead to a better static page placement than a first-touch policy. Recall that this hypothetical application allocates three memory regions in succession whose total size is larger than the size of local memory. Figure 3.4(a) depicts the placement of each region resulting from the application of first-touch placement.

Although region 3 is mostly allocated in remote memory under first-touch, its placement there may be suboptimal depending on the relative numbers of future accesses to each region. For example, if the pages of region 3 are more heavily accessed than those of region 1, then placing region 3 in local memory and region 1 in remote memory will result in better performance. In the comparison between first-touch and a priori placement described in Section 3.1.1, we used a priori knowledge of such differences in memory reference behaviour on a page granularity to achieve a better static placement that minimizes remote memory accesses. Although gathering such a priori knowledge is impractical, one way to approximate such a placement policy is to develop a heuristic that predicts the relative memory access behaviour before pages are allocated. For example, one might imagine a heuristic that is able to divide regions into two categories: lightly accessed, and heavily accessed.[5] Given such a heuristic, we could devise a placement policy where the lightly accessed regions are placed in remote memory to make room for heavily accessed

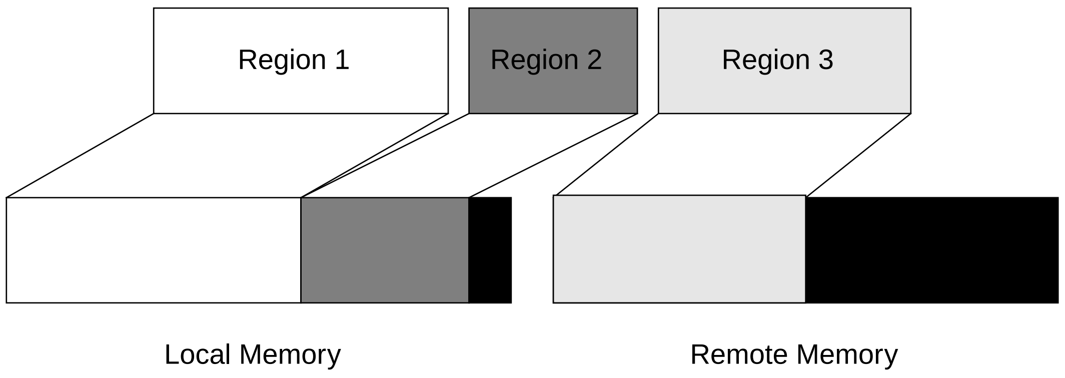
[5]We will introduce such a heuristic of our own devising in Chapter 4.



(a) First-touch placement



(b) Lightly accessed Region 1 placed remotely



(c) Lightly accessed Region 3 placed remotely

Figure 3.4: In this figure, we show several of the possible outcomes for region placement of an application with three regions that are paged in consecutively, i.e., all of *region 1*, is accessed first, then *region 2*, then *region 3*. In (a), we show the region placement given a first-touch placement. In (b), we show the placement resulting when *region 1* is chosen for remote placement, allowing *region 3* to fit entirely in local memory. In (c), *region 3* is chosen for remote placement, causing no changes for *region 1* and *region 2*.

regions in local memory.

Figure 3.4(b) shows the static page placement resulting from such a policy when region 1 is lightly accessed, and regions 2 and 3 are heavily accessed. Even though region 1 is paged in first, it is placed in remote memory, leaving enough room for region 2 and region 3 to fit in local memory when they are paged in. Compared to the first-touch placement shown in (a), region 3 is completely held in local memory, with the expectation that this will improve overall performance by making all accesses to its pages local.

Figure 3.4(c) presents the resulting placement under a different scenario, where region 3 is lightly accessed, and regions 1 and 2 are heavily accessed. In this scenario, regions 1 and 2 are paged into local memory first, and then region 3 is paged entirely into remote memory. Compared to the first-touch placement shown in (a), region 3 is entirely in remote memory, rather than partly in local memory and partly in remote memory. In this case, we can expect no improvement due to this change because we have not reduced the number of remote memory accesses. In fact, we have slightly increased the number of remote memory accesses because all of region 3 is now in remote memory, whereas only some of region 3 was in remote memory under first-touch placement.

Thus, if our heuristic is accurate in its predictions of future memory accesses, we can expect to improve the performance of applications that resemble the type shown in Figure 3.4(b). However, we would expect no improvement when our proposed policy is applied to applications resembling the type shown in Figures 3.4(c). In fact, it is possible that applications of these types may suffer a slight performance loss due to the small increase in remote memory references. Nevertheless, we will show in the next section that applying our policy to applications of the type shown in Figures 3.4(c) can increase the amount of local memory available to other applications in a multiprogrammed environment, helping to improve their performance.

3.2.2 Reducing Remote Memory Accesses in Multiprogrammed Workloads

The allocation of memory under a multiprogrammed workload such as the one shown in Figure 3.3 is more complicated than that of a single application workload. In this case, when we begin the execution of an application on a node, we have knowledge of what resources are being currently used, but we have no knowledge of what resources may be requested or used in the future over the lifetime of its execution. We also must be careful when using the memory resources of other nodes, as this may affect the performance of applications that are running or will run in the future on those nodes. However, multiprogrammed environments can also provide greater opportunity for performance optimization over a single program environment, with respect to memory allocation and placement, due to the increased contention for local resources that such an environment can create.

In a multiprogrammed environment where two applications are executing on the same processing node with a local memory of size M frames, being within the first M initial page faults for either application no longer guarantees a frame in local memory. This is because some of those M frames may have already been allocated to another application's pages when those faults occur. Instead, a page fault must be within the first M' initial page faults of an application to be allocated a local memory frame, where M' is the number of available local memory frames when the application begins allocating pages (assuming that there is no overlap in allocation periods between both applications), and where $M' < M$.

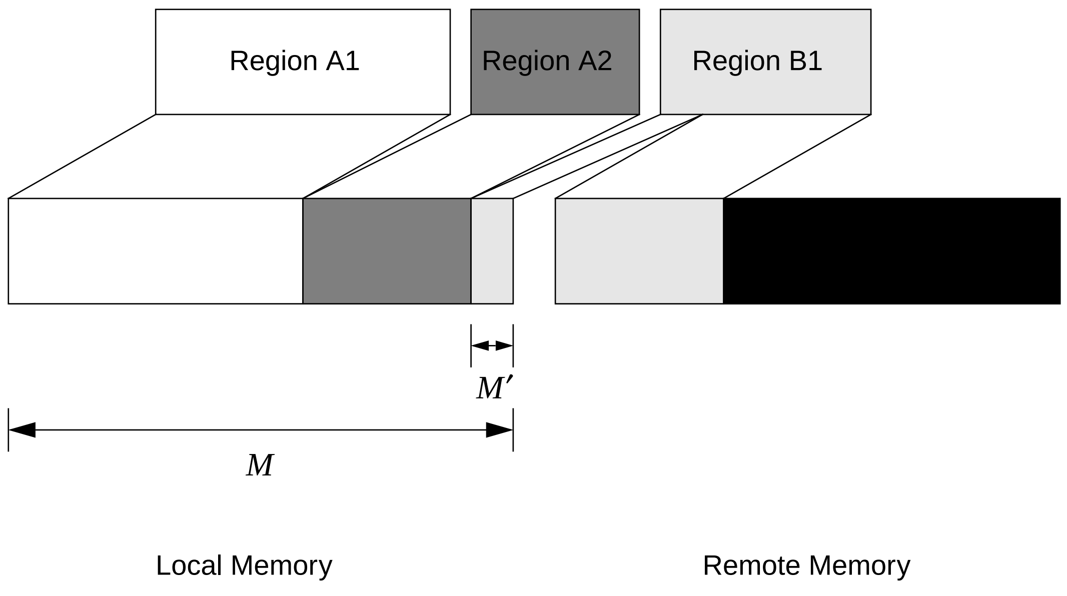
Because of the possibility of increased competition for local memory frames in such an environment, applications that do not exhibit improved performance under single application workloads can still benefit under a multiprogrammed workload where more than one application is running on the same node when our proposed placement policy is applied. This is because moving some regions to remote memory reduces the pressure on local memory and effectively increases M'.

Figure 3.5 illustrates how this might apply to the hypothetical workload introduced in Section 3.2. Application A has two user memory regions, $A1$ and $A2$, while application B has a single user memory region, $B1$. Let us consider a scenario where regions $A1$ and $B1$ are heavily accessed, while region $A2$ is lightly accessed.

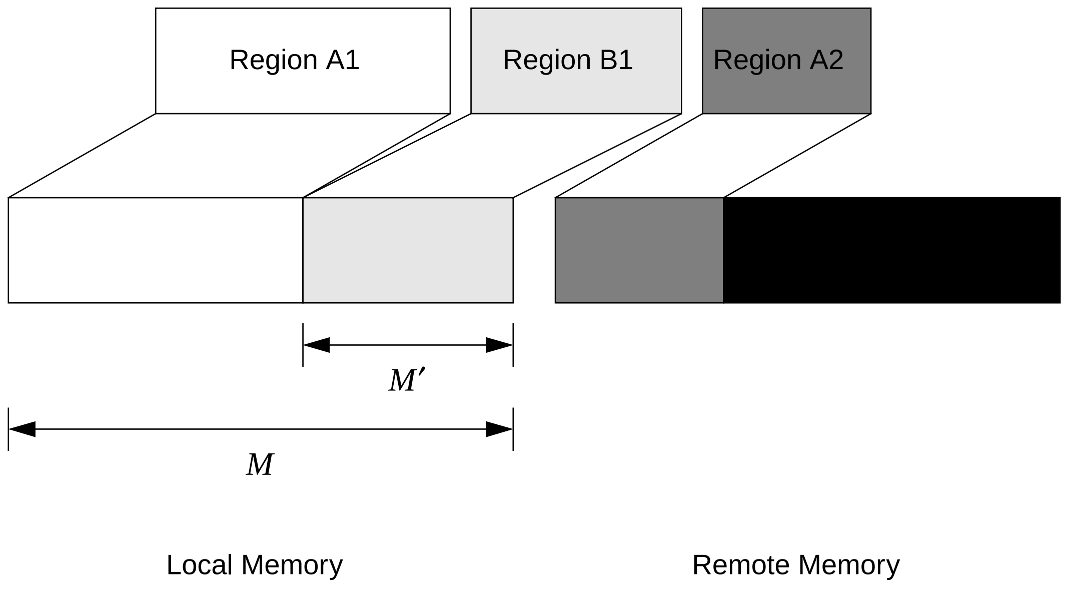
Figure 3.5(a) shows how these regions might be placed in local and remote memory under first-touch placement, assuming that the pages of regions $A1$ and $A2$ are paged into memory first, followed at a later time by the pages of region $B1$. Because of the ordering of the page faults, $A1$ and $A2$ reside completely in local memory. When $B1$ is paged into memory, the remaining amount of local memory, M', is not large enough to hold all of the pages of $B1$, and so most of them are relegated to remote memory. Since the pages of $B1$ are heavily accessed, application B will likely suffer from poor performance due to the large number of remote memory accesses it must make.

Figure 3.5(b) shows how the static page placement would change under our proposed placement policy. Because $A2$ is lightly accessed, it is allocated in remote memory, even though it is paged in before the $B1$. Thus, when $B1$ is paged in, the available local memory M' is large enough to accommodate all of its pages. This results in fewer remote memory accesses for B compared to the first-touch placement shown in Figure 3.5(a). Note that if application B was run in isolation, its static page placement under first-touch placement would be identical to that of the new placement policy; under either policy, the heavily accessed pages of $B1$ would be placed in local memory (since $B1 < M$). In the multiprogrammed case, the increase in competition for local memory frames that forces $B1$ into remote memory provides the opportunity for improvement over first-touch placement.

Note that while basing placement on the identification of lightly accessed and heavily accessed regions in this manner can significantly reduce remote memory accesses over



(a) First-touch placement



(b) Improved placement with Region A2 placed remotely

Figure 3.5: In this figure, we show how the placement of regions in a multiprogrammed environment can be altered by our placement policy. In this scenario, applications A and B allocate three regions, $A1$, $A2$, and $B1$, which are paged in in that order. In (a), we show the region placement given first-touch placement. In (b), we show the placement resulting when Region $A2$ is chosen for remote placement, allowing Region $B1$ to fit entirely in local memory.

first-touch placement, there is a tradeoff in that it can sometimes also increase remote memory accesses. For example, if $B1$ is also identified as being lightly accessed, then it will also be placed in remote memory, and some local memory that could hold pages from either $A2$ or $B1$ will go unused. In this case, the number of remote memory accesses will be even greater than in first-touch (which places $A2$ in local memory). However, if the number of accesses to $A2$ and $B1$ is small enough, then their placement in remote memory should have a minor impact on performance.

The example in Figure 3.5 illustrates the notion that in a multiprogrammed environment, our proposed placement policy may adjust the page placement of an application so that other applications may benefit, even if that application does not benefit directly from the new page placement. In the above example, application A does not benefit from the placement of region $A2$ in remote memory, because the placement of its heavily accessed region $A1$ is not affected. In fact, as we noted previously, we increase the number of remote memory accesses incurred by application A by placing $A2$ in remote memory. However, application B derives significant benefit from the remote placement of $A2$, and if the number of accesses to $A2$ is sufficiently low, the impact on A should be minor. It should also be noted that although we say that we increase the number of remote memory accesses for A by moving $A2$ out of local memory, this increase is compared to a best case scenario for A under first-touch placement, where A is allowed to page all of its memory into the local node. The average first-touch placement for A may be significantly worse, depending on the characteristics of the environment. For example, it may be the case that A often begins execution on a node where there is not enough free memory for all of its pages.

3.3 Summary

For a thread or process that uses more memory than is available in the local processing node, first-touch placement leads to first come first served allocation of local memory frames to its page faults. This can lead to poor performance if the pages that are accessed the most are faulted in last, causing them to be placed in remote memory. For workloads that include several applications, the demands on local memory can be even greater than single application workloads, making it even more likely that a memory node will not be able to satisfy all local allocation requests. Given these considerations, it appears that some type of prioritization of local memory allocation that considers how often memory regions are accessed, rather than simply first come first served as in first-touch placement, may decrease the number of remote memory accesses, leading to lower average memory access latencies and improved performance for an application. The success of such a policy relies on the development of an accurate method for predicting how often different parts of memory will be accessed.

Chapter 4

Cache Aware Page Placement

In the previous chapter, we discussed how the commonly used first-touch placement policy can lead to suboptimal performance when applied to workloads of single threaded applications. We also showed that remote memory accesses could be reduced by basing placement decisions on how often different regions in memory are accessed. In this chapter, we describe a new placement policy called *cache aware placement* that uses this concept to improve application performance.

4.1 Objectives, Constraints, and Assumptions

Our primary goal in developing a new page placement policy is to improve application performance on CC-NUMA multiprocessor systems by making more efficient use of the memory at each node than first-touch placement does. As we discussed in Chapter 3, a weakness of first-touch placement is that it implicitly allocates local memory frames on a first come first served basis, making it possible for heavily accessed pages to be placed in remote memory simply by virtue of being allocated last. A more efficient use of local memory would be to give priority to placing highly accessed pages in the local memory of the accessing processor, while giving lower priority to placing infrequently accessed pages in local memory.

A secondary goal for the design and implementation of this new policy is to avoid reliance on specialized hardware or computationally complex algorithms. Some advanced dynamic placement techniques, such as page migration, already address the shortcomings of first-touch placement by prioritizing the allocation of local memory frames, using the processor cache miss rate for references to a page to decide which pages should be migrated to local memory. However, page migration systems typically require special monitoring tools to gather this cache miss information. Our goal is to avoid additional hardware tools by using information that is easily gathered through existing operating systems mechanisms. Furthermore, we would like to minimize the need for programmer intervention to supplement this information.

Along with these goals, we have made several simplifying assumptions to aid in the design and evaluation of our work. While some of these assumptions may not be warranted in a real multiprocessor system, they have been generally been made to aid in our initial understanding of the issues involved in the locality problem, with the hope that future work can build on this understanding by including some of the complexities that we have omitted.

Although previous work on dynamic placement techniques has relied to varying degrees on the use of process migration by the scheduler to create opportunities for optimization [40, 55], we have developed our policy for an environment where little or no process migration occurs. Such an assumption is likely to be valid for environments similar to the one we are targeting (i.e., a multiprogrammed environment with many independent single threaded processes). In these systems, scheduling is often implemented using a method known as *two-level scheduling* [51]. The top level of a two-level scheduler assigns processes to the run queue of a specific processor (typically the one with the lowest load) at process creation time. The bottom level is specific to each processor, and determines which process from the local run queue should be scheduled. This method of scheduling keeps a process on the same processor for its entire lifetime, allowing it to develop an affinity for that processor (e.g., a process may find some of its cache blocks still loaded in the cache when it is rescheduled on the same processor). For this reason, two-level scheduling is considered a type of scheduling more generally referred to as *affinity scheduling*. Two-level scheduling also has the advantage of avoiding a highly contended global run queue, while still distributing load relatively evenly throughout the system.

Additionally, we have constrained our application domain to single threaded array-based scientific applications. While multi-threaded applications and applications with pointer-based data structures may also prove to be of interest, we have decided to initially focus on single threaded scientific applications as they have characteristics that facilitate their study, (e.g., no need for complex pointer analysis, and no shared data between threads).

Finally, we have assumed that the total amount of memory in our target system is large enough to satisfy the memory requirements of all applications being run at any given time. In other words, we do not explicitly consider the interaction of our policy with the page replacement system in our experiments.

4.2 Region-Based Cache Aware Page Placement

In accordance with the goals we have previously identified, the algorithm we have developed for determining page placement prioritizes the allocation of local memory frames based on predicted cache behaviour. This prioritization can be summed up as follows: regions that are predicted to have a high processor cache miss rate are preferentially placed in local memory over regions that are predicted to have a low processor cache miss rate. The basis for this prioritization stems from the observation that memory accesses only occur on processor cache misses, and so processor cache miss rates should provide a good approximation for which pages will incur the most memory accesses. Because we use predicted cache behaviour to determine page placement, we call our policy *cache aware placement*.

It is important to note that although we use the term *cache aware placement* to describe our policy, we do not directly measure cache miss rates. As we described earlier, one of our goals was to implement this policy using information that is easily gathered within existing operating systems mechanisms, avoiding the need for special hardware monitoring. The term *cache aware placement* refers to placement based on the *inferred* cache behaviour of regions of memory based on the observed page level access patterns within the operating system.

One of the cornerstones of such a placement policy is the construction of an accurate method of predicting the caching behaviour of pages. We believe that a suitable basis for such predictions is the order in which pages are faulted into memory, which gives an approximate indication of how these pages will be accessed during the application's lifetime. Since cache miss rates are often determined by memory access patterns, we hypothesize that the order in which pages are accessed can be correlated with either high or low cache hit rates over the lifetime of the application. Such a correlation is demonstrated in Section 4.3, where we show that a sequential page fault order for a set of virtual address pages is correlated with low cache miss rates for these pages.

Basing cache predictions on page fault ordering implies that the granularity for these predictions is larger than a single page. Making page-by-page predictions based on page fault data is problematic because the first access to a page immediately triggers its allocation and placement, yielding only a single data point with which to infer the cache behaviour of that page. Instead, cache aware placement makes cache behaviour predictions for groups of non-overlapping, contiguous sequences of virtual memory pages called *regions*. By making a region the basic unit for cache behaviour prediction, several data points can be collected before a prediction for the entire region is made. This is accomplished by treating the first n page faults to a region as a data gathering phase, where n

is some predetermined threshold much smaller than the total size of the region. Although these first n pages must be allocated and placed without the benefit of cache prediction information, their ordering forms the basis for predicting the caching behaviour of the remaining pages in the region.

The basic framework for our algorithm to determine page placement can be summarized as follows. User memory is divided into groups of contiguous virtual pages called regions. The first n page faults to each region are recorded, where n is a preset threshold whose determination will be discussed later. If the ordering of these page faults matches an ordering that is correlated with high processor cache hit rates, all future page allocations for that region are provisionally marked for allocation in remote memory, with the choice of remote node for each future page allocation depending on the measured memory usage at each node.[1] Otherwise, all future page allocations for that region are marked for allocation in local memory.

Since the accuracy of cache behaviour prediction is of paramount importance in the success of such an algorithm, we discuss our method of prediction in the following section. Following this, we present our full algorithm in greater detail, and discuss some of the issues and alternatives we considered in its design.

4.3 Predicting Cache Behaviour for User Memory Regions

One of the main contributions of this dissertation is the development of a novel method of predicting future cache behaviour for regions of pages. This prediction method involves a heuristic that correlates the ordering of page faults in a region with that region's future caching behaviour. The basis for this heuristic is the hypothesis that the page fault ordering for a region is determined by the memory access patterns of that region. Since

[1]We will propose several different definitions of memory usage in Section 4.5.3.

memory access patterns also largely determine cache miss rates, it may be possible to correlate certain page fault orderings with the caching behaviour of a region.

In Chapter 3 we described how a dichotomy of *heavily accessed* and *lightly accessed* regions could be used to in a placement policy that reduces remote memory accesses. Since regions with high cache miss rates are more likely to be heavily accessed than regions with low cache miss rates, it follows that the ability to distinguish between regions with high cache miss rates and low cache miss rates could also be used to inform placement decisions. To that end, we considered what type of cache miss rates would be observed for commonly used memory access patterns, and what page fault ordering would arise from these patterns.

For example, in many scientific applications, large data arrays are often sequentially accessed inside inner loop iterations. These access patterns exhibit high spatial locality[2] and can result in a low cache miss rate for systems with large cache line sizes that can hold several array elements. In such cases, a single cache miss to an element located in a previously unloaded cache block is immediately followed by several cache hits to the remaining elements in that block.[3] Similarly, it is often the case that inner loops will repeatedly iterate over a section of an array, before moving on to iterate over an adjacent section of the array (and so on, until the entire array has been accessed). Such an access pattern would likely exhibit both spatial and temporal locality, also resulting in a low cache miss rate. In both cases, the page fault ordering for the pages of these arrays will be sequential in the virtual address space. Conversely, some large data arrays in scientific applications are accessed in an essentially random pattern. Such patterns tend to exhibit poor spatial locality and higher cache miss rates since consecutive accesses are often to different cache blocks. In these cases, successive page faults to these arrays will have no

[2]Spatial locality refers to the tendency for an address to be accessed when nearby addresses have also been recently accessed.

[3]For a typical 128-byte L2 cache line size and 4-byte array element size, this would give a $4/128 = 3.125\%$ cache miss rate.

relationship to each other.

Such examples led us to formulate a heuristic for predicting future cache behaviour based on the differentiation of regions with *sequential* page faults from those regions with *non-sequential* faults. We define a sequential order as faulted within a window of Y total page faults, where X and Y are preset thresholds.[4] This definition allows us to capture sequential fault orderings that may be partially interleaved with another accesses to a region. We hypothesize that pages exhibiting sequential fault patterns are most often associated with memory access patterns that result in higher cache locality, and correspondingly lower cache miss rates, than those memory access patterns most often associated with non-sequential patterns. We note that there exist memory access patterns that will result in a sequential page fault ordering, but low cache locality and a high cache miss rate. For example, in the worst case, accessing one array element per successive cache line with no reuse of these lines will result in a 100% miss rate, yet still appear as a sequential fault pattern. Similarly, a loop that sequentially accesses the elements of an array located on a page, then skips a page, then sequentially accesses the elements on the next page etc. will have a relatively low miss rate, but will be observed to have a non-sequential fault ordering. Nevertheless, while such misidentifications are possible, it is our hypothesis that such access patterns occur relatively infrequently compared to access patterns that conform to our hypothesized correlation.

Table 4.1 shows the results of an experiment designed to test our hypothesized correlation between sequential page fault orderings and low cache miss rates in user data pages. This table compares the L2 cache miss rates of sequentially faulted regions and non-sequentially faulted regions (using the definition of a sequential order described above). Once again, these applications were drawn mostly from the SPEC95fp application suite,[5]

[4]The applications in our test suite have proven to be relatively insensitive to the exact values of these thresholds. For our experiments, we used values for X and Y of 5 and 10 respectively.

[5]In this experiment, each user array is designated as a separate region. This decision is discussed in greater detail in Sections 4.4 and 4.5.2.

Application	Sequential	Non-sequential
applu	2.0	0.4
apsi	–	9.7
buk	3.1	62.5
hydro2d	1.9	2.1
mgrid	0.6	–
mxm	3.1	*
swim	1.1	–
tomcatv	2.1	11.7
turb3d	2.6	4.6
wave	0.4	–

Table 4.1: Processor cache miss rates for sequential and non-sequential user data regions (dashes indicate that no region of that type exists for that application). These rates are the average rates for all regions in each category for each application.

and the experimental environment was a MINT-based simulator configured to reproduce the NUMAchine multiprocessor. Although we describe the SPEC95fp benchmark suite, as well as the simulator and the NUMAchine environment, in greater detail in Chapter 5 (where we present the main results of our experimental evaluation of cache aware placement), we note that the L2 cache in this system is a 1-megabyte unified instruction/data cache with a 128-byte line size. Furthermore, 8 of the 10 applications use 8-byte double array elements (*buk* and *mxm* uses 4-byte integer elements).

The miss rates given in Table 4.1 are the average rates for all memory references to all sequentially and non-sequentially faulted regions in each application. This data provides strong evidence for a correlation between sequentially faulted regions and low miss rates, as the average miss rate for the sequentially faulted regions in each application is equal to or less than 3.1%. With respect to the miss rates of individual regions in this category, only three regions (out of 41) had miss rates greater than the maximum average rate of 3.1%. Additionally, a majority of the applications exhibit average miss rates for their non-sequentially faulted regions that are much higher than the maximum aggregate rate for the sequential regions. The two exceptions are *applu* and *hydro*, with average rates of 0.4% and 2.1% respectively. In the case of *applu*, this appears to be due to an initial non-sequential access phase (with a 6.1% miss rate) that changes to a different access pattern with a much lower miss rate.

Mxm, a matrix multiply program, is a special case whose non-sequential region's cache miss rate is dependent on the size of the matrices being multiplied, and roughly proportional to the ratio between the number of matrix rows and the number of cache lines in the processor data cache. This dependence arises due to the increasing probability of conflict misses when traversing a matrix column as the size of the column (i.e., the number of rows) increases. The exact miss rate is not listed because a limitation in the way our simulator performs virtual-to-physical memory translation prevents it from producing accurate numbers; depending on the matrix size, the method of virtual-to-physical translation will result in either a 100% miss rate for matrices with a number of rows evenly divisible by the number of lines in the processor cache, or a 0% miss rate for all other matrices. We have no reason to believe that this limitation significantly affects the other applications in our test suite (and the simulator has been extensively tested for correctness by others [22]).

4.4 Algorithm Details and Issues

Given this heuristic of equating sequential page fault accesses with low cache miss rates, and non-sequential page fault accesses with high cache miss rates, we devised an algorithm that bases page placement decisions on these predictions called *cache aware placement*. This algorithm operates on regions of pages that are automatically generated at the start of execution for each application when the main data arrays are declared,[6] with each array designated as a separate region. For simplicity, the page placement of all other user data, code pages, and system-allocated memory were not controlled by our placement algorithm, but were allocated using standard first-touch placement.

The rationale for choosing to make each array a separate region comes from a desire for the pages of a given region to be relatively uniform with respect to to the way in which they are accessed, and in their subsequent caching behaviour. Uniform access behaviour over the pages of a region allows us to infer the access patterns for a whole region based on the patterns of a few pages, and allows us to treat an entire region as a single unit for placement decisions. Basing regions on arrays is an attempt to infer this uniformity by making use of user level contextual information.

The flowchart shown in Figure 4.1 describes what occurs when a page fault is encountered. The cache aware algorithm can be divided into two main sections: sequence detection and page placement.

4.4.1 Sequence Detection

Upon creation, each region is initially designated as a *local* region, indicating that its page faults should be satisfied, if possible, by a local memory frame. When a page fault occurs, the page fault handler identifies the region that the page maps to, and examines the region to determine if it is still designated as a *local* region. If so, the page fault handler records the virtual address of the faulting page and compares it against a list of previously recorded page fault sequences. If the currently faulting page is adjacent to the end of previous sequence, the page is considered an extension to it. If this sequence exceeds a predefined *sequence size threshold*, the access pattern for the region is deemed to be sequential, and the region is redesignated as a *remote* region. In the case where the

[6]Recall that we have chosen to design our placement algorithm to target scientific applications, which typically employ large arrays that are statically declared at the beginning of the program.

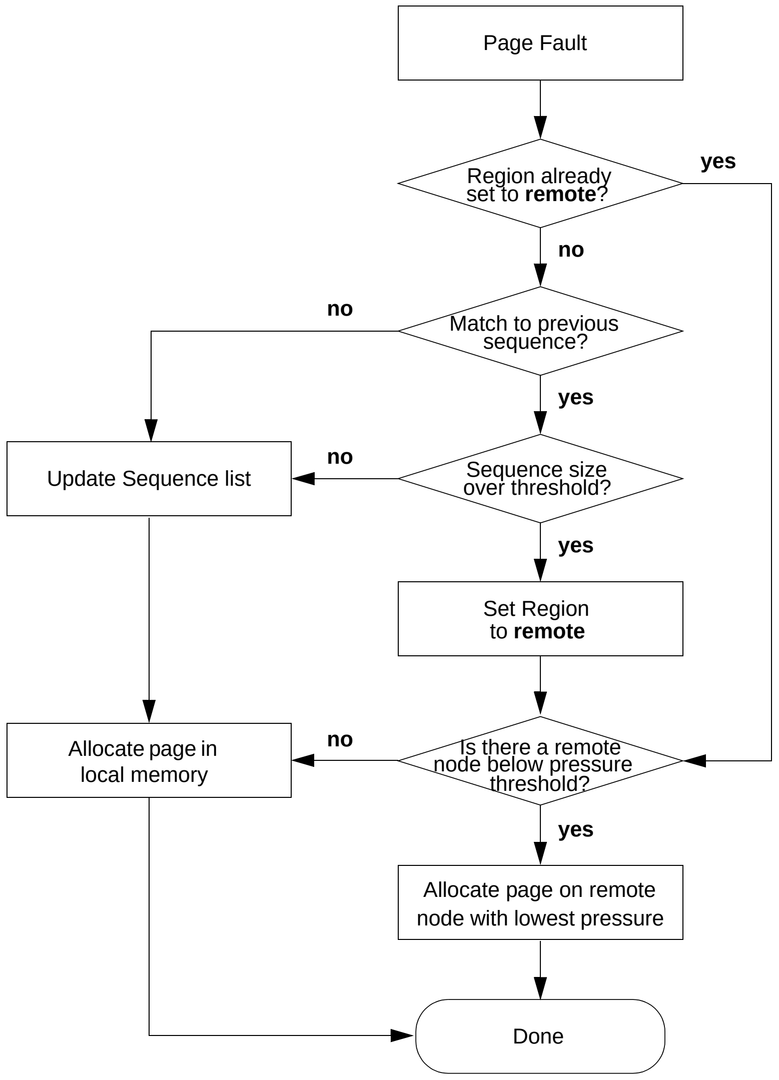

Page Fault
Region already set to **remote**?
yes
no
Match to previous sequence?
no
yes
Update Sequence list
no
Sequence size over threshold?
yes
Set Region to **remote**
Allocate page in local memory
no
Is there a remote node below pressure threshold?
yes
Allocate page on remote node with lowest pressure
Done

Figure 4.1: A flowchart outlining the algorithm for page placement.

faulting page does not extend a previous sequence, the page is recorded as the possible start to a new sequence. In either case, a *fault counter* is incremented to record the total number of faults. If the region is not designated as *remote* before this counter reaches a set threshold, the region is permanently set to be *local*, in which case further page faults to the region bypass the sequence detector.

4.4.2 Page Placement

After the sequence detection phase of the algorithm is complete, the algorithm chooses a memory node on which to allocate the faulting page. Although each region has a *local* or *remote* designation, the placement phase must also consider the *memory usage* at each node before making a placement decision. The *memory usage* at a node consists of all pages on that node belonging to the working set of a currently running process in the system, where the working set of a process is the set of pages accessed by that process during a given time interval. As memory usage on a node increases, the allocation of a local frame to a remote process has an increasing impact on the performance of locally running processes due to the reduction in local memory availability. Because of this, it is important to ensure that memory usage is considered when choosing a node to allocate a page.

Since one of our initial assumptions was that we would not consider the effects of page replacement, a suitable approximation of memory usage is the amount of memory allocated at each node. Although it is generally not true that all allocated pages belong to the working set of a process, an allocated page remains unavailable until freed by the owning process if there is no page replacement. This means that the number of currently allocated pages at a node is an exact measure of the number of unavailable pages.

Given this measure of usage, when allocating a page belonging to a *remote* region, our algorithm initially targets the remote node with the most unallocated frames. However, this selection may be overridden if the usage at the selected node is greater than a predefined threshold. If this threshold is exceeded, and the usage at the local node is below the threshold usage, the local node becomes the target node. This prevents the memory at a remote node from being completely consumed when the usage on the local node is still relatively low. However, if the local node also exceeds the usage threshold, the node with the lowest usage (local or remote) is selected. For pages belonging to *local* regions, the algorithm selects the local node unless no local frames remain. If there are no available local frames, the remote node with the lowest usage is selected.

4.4.3 Limiting Remote Page Placement

When placing pages belonging to *remote* regions, one final consideration we have included is the number of pages that have already been placed in remote memory for a given application. With the algorithm as currently defined, it is possible for an application to have a majority of its pages placed in remote memory if most or all of its regions are designated as *remote* regions. Although *remote* regions have been so designated because they will have a limited impact on performance (due to their predicted low cache miss rate), placing most or all of these regions in remote memory may still amount to a significant number of remote accesses compared to a first-touch placement. With this in mind, we set a *remote allocation threshold* that specifies the fraction of an application's memory that may be placed remotely. This requires that we know the memory requirements of each application in advance, as is the case for the majority of scientific applications. Although such knowledge might seem to imply that we can ensure that no pages be placed in remote memory if there is local memory available, we would also need to know the allocation requirements of other applications on the same node to make this guarantee. When the number of remote allocations exceeds this threshold, all regions in an application are redesignated as *local* regions.

4.5 Discussion

In developing cache aware placement, many design decisions, such as which data to base cache predictions on, the choice of granularity for cache predictions and placement decisions, and how to measure memory usage, required choosing from a number of possible alternatives. In the following sections, we outline some of the alternatives we considered and discuss the reasoning behind the choices we made.

4.5.1 Predicting Cache Behaviour

The core of our placement algorithm is the use of page fault ordering to predict cache behaviour for a region. As we discussed previously, the rationale for using page fault ordering is that it approximately describes the initial access pattern to a region, which we hypothesized would correlate well with cache behaviour. As we showed in Section 4.3, despite the relatively coarse granularity of the access patterns detected, we discovered a correlation between sequential and non-sequential fault patterns and low and high cache miss rates for the regions in our test applications. Additionally, using page fault ordering to predict cache behaviour is consistent with our goals of avoiding specialized hardware, complex computations, or programmer input. Page fault information is easy to extract using existing operating system constructs, requiring the addition of small amounts of data gathering code in the page fault mechanism. These additions produced no noticeable overhead in our implementation.

An alternative to using page fault ordering to infer access patterns to a region is to observe a finer granularity of access to each region. Choosing a finer granularity would allow our algorithm to make finer distinctions between different access patterns. For example, observing accesses at a cache line granularity would let us distinguish between a pattern that sequentially reads every cache line in a region from a pattern that strides through the region, i.e., regularly reads every x'th line. Both of these patterns would result in sequential page faults, but the former cache line pattern is more likely to arise from an access pattern that has a higher cache hit rate (e.g., the former cache line pattern can arise from the commonly seen access pattern that sequentially touches every element on every cache line, while the latter one cannot). However, choosing a finer granularity would require the use of specialized hardware to monitor cache line accesses, or possibly extensive modifications to the memory subsystem with associated high overheads to capture this information, tradeoffs we were unwilling to explore given the success of our prediction scheme using page fault information.

A second alternative is to measure the actual cache hit rates for the first few pages accessed in a region. This would give us the benefit of using the actual cache hit rates for these first few pages to predict the future cache hit rates for all the pages in the region, rather than having to infer these cache hit rates from the page fault ordering as we have done. However, choosing this option would also have required the use of monitoring hardware to observe cache hits and misses, once again violating our goal of avoiding specialized hardware.

4.5.2 Granularity of Cache Predictions and Placement Decisions

As we discussed in Section 4.3, making predictions on a page granularity based on page fault data is problematic because the first access to a page immediately triggers its allocation and placement. Instead, our algorithm makes predictions and placement decisions on groups of contiguous pages called regions to allow the gathering of multiple data points.

Having decided on regions consisting of multiple pages, there is still the issue of how large these regions should be. As described in Section 4.4, we chose to designate each array in a target application as a separate region in an attempt to keep the access patterns and caching behaviour over a region relatively uniform. Choosing to make placement decisions at a larger granularity (e.g., multiple array regions) would likely result in less uniformity of these characteristics since we would have less reason to expect the pages of multiple arbitrary arrays to exhibit similar access patterns or caching behaviour than the pages of a single array.

The other alternative is to make placement decisions on a smaller granularity than a single array. Although we show later in Section 5.3 that the majority of arrays in the applications we studied have uniform caching behaviour, a small number of applications have arrays that contain groups of pages with distinctly different memory access behaviour. In such cases, it might be more appropriate to divide an array into several regions corresponding to these distinct groups. It might be possible to identify these groups by their page fault ordering as well. For example, one could imagine that allocating a local or a remote frame for the next page fault to a region could depend on whether the previous X page faults to that region were sequential or non-sequential.

Failing that, dividing an array into several regions would likely increase the complexity of region creation, since these subgroups of pages with differing memory access characteristics would be difficult to identify a priori. One might imagine a compiler prepass that would attempt to analyze the access patterns to each array and determine which sections might be placed in different regions, or having the user specify regions based on their knowledge of the access patterns to their arrays. However, such solutions do not adhere to our goal of low complexity and minimization of user responsibilities.

4.5.3 Selecting a Target Memory Node based on Memory Usage

Although our algorithm bases its placement recommendations on the predicted cache behaviour of each region, these recommendations are considered in the context of the memory usage at each node in the system. As we have stated previously, we use *memory usage* to denote the set of all pages on a node that belong to the working set of a currently running process in the system, where the working set of a process is the set of pages used by that process during a given time interval. Given a decision to allocate the pages of a region on a remote node, we would like to choose a node or set of nodes where there exist a sufficient number of currently unused pages to meet our needs.

Since have assumed the absence of page replacement in our system, the number of allocated pages is an exact measure of the unavailable pages on a node. However, our algorithm uses the measured usage at a node at the moment a placement decision must be made. An alternative option is to measure the usage over a set time interval to gain an indication of the historical memory usage at the node. However, a measure of this kind is likely to be very highly workload dependent, and we would need to investigate real workloads being run on typical CC-NUMA systems to see whether such a scheme would be beneficial.

Alternatively, if we were to allow page replacement, measuring allocated pages might constantly overstate the usage on each node, as some programs could give up allocated pages not in their working set and still suffer very little performance degradation. In such an environment, one of the following alternatives for measuring memory usage might be considered.

Free List Size

One approach to measuring memory usage in the presence of page replacement is to use the length of the free list[7] at each node. Since the free list contains pages that are available for fulfilling future allocation requests, the mean length of the free list over a given time interval might be a good estimate of the availability of memory pages for allocation. However, this presumes that the working set of the applications we are running can be accurately determined by an LRU-type algorithm. While studies have shown that LRU-type algorithms tend to produce the best results on average, we have previously discussed how LRU and its derivatives do not give an accurate reflection of the current working set for some applications.

Another complication involved with using the length of the free list as a memory usage indicator is that the application of the replacement algorithms such as the second chance algorithm may not be uniform throughout the system. More specifically, the free list length can be influenced by the speed of the clock algorithm sweeping through the pages on a node. Given differing second chance clock speeds on different nodes might lead to

[7]Recall that a clock-based algorithm (see Section 2.4.1) identifies the working set of an application by approximating an LRU ordering on pages and placing those pages that have not been accessed within a certain number of sweeps of the algorithm on the free list (a list of pages that are available for satisfying future allocation requests). This leaves only those pages that have been accessed in the recent past in memory, with the assumption that such pages are the most likely ones to be accessed in the near future.

variability in the interpretation of what lengths constitute high and low memory usage. In the Tornado operating system, under which we have conducted our experiments, there is the further complication that there can be several lists of free memory on each node, each possibly operating on its own clock speed. Coordinating these clocks becomes important so that if we are to use the lengths of these free lists to measure the total usage at a node.

Statistical Approximation of Memory in Use

One way to avoid the complication of differing clock speeds is to uncouple the measurement of memory usage from the page replacement mechanisms being used and perform separate measurements on the amount of memory in use. Borrowing concepts from the second chance approach, we can randomly select a small set of memory pages in each node to be unmapped from the TLB at the beginning of a set clock period. During the clock period, we can make note of any of these unmapped pages that are referenced.[8] At the end of each clock period, the fraction of these unmapped pages that have been referenced during that period can be counted, and this fraction used to approximate the entire fraction of memory at the node currently in use. Such an approach has been proposed in recent work on virtual machine systems that share common resource pools, where the authors have looked at precise memory usage accounting to allow for efficient memory sharing amongst virtual machines [58].

4.6 Open Issues

In addition to the specific algorithmic issues we have addressed in Section 4.5, there are also some more general concerns relating to how a static page placement policy such

[8]Since the page mappings are not cached in the TLB, a memory reference traps to the operating system which must reload the entry, at which time a reference bit in the page table entry for the page can be set.

as our proposed policy can interact with other parts of the operating system. In the following sections, we discuss how scheduling concerns, dynamic placement policies, and page replacement relate to static page placement.

4.6.1 Scheduling

The way processes are scheduled can have a significant impact on the availability of memory at each node. For example, consider a situation where two nodes of a multiprocessor system have unused processors. One of the nodes is completely unused, i.e., all of its processors are free, and the other has some unused processors and some processors in use. If the scheduler considers only processor availability when making its scheduling decisions, then scheduling the new process on an unused processor on either node would be acceptable. However, it is clear that scheduling the process on the partially used node will increase the competition for local memory on that node.

The algorithms used for scheduling multiprogrammed workloads in a multiprocessor environment are often simple extensions of techniques developed for uniprocessor operating systems. In particular, the overriding concern of most uniprocessor and multiprocessor schedulers is the efficient utilization of the CPU [51]. For a uniprocessor scheduler, this concern can be distilled down to the decision of when and for how long a process should be scheduled to run. For multiprocessors, the scheduler must decide not only when and for how long a process will be scheduled, but also on which processor it will be scheduled. In answering these questions, a scheduler attempts to manage the use of a resource, i.e., CPU cycles, such that a particular goal is achieved, e.g., maximize throughput, enforce relative priorities between processes, etc.

For the most part, existing schedulers tend to ignore the availability of memory resources, and attempt to achieve their goals while considering only CPU cycles as the resource that affects efficiency and performance. Early on, it was recognized that cache context is a very important factor in performance, giving rise to scheduling policies that tend to keep processes on the same processor once they have been assigned there, e.g., affinity scheduling [51]. Previous studies on NUMA systems have also shown that locating the threads of a parallel application close together can improve performance by minimizing the costs of accessing shared memory pages [7]. However, it is typically important that two competing processes not be assigned to the same processor. Hence, process placement is typically done by the application, and is static. Despite this, taking memory into consideration can still be important for performance.

Cache aware placement addresses the problem of memory locality strictly from a memory management view of things. However, integrating memory management and scheduling decision making could also be effective. For example, one could incorporate the scheduler by making it aware of memory pressure in the system, and having it use this information to guide its decisions. Such a solution could become very complex, as the scheduler must now deal with balancing the usage of an additional resource, i.e., memory.

4.6.2 Program Phase Changes and Process Migration

By choosing a static page placement at allocation time, we may leave ourselves vulnerable to phase changes in the memory usage patterns of our applications. For applications that undergo phase changes in their memory access behaviour, we may choose a static page placement that makes sense for the initial memory reference patterns of an application, but becomes inappropriate when these memory reference patterns change during the application's execution. Similarly, a static page placement can be undermined if the scheduler chooses to migrate a process away from the processing node that it originated on.

While our solution is a static one, phase changes and process migration can be addressed using existing dynamic solutions to the memory locality problem that we discussed earlier, such as page migration. For example, one could imagine a page placement solution, such as our own, working in conjunction with a page migration policy. The static page placement made at allocation time could achieve a good placement from the outset of the application execution, and the migration system could help this placement adapt to future changes in memory usage.

4.6.3 Page Replacement

One final issue to consider is the interaction between our proposed cache aware placement policy and the page replacement policy, as the choice of a victim page for page replacement could have an impact on the effectiveness of our policy by perturbing the static page placement. For example, evicting a page that we have predicted will be poorly cached and that has been placed in local memory could be harmful if that page is later faulted back into remote memory (e.g., because there are no available frames in local memory). While the minimization of page fault activity is the primary concern of the page replacement policy, it might be fruitful to consider placement concerns when the replacement policy has a choice between two or more equally likely victims. This can be further complicated by multi level approaches to page replacement such as the popular two-level approach consisting of a global replacement policy to choose a victim process, and a local policy specific to the victim process that chooses one of its pages for reclamation[9]. In such a case, it might be necessary to incorporate placement information at both levels of the replacement scheme while ensuring that such changes do not have a negative effect on the amount of page fault activity.

The choice of a usage-based memory pressure heuristic can also be affected by the type of page replacement scheme in use. If we choose to allocate a page on a node that has a high number of allocated pages and low usage, we implicitly assume that a page from that node will be reclaimed to make room for our new allocation. This means that our notion of memory usage should be compatible with the replacement policy. For example, it may or may not be appropriate to choose a memory pressure heuristic based on the free list size if the global replacement policy is based on the relative page fault rates of processes. Biasing our local replacement policy to choose a remote memory page over a local one as we suggested above could also have a negative effect by making it less likely that a victim page is chosen from the node we are allocating on.

Finally, a further avenue of research might be to investigate whether the differences in performance attributable to a change in placement policy are noticeable in an environment where there is significant page replacement activity. Since the latency to disk is several orders of magnitude greater than the difference in latency between local and remote memory in most multiprocessors, it may be valuable to determine the level of page fault activity that makes placement concerns insignificant.

4.7 Summary

The main shortcoming of first-touch placement for single threaded applications is that it allocates local memory frames to page faults on a first come first served basis, rather than to those pages that will be most highly accessed. To address this shortcoming, we propose a new policy called *cache aware placement* that differentiates regions of pages based on their predicted cache behaviour. These predictions are based on differentiating between sequential and non-sequential page fault orderings in the virtual address space. We have found that regions of memory that exhibit sequential page faults correlate well with low cache miss rates, while regions that exhibit non-sequential faults often have higher cache miss rates than sequentially faulted regions in the same application. By prioritizing the placement of non-sequential regions in local memory, cache aware placement attempts to reduce the number of remote memory accesses made by an application compared to first-touch placement.

Chapter 5

Experimental Methodology and Results

In this chapter we present the experimental evaluation of our allocation policy in both simulated and hardware multiprocessor environments. These experiments show that the application of cache aware placement can result in a noticeable improvement in the performance of several programs in our test application suite compared to first-touch placement by reducing the number of remote memory accesses made.

5.1 Experimental Environment

The experimental results presented in this chapter were produced under two different environments. The first of these environments consisted of the University of Toronto NUMAchine multiprocessor[21, 23] running the Tornado operating system[17]. The second environment consisted of an internally modified MINT-based simulator[22, 54] configured to simulate the NUMAchine hardware. We describe these environments in detail below.

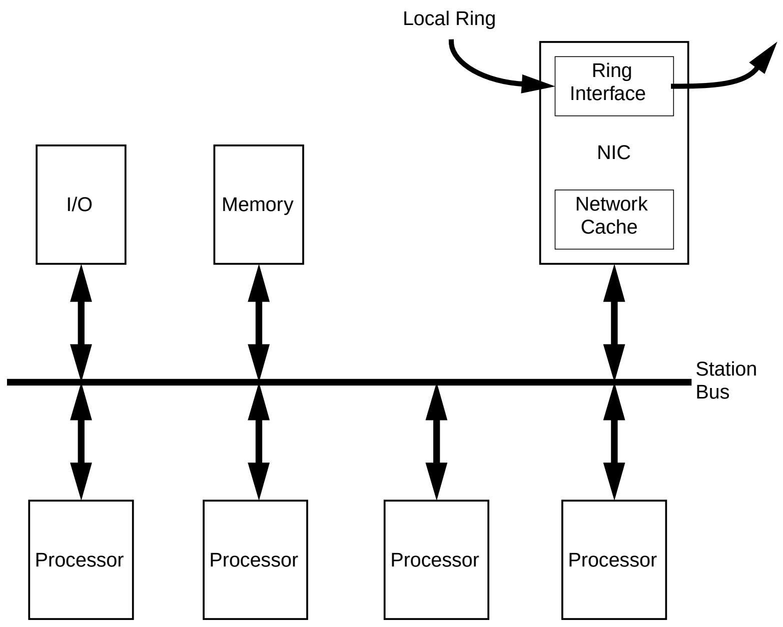


Figure 5.1: A NUMAchine station. Each station consists of four processors, a local memory, an I/O card, and network interface card (NIC) connected by a station bus. The NIC connects to the local ring.

5.1.1 The NUMAchine Multiprocessor

NUMAchine is a hierarchical, shared memory CC-NUMA multiprocessor constructed entirely with commodity components and custom designed boards utilizing programmable logic devices. The NUMAchine architecture is based on *stations* of up to four MIPS processors. The configuration of each station is similar to an SMP multiprocessor and is shown in Figure 5.1. All processors on a station are connected by a shared bus to the same local memory, I/O card, and network interface card. The network topology is a hierarchical ring design with several stations connected together by a single unidirectional *local ring* at the lowest level of the hierarchy. Several of these local rings may be connected together by another unidirectional *global ring* to form a larger system. Figure 5.2 shows the basic architecture for the NUMAchine system.

The specific configuration used for our experiments includes 16 MIPS R4400 proces-

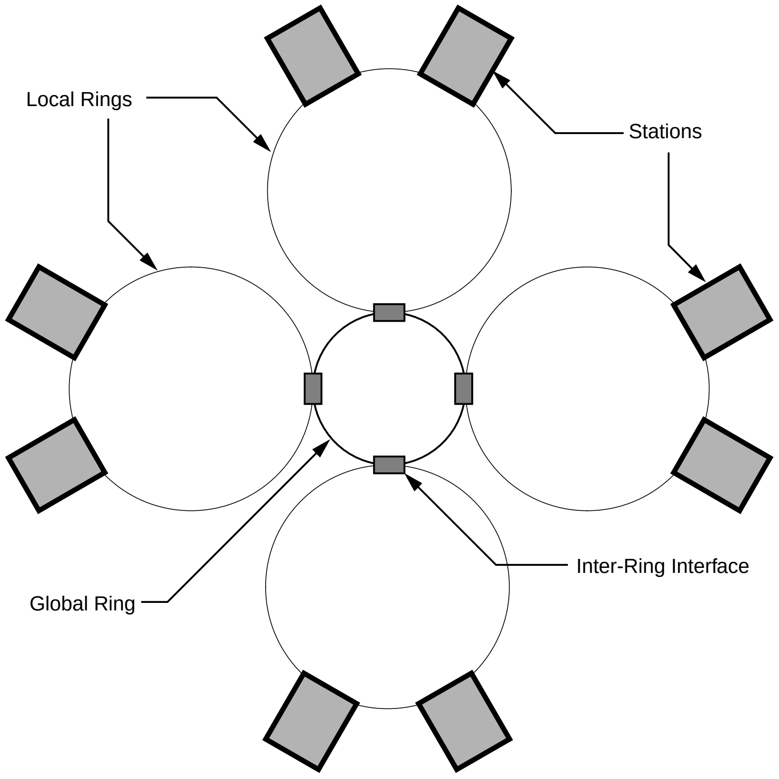


Figure 5.2: The NUMAchine multiprocessor architecture. The network topology is a two-level ring hierarchy. Although only two stations are shown per local ring, as many as eight stations can be accommodated on a local ring.

sors running at 150 MHz divided into nodes or stations of four processors. Each processor features separate 16k L1 instruction and data caches with 32 byte line sizes, and a 1Mb unified L2 cache with a 128 byte line size. Each station shares 96 megabytes of local memory and an 8 megabyte network cache. The four station system is connected together using a single local ring. The use of only a single local ring with no global ring was due largely to unresolvable technical difficulties with the global ring implementation.[1]

Adding a global ring would imply that remote stations could differentiated into those that reside on the same local ring, and those that reside on another local ring. These

[1]The global ring implementation had a tendency to produce random bit errors.

level of hierarchy	150-MHz PCLKs	50-MHz SCLKs
L1 cache	1	*n/a*
L2 cache	6	*n/a*
Local memory	135	45
Local network cache	165	55
Other L2 cache (on station)	255	85
Rem. mem.	594	198

Table 5.1: The measured access latencies to different parts of the memory hierarchy in NUMAchine. The latencies in the first column are measured in processor clock cycles, while the second column lists them in system clock cycles. These measurements were taken on an unloaded system.

latter stations would require more latency to access, since accessing them would involve traversing the global ring. In such an environment, we could envision changing our algorithm that take advantage of this knowledge. In particular, it might be appropriate to order nodes based on the latency to access them when attempting to allocate local or remote regions. For example, under our current policy, if a region is non-sequentially faulted, it is placed in local memory if there are local frames available, or remote memory if there are no local frames. With a global ring, we might order remote nodes in terms of their latency, so that if there are no local frames, such a region will be preferentially placed in the remote node with the smallest access latency. Similarly, we might place sequential regions in the nodes that are the furthest away, to save room in closer remote nodes for non-sequential regions that cannot be placed in local memory.

Table 5.1 summarizes the measured read latencies to different parts of the memory hierarchy on an unloaded system.[2] The first column refers to the number of processor

[2]Since will not use shared memory between processes in our experiments, the write latencies we experience will be the same as these read latencies.

clock cycles for each type of reference. The second column gives the number of system clock cycles for each reference type (the ring network and various controllers in the system run on a 50 MHz clock).

5.1.2 The Tornado Operating System

The Tornado operating system running on NUMAchine is an object oriented operating system designed specifically for shared memory multiprocessors. Tornado was initially designed with the goal of developing novel structuring techniques to deal with the issue of scalability in shared memory systems. Tornado was also designed to be flexible, providing infrastructure that allows different policies and solutions to various problems to be made available to user applications.

Implementing Cache Aware Placement in Tornado

Figure 5.3 depicts a simplified view of the object model supporting memory operations for a program in Tornado. Each application in Tornado is associated with an address space, which defines its protection domain and contains non-overlapping, contiguous sequences of virtual memory pages, called regions, that it can access. Since Tornado uses a *memory mapped file interface* [34], which abstracts all accesses to a file into virtual address space references, each region is mapped directly to a continuous portion of a file that may reside on secondary store (for user allocated memory, this is a swap file). The key objects depicted in this figure include the Program object, Region object, File Cache Manager, and Cached Object Representative:

Program: A *program* is the root object for memory management for an application. All TLB misses are initially issued to the program object, which forwards them to the appropriate *region* object.

Region: We have previously used the term *region* to denote a contiguous portion of the virtual address space. In Tornado, a region is more specifically defined as a program

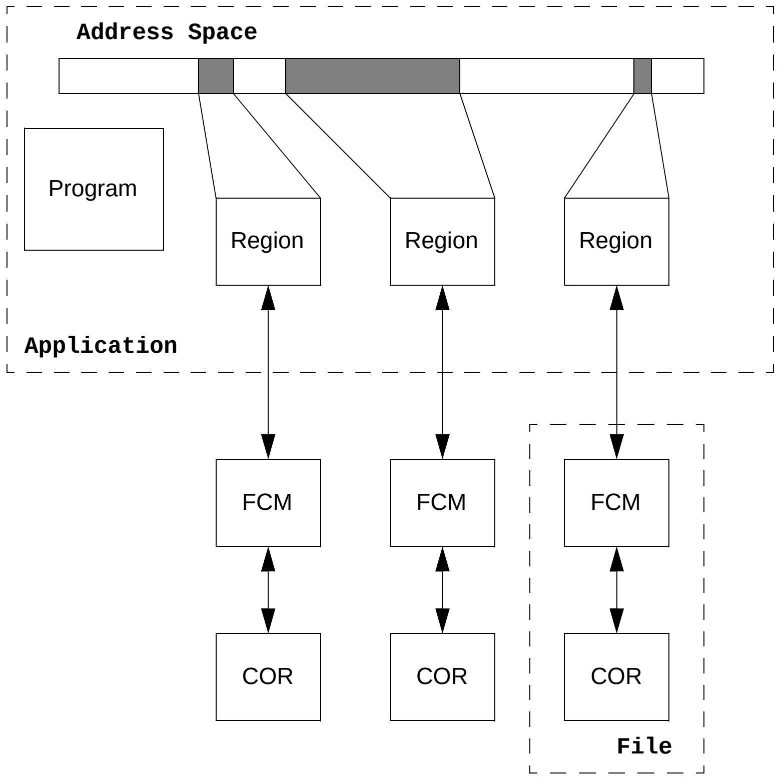


Figure 5.3: Tornado memory subsystem infrastructure.

addressable portion of the address space with corresponding protection attributes (read only, read/write). The region object resolves page faults by communicating with the *file cache manager*.

File Cache Manager (FCM): The FCM is responsible for managing the file cache of a specific file. This includes the allocation and placement of frames to satisfy page faults. Each FCM is associated with a single *cached object representative*, which handles read and write requests generated by the FCM.

Cached Object Representative (COR): The COR is the file system representative in the kernel, with each open file having an associated COR. The COR is file system specific; there are several types of CORs that handle reads and writes from and to different types of file systems.

Our implementation of cache aware placement involved changes to the page fault handling portions of the region object and FCM. The sequence detection phase of the algorithm was incorporated in the page fault handling path through the region object. The FCM was modified to handle allocation to a memory module specified by the cache aware code in the region object.

In a UNIX-based system, implementation of cache aware placement would require replicating some of the above infrastructure. In particular, we would require a data structure to store region information for the arrays allocated by each UNIX process, as well as library code to create these regions. The page fault handler would also need to be modified to match page faults with regions in this data structure, as well as to implement the sequence detection handled by the region object in Tornado.

Defining Algorithm Thresholds

In Section 4.4, we described a number of predefined thresholds to be used in our cache aware placement implementation. In the sequence detection portion of the placement algorithm, these thresholds include the *sequence size threshold* (the number of pages in a sequence necessary to declare that a region is being accessed sequentially), and the *fault threshold* (the number of total page faults allowed before a region is declared as non-sequential). The minimum number of page faults to determine a sequential or non-sequential series is two. Having experimented with several different values for each threshold, we found that the identification of sequential and non-sequential regions was relatively insensitive to a wide range of values. For the experimental results reported in this chapter, we set the sequence size threshold to five, and the fault threshold to ten.

The other threshold discussed in Section 4.4 relates to the number of pages that can be placed in remote memory before no more remote placements are allowed. We arbitrarily chose to begin with a 50% limit on the percentage of pages that an application can place in remote memory. Since this 50% threshold appeared to strike a good balance between local and remote allocations, we did not vary it further for our experiments.

5.1.3 The NUMAchine Simulator

The second environment under which experiments were run consisted of an internally modified MINT-based simulator configured to reproduce the same operating parameters of the NUMAchine hardware. Although the NUMAchine hardware and Tornado operating system are fully functional, we required the simulator to probe more deeply into the behaviour of our applications and to obtain data that the monitoring systems in our hardware could not provide. For example, we were able to record the hit and miss data at all levels of the memory hierarchy in the simulator environment, as well as break down this data on a per page basis.

The NUMAchine simulator is an execution driven simulator, meaning that it takes binary code as input and executes it using an interpreter and virtual model of the processor. The simulator can be divided into a front end and back end. The front end is the MINT half of the simulator and is responsible for creating virtual processors, as well as executing most of the instruction stream. The back end is called when the front end encounters a load, store, or synchronization operation. When one of these instructions is encountered, the virtual processor that is executing the instruction is blocked, and the back end is called upon to calculate the appropriate delay for a response to occur by simulating the request passing through the various structures in the NUMAchine architecture, e.g., caches, bus, network, etc. In particular, the network structures have been modeled in detail to accurately reflect congestion and occupancy in the network. When the appropriate delay has been calculated, the virtual processor is scheduled to be unblocked at the proper simulation time with the proper result from the request.

Finally, while the simulator actually executes the binary calls, MINT does not simulate kernel calls, but executes these calls natively on the host machine or mimics the behaviour internally. Similarly, OS-related behaviour such as memory management is not simulated; the functionality is reproduced outside the simulation environment. Thus, no OS overhead is included in the simulation results.

Experimental results examining the correctness of this simulator, as well as comparing how closely simulated results correspond to results obtained on the NUMAchine hardware have been presented elsewhere[22]. In the work cited, simulator correctness was validated using synthetic benchmarks measuring easily calculated results, e.g., the number of cache hits and misses of a well described read pattern, as well as comparing the results of benchmark programs run on the simulator and on NUMAchine. These latter experiments were also used to compare performance results between the two platforms. They found that while the simulator tended to under report execution times by a factor of 2 to 3, the relative relationship of execution times as the number of processors were changed correlated very closely, i.e., the speedup curves for each platform were very similar. The difference in absolute reported times may be a function of the use of parallel applications; in our experiments on single threaded applications, the absolute difference in execution times was much smaller, with times differing by approximately 10%.

For all of the following experimental results, we will clearly distinguish between those produced under the simulated environment, and those produced from experiments using the actual hardware.

5.2 Application Test Suite

For our test application suite, we have chosen ten applications from a variety of sources. *Buk* is an implementation of a bucketsort algorithm taken from the NAS Perfect Benchmark suite[3]. *Mxm* is a personal implementation of matrix multiply. Both applications use 4 byte integers as their main data type. The remaining applications (*applu*, *apsi*, *hydro2d*, *mgrid*, *swim*, *tomcatv*, *turb3d*, and *wave*) were taken from the SpecFP95 benchmark suite[13]. These applications all used 8 byte doubles as their main data type. Table 5.2 gives a description of each application, along with the number of regions and the total amount of user memory allocated for both our single application workload and multipro-

Application	Regions	Memory (single/multi)	Description
applu	7	240MB/103MB	PDE solver
apsi	1	63MB/63MB	solver for mesoscale and synoptic potential temps
buk	3	134MB/134MB	bucketsort
hydro2d	6	172MB/172MB	solver for hydrodynamical Navier-Stokes equations
mgrid	3	216MB/55MB	multigrid solver for 3D potential fields
mxm	3	192MB/108MB	matrix multiply
swim	11	225MB/88MB	weather predictor
tomcatv	7	126MB/126MB	mesh generation with Thomson solver
turb3d	12	195MB/130MB	Navier-Stokes solver using pseudospectral method
wave	1	107MB/107MB	particle simulator

Table 5.2: A summary of the applications in our benchmark suite. For each application, the number of regions above 1-megabyte is given, as well as the amount of memory allocated in both our single program and multiprogrammed experiments.

grammed workload experiments. The input data size for each application in the single application experiments was chosen such that the memory allocated was larger than the size of local memory (96MB).[3] Some of these sizes were scaled down in the multiprogrammed case so that total memory allocated did not exceed the total memory in the system.

[3]The lone exception for this was *apsi*, whose smaller size was chosen to keep execution times reasonable.

5.3 Distribution of Memory References in User Regions

The first experiment we present examines the cache hit rates and memory reference patterns for the pages in each of the regions in the applications of our test suite. One of the underlying assumptions we have made in designing our page placement policy is that the number of memory accesses for each page in a given region would be relatively uniform over the entire region. This assumption is implicit in our choice to make placement decisions for entire regions, rather than for individual pages, as well as our choice to base these placement decisions on the pattern of the first page faults occurring in each region. If this assumption is not true, then making a single placement decision for an entire region could be a poor choice. For example, if each page in a region is accessed a small number of times, then placing that region in remote memory should not affect performance. However, if some pages in a region are lightly accessed, and others are heavily accessed, then making a single decision to place that region in local memory or remote memory may not make as much sense as making decisions at a smaller granularity, i.e., placing part of the region in local memory, and part in remote memory. In this experiment, we examine the validity of this assumption by plotting the distribution of memory reference counts and cache hit rates over all the pages in each region in our applications and show that in the vast majority of cases, these characteristics do not vary greatly over the pages of each region.

For this and all subsequent experiments, we chose to limit the scope of our placement policy to user data regions. User arrays that were larger than 1-megabyte were treated as separate regions. These arrays were identified by manually examining the source code, and adding a library call to a region creation routine for each array declaration.[4] The

[4]One could imagination a straightforward compiler preprocessing source-to-source transformation pass that would automate this process.

minimum 1-megabyte size for an array to be considered a region was chosen to correspond to the size of the L2 cache on the MIPS R4400 processors. By only considering arrays larger than this size, our policy only involves itself with regions that do not fit entirely in the processor cache hierarchy and are thus, more likely to be poorly cached. As described in Chapter 4, all other user data, code pages, and system regions were allocated under a first-touch placement policy.

The experimental environment we used for these tests was the NUMAchine simulator running each application in isolation. Figure 5.4 shows an example of the data we collected on the memory reference counts[5] for each application. Specifically, the figure shows three graphs representing the data for each of the three regions in *buk*. These graphs were created by first recording the number of L2 cache misses going to memory for each page in each region. For each region, a median value was calculated from these per page memory reference counts, which was then used to normalize the counts for each page so that the x-axis range on each histogram would be the same.

For example, consider a hypothetical region with five pages. In the simulator, we record that there were a total of 60 memory references to addresses within this region. We also record that the number of memory references for the five individual pages are 16, 10, 8, 8, and 18. The median value for the memory references per page is 10, which we use to normalize the counts for all the pages in the region. This means that page 1 has 160% of the median number of memory references, page 2 has 100%, and pages 3, 4, and 5 have 80%, 80%, and 180% respectively.

Once these normalized values were calculated, the per page memory reference distributions for each region were plotted as histograms. Figures 5.4 and 5.5 show the histograms for the user regions in *buk* and *mgrid*, whose distributions are representative of the majority of regions found in our benchmark suite. Each histogram was created using a bin size of 1% of the median memory reference count for the region, with the

[5]A memory reference is generated by an L2 cache miss.

application	region (array name)	within 1%	within 5%	within 10%
buk	key	37%	93%	99%
	rank	100%	100%	100%
	keyden	100%	100%	100%
mgrid	u	32%	59%	83%
	v	92%	93%	94%
	r	49%	77%	83%

Table 5.3: The distribution of per page memory reference counts over all pages for each region in *buk* and *mgrid*. The columns show the percentage of pages in the region that are within 1%, 5%, and 10% of the median count for each region.

y-axis giving the percentage of pages in the region falling into each bin. From these histograms, we can see that most pages in each region are accessed the same number of times.

The same memory reference data for *buk* and *mgrid* is summarized in Table 5.3. The first column in this table gives the user array name corresponding to each region. The second column in this table gives the percentage of pages for each region that are within plus or minus 1% of the memory reference counts for the median page in each category. Similarly, the third and fourth columns give the percentage of pages for each region that are within plus or minus 5% and 10%, respectively. From this table we can see that in three regions, over 90% of the pages have memory reference counts within 1% of the region median. In all regions in these two applications, over 80% of the pages have counts within 10% of the median.

Another way to examine the variability of access behaviour to a region is to look at the per page L2 cache miss rate. Figures 5.6 and 5.7 show histograms of the per page L2 cache miss rates for the regions in *buk* and *mgrid*. These histograms were created in a similar fashion to those in Figure 5.4. However, rather than use the per page memory

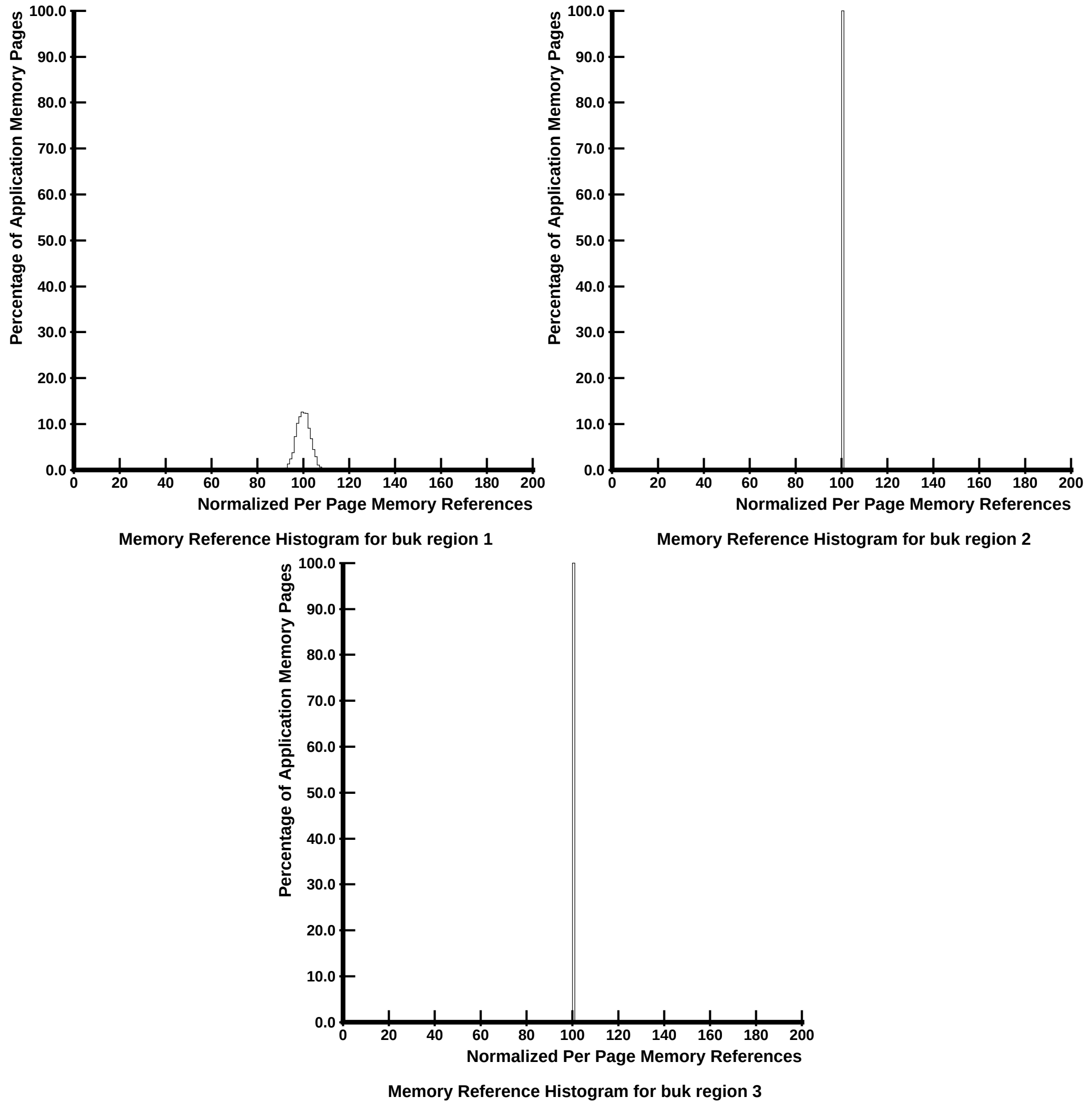


Figure 5.4: The distribution of per page memory references over all the pages in each region in *buk*. The x-axis for each histogram is divided into bins whose size is 1% of the median memory reference count for each region.

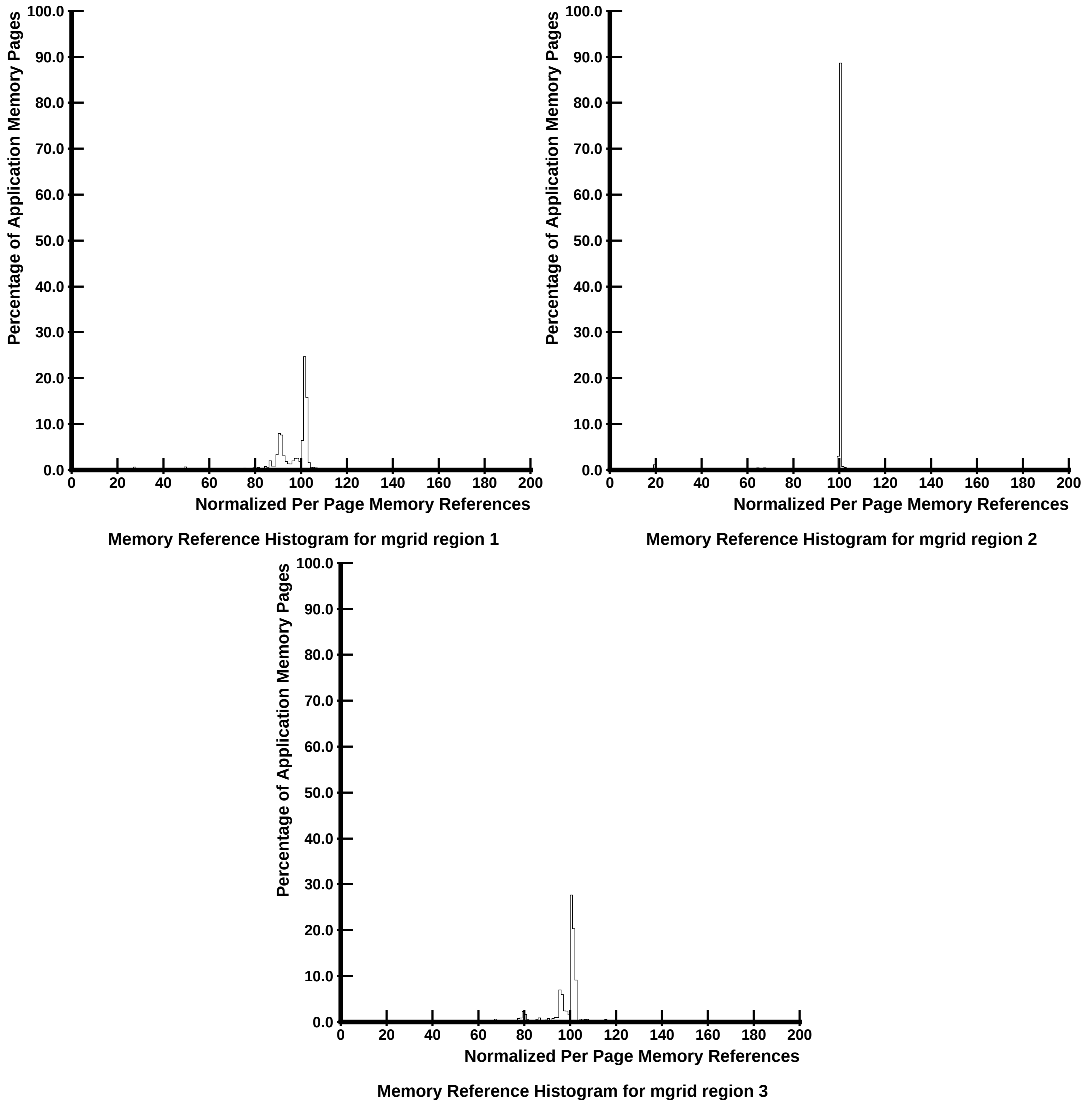


Figure 5.5: The distribution of memory references per page over all the pages in each region in *mgrid*. The x-axis for each histogram is divided into bins whose size is 1% of the median memory reference count for each region.

reference count, these histograms use the L2 cache miss rate to place pages into bins. We also left the cache miss rates for each page unmodified rather than normalizing them to the median value since cache miss rates are already normalized to the same 0% to 100% range. In each histogram, the y-axis shows the percentage of pages that have the same L2 cache miss rate indicated by the x-axis. As is evident from these graphs, the per page miss rate data also shows very little variance across each region in *buk* and *mgrid*,

From these results, we conclude that the pages in these regions do not have much variance with respect to memory reference counts, which, in turn, implies that the impact on performance of placement in a local or remote memory node does not vary from page to page in a given region. For the purposes of the cache aware placement policy described in Chapter 4, these clustered distributions lend support to the idea that making placement decisions based on a region granularity, rather than differentiating between pages in a region, is reasonable.

The complete data for the remaining applications in our test suite, including graphs showing distributions of both per page memory references and L2 cache miss rates, can be found in Appendices A and B. The data in these graphs follow similar distributions to the results shown above, with the exception of two applications, which we examine in the following section.

Memory and Cache Behaviour for *Apsi* and *Wave*

Although the majority of applications in our benchmark suite have low variance distributions of their per page memory reference and cache miss data similar to *buk* and *mgrid*, there are two applications that do not follow this pattern. Figures 5.8 and 5.9 show the per page memory reference count distribution and per page cache miss distribution for *apsi* and *wave*, each of which contains a single user region. For both of these applications, the histograms show several spikes at distinct memory reference counts and cache miss rates, possibly indicating distinct subsets of pages in each region with their own differing

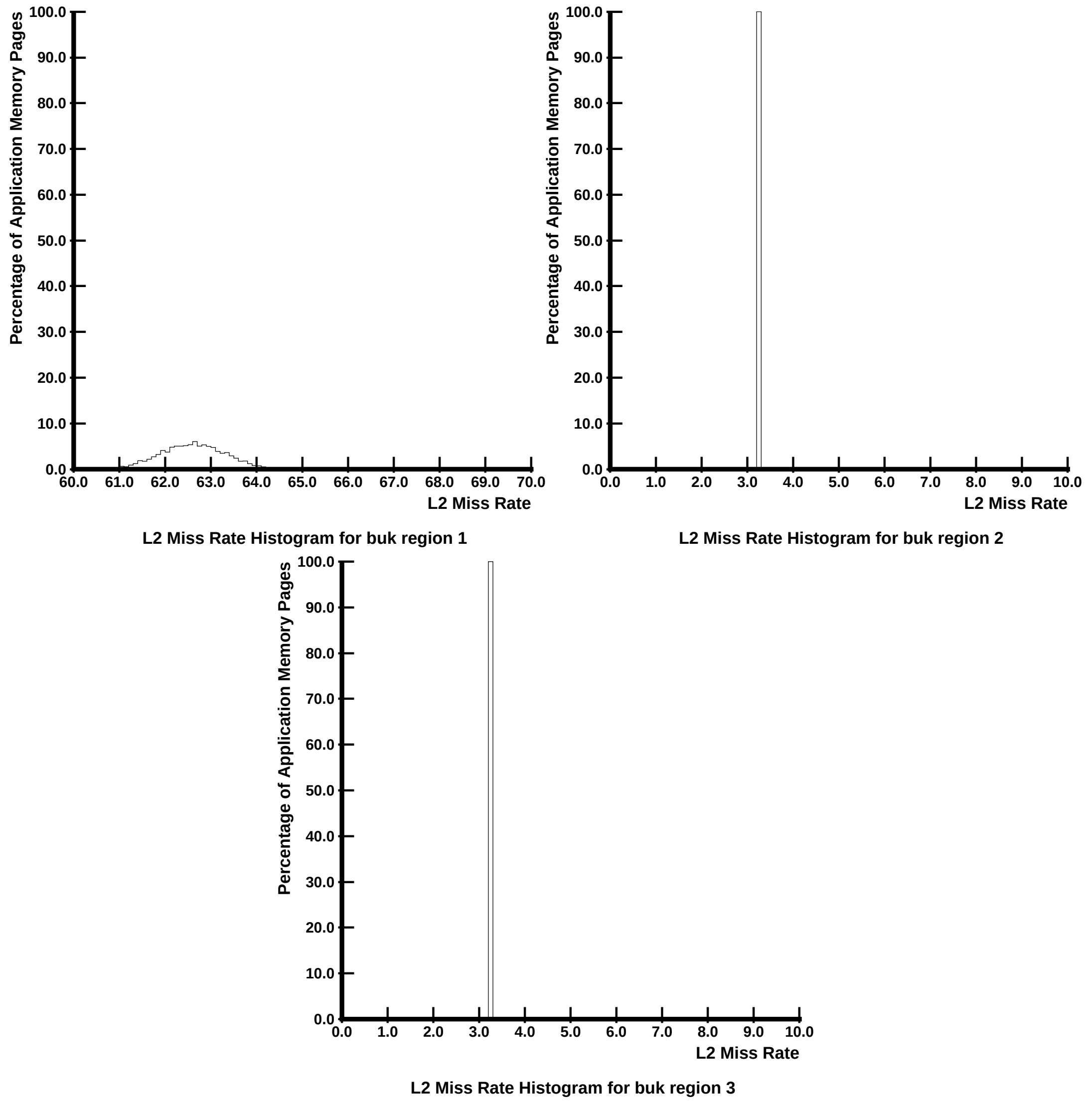


Figure 5.6: The histogram distribution of per page L2 miss rates for each region in *buk*.

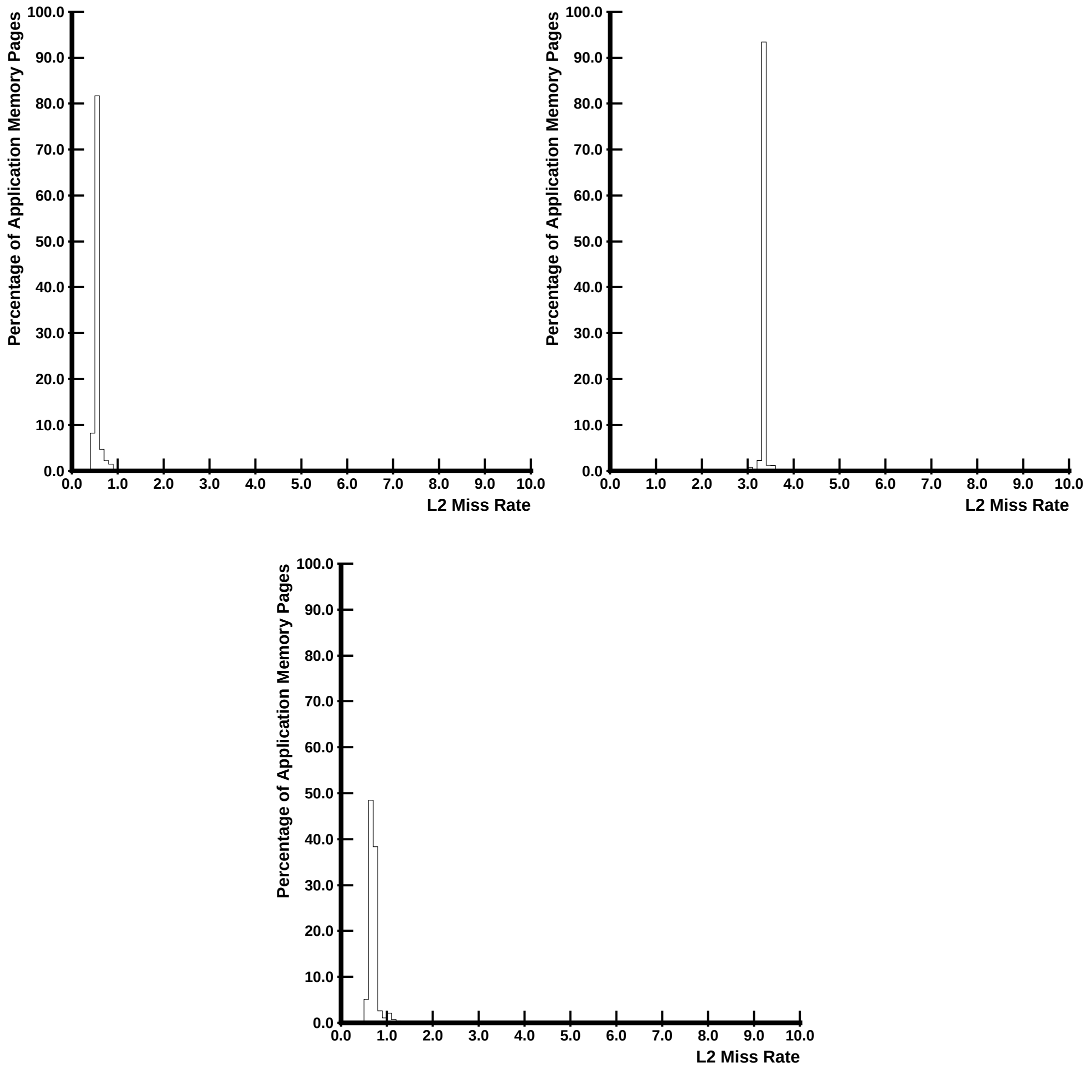


Figure 5.7: The histogram distribution of per page L2 miss rates for each region in *mgrid*.

memory reference behaviour. Regions 3 and 5 from *hydro2d* (see Appendices A and B) also show some variance as well, though not as severe as in these two cases.

Examining the source code for these applications shows that although each application declares a single large array with which we have associated a single region, this array is divided into several distinct sections that are each accessed in a different manner. It is likely this division that has caused the disjoint clusters of pages found in the histograms. The cause of this phenomenon is our method of region creation, which was based on identifying regions through array declarations. Despite this non-uniform access behaviour, treating these arrays as single regions appears to have caused no significant problems for our later experiments. However, this might not be true in all cases where non-uniform behaviour is observed. For example, the region in *wave* is identified as sequentially faulted and placed in remote memory. This does not pose a problem because although the cache miss rates to this region vary from page to page, the range over which these rates vary is still at or below 3%. If a significant portion of those pages had a very high cache miss rate, then placing the entire region in remote memory could have a detrimental effect on performance compared to a policy that allowed some of the region to be placed in remote memory, and some of the region to be placed in local memory.

5.4 Single Application Workloads

In this section we present the results of experiments that compare the performance of applications under cache aware placement against their performance under first-touch placement in single application workloads. As we explained in Chapter 3, although single application workloads are not commonly found in multiprocessor server environments, examining the effects that page placement has on performance in these workloads allows us to better understand the effects of placement in more complicated multiprogrammed workloads. Additionally, the simplicity of the single application environment allows us

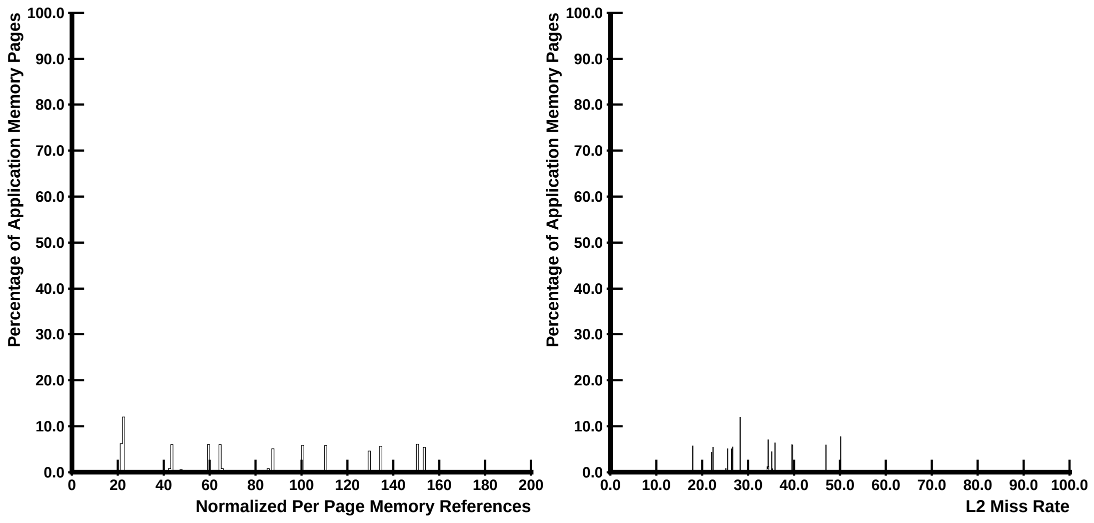


Figure 5.8: The histogram distribution of per page memory counts (left) and L2 miss rates (right) for the single region in *apsi*.

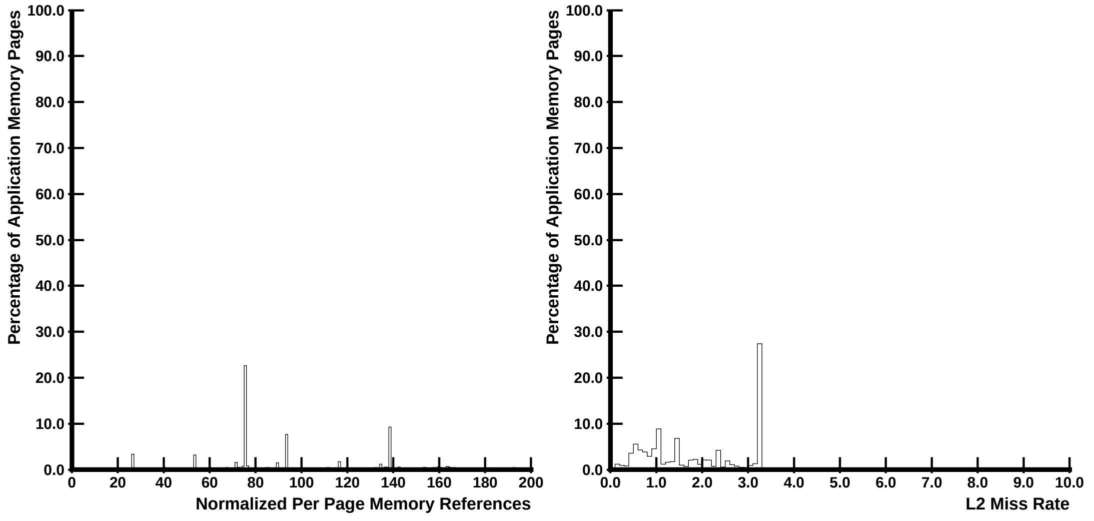


Figure 5.9: The histogram distribution of per page memory counts (left) and L2 miss rates (right) for the single region in *wave*.

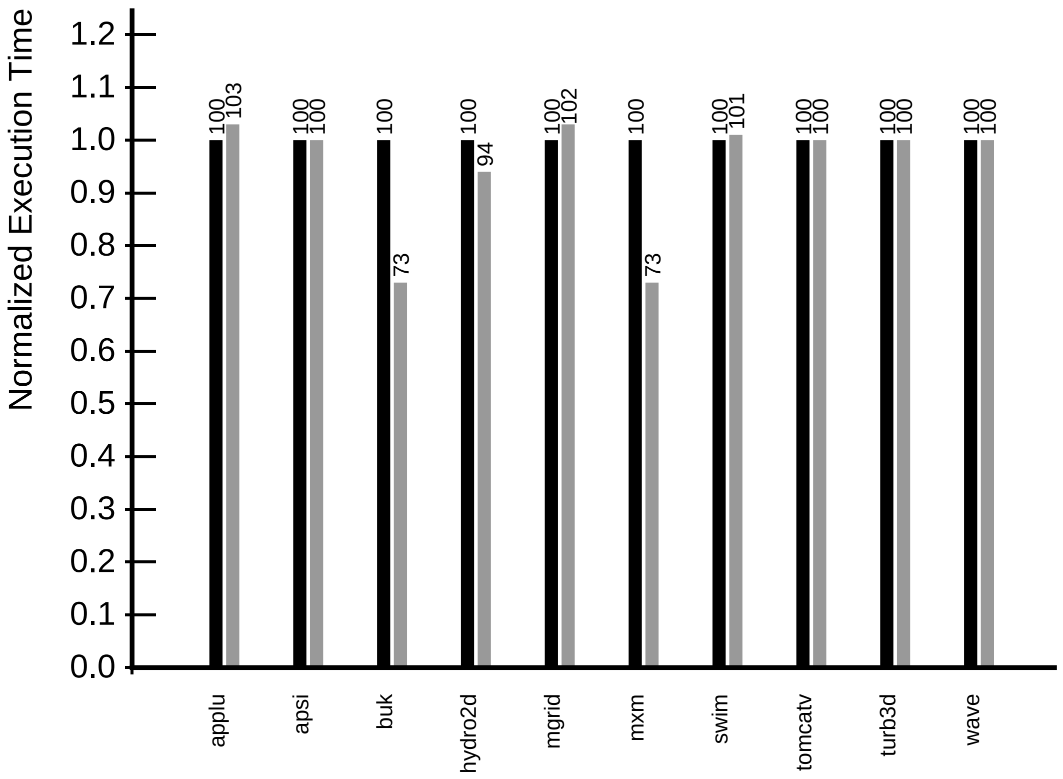


Figure 5.10: Execution times for each application running under first-touch placement (black) and cache aware placement (grey), where each application was run in isolation under each policy on the NUMAchine hardware. Times for each application are normalized to the execution time under first-touch placement.

to more easily verify the effects of our placement policy.

In this set of experiments, we ran each of the applications in our test suite individually on the NUMAchine hardware. Each application was executed under both first-touch placement, and cache aware placement, using identical input data sets for each of these trials.

The results of our experiments are shown in Figure 5.10. For each application, we show two bars representing normalized execution time for the application. The left bar shows the average execution time for each application under the first-touch policy. The

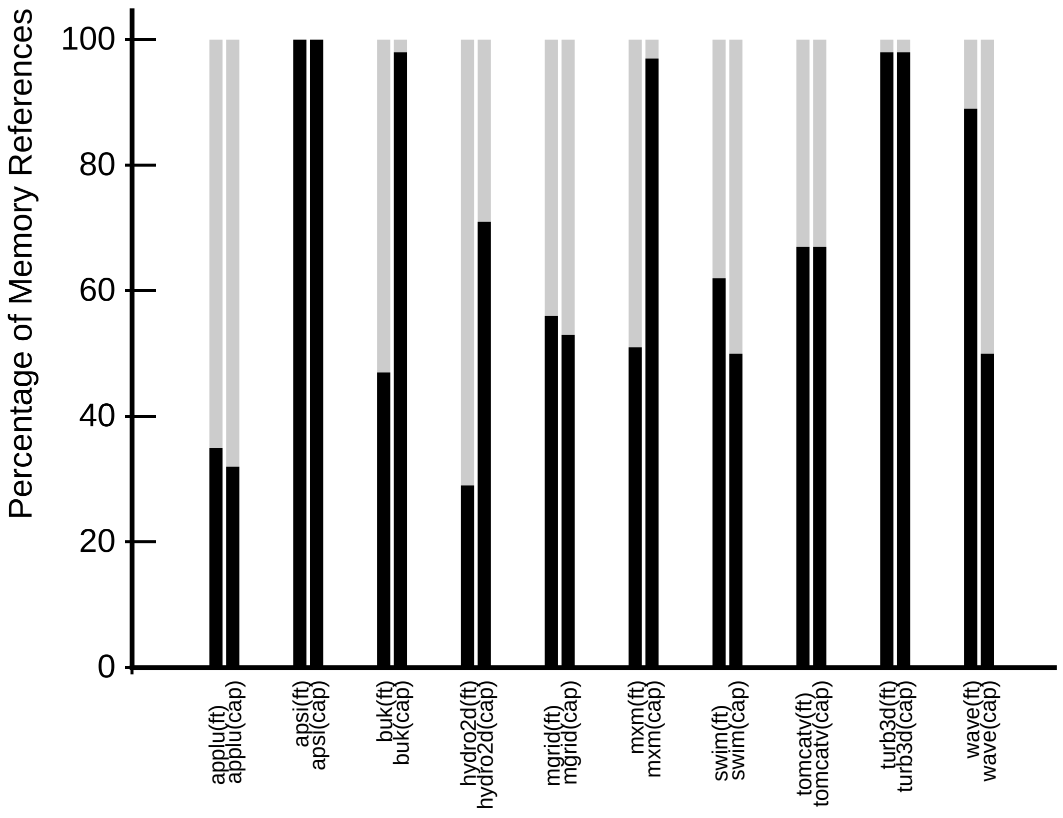


Figure 5.11: The breakdown of memory references by location using cache aware placement. Local memory references are shown by the black portion of each bar, while remote memory references are shown by the gray portion.

right bar shows the average execution time for each application under our new cache aware placement policy. Both bars are normalized to the original performance under first-touch placement, i.e., the left bars are all equal to 100. As we can see, this policy results in significantly improved performance for *buk*, *hydro2d*, and *mxm*, with improvements in execution time of up to 27%. The remaining applications (*applu*, *apsi*, *mgrid*, *swim*, *tomcatv*, *turb3d*, and *wave* show no appreciable difference in execution time.

To gain an understanding as to why some applications showed improved execution times while others showed no change or increases in execution time, we used the NUMAchine simulator to gather further information with regard to the utilization of the memory hierarchy. Figure 5.11 shows the breakdown for each application of the total data memory references (i.e., all data references that missed in the L2 processor cache) divided between those going to local memory, and those going to remote memory. This information is provided for execution under both first-touch and cache aware placement.

To aid in our examination of these results, we refer to the hypothetical example shown in Figure 3.4 of Section 3.2.1. Recall that Figure 3.4(a) shows the first-touch placement for a hypothetical application with 3 regions accessed in the following order: *region 1*, *region 2*, *region 3*. Figure 3.4(b) shows the cache aware placement for the application if *region 1* has good cache behaviour, and *region 3* has poor cache behaviour, with *region 1* being placed in remote memory to allow *region 3* to fit in local memory. This results in a decrease in remote memory accesses over first-touch placement if *region 3* is accessed more often than *region 1*. Figure 3.4(c) shows the cache aware placement if *region 3* is predicted to have a low cache miss rate, with *region 3* being placed in remote memory. In this case, no performance improvement is expected because *region 3* was mostly already in remote memory under first-touch placement. In fact, there is a slight increase in remote memory accesses under cache aware placement because part of *region 3* was in local memory under first-touch placement, whereas it is entirely in remote memory under cache aware placement.

From the results in Figure 5.11 and our observation of which regions were categorized as having good or bad cache behaviour in each application, we can divide our application test suite into three groups. The first group consists of *buk*, *hydro2d*, and *mxm*, all of which experience an increase in the percentage of memory accesses going to local memory under cache aware placement. Both *buk* and *mxm* had approximately 50% of their memory references going to local memory under first-touch placement. This number increases to over 95% in both cases under cache aware placement, with a corresponding significant improvement in execution time. Similarly, *hydro2d* experiences an increase from 29% to 71% of memory references going to local memory, resulting in a modest improvement in performance. These applications correspond to the hypothetical application shown in Figure 3.4(b) of Section 3.2.1.

The remaining applications show very minor changes in the percentage of memory references going to local memory when cache aware placement is used. These applications can also be divided into two groups. The first of these groups includes *apsi*, *tomcatv*, and *turb3d*, and consists of those applications whose ordering of regions is similar to the hypothetical application described in Figure 3.4(c). As described in Section 3.2.1, these types of applications have some regions that are identified as having good cache behaviour and some with poor cache behaviour. However, those regions that have good cache behaviour are the first ones accessed by their respective applications, and as such, are already allocated in local memory under first-touch placement. *Apsi* is a special case of this type of application as it allocates a single region that is identified as having poor cache behaviour and is placed locally by both first-touch placement and cache aware placement. For *tomcatv* and *turb3d*, those regions identified as having good cache behaviour and placed remotely under cache aware placement were also placed in remote memory under first-touch placement by virtue of being initially accessed last, i.e., they correspond to *region 3* in Figure 3.4(c). Thus, the page placements under first-touch placement and cache aware placement are identical, as are the numbers of references that go to local memory and remote memory, and the execution time under both policies.

The remaining applications (*applu*, *mgrid*, *swim*, and *wave*) show slight-to-moderate decreases in the percentage of memory references going to local memory under cache aware placement. These applications are distinguished from the other applications in our test suite by the fact that almost all of their regions are marked for remote allocation by cache aware placement (of the four, only *applu* does not have all of its regions marked as having good cache behaviour, with 6 out of 7 possible regions being marked so). In all cases, the policy imposed limit of 50% on the amount of memory placed on a remote node is triggered. As stated previously in Section 3.2.1, applications of this type can experience an increase in remote memory references under our placement policy, depending mainly on how many additional pages are placed in remote memory compared to first-touch placement. For example, in the case of *wave*, the input data size was such that almost all of the user allocated memory could fit in local memory under first-touch placement. Since only half of this memory was placed in local memory under cache aware placement, approximately half of the memory accesses become remote. Conversely, almost half of the data for *mgrid* was already in remote memory under first-touch placement, making the difference in local memory accesses between first-touch and cache aware placement very slight.

Although the second and third groups of applications showed a lack of improvement under cache aware placement, we show in the next section that applications of the second group can show improvement in the more common multiprogrammed environment. Applications of the third group have the least potential for improvement in a multiprogrammed environment, but can still benefit other applications by offloading some of their memory needs to remote memory nodes, alleviating the pressure on the local memory node.

5.5 Multiprogrammed Workloads

Having examined the performance and behaviour of cache aware placement in a single application environment, we now turn our attention to multiprogrammed workloads more commonly found on medium- and large-scale multiprocessor systems. As we described in the example shown in Figure 3.3, our cache aware placement policy can improve performance in a multiprogrammed environment where applications come into direct competition for local memory resources by virtue of being scheduled onto the same processing node.

Generally, existing schedulers for these types of environments are concerned with the efficient utilization of CPU resources, with little more than rudimentary consideration for memory locality[51]. Given a hierarchical NUMA-type multiprocessor such as NUMAchine where two or more processors share the same local memory, failure to consider such complications as the availability of local memory can result in processes being placed in direct competition with each other for local memory resources. For example, a scheduler whose primary concern is to efficiently utilize CPU resources might not differentiate between a free processor on a node where all the other processors are also free, and a free processor on a node where some or all of the other processors are busy. In the latter case, scheduling a new process there will possibly create greater contention for the local memory resources at that node. With this in mind, this section presents experimental results showing how our placement policy can help in environments where there are multiple applications competing for the same local memory.

To illustrate the effectiveness of our placement policy in a multiprogrammed environment, we devised an experiment using all ten of the applications from our test suite. For each trial run in our experiment, we created two ordered lists composed of between 5 and 10 of these applications. The size of both lists was the same for each trial and was randomly generated, as was the ordering of each list, and applications were not constrained from appearing multiple times on either list. One list of applications was executed on a single processor as follows: the processor executed the first application on the list, recorded its execution time, then executed the next application, and so on until the list was done. The other list was executed on a second processor on the same station and in the same fashion. Execution times were kept only for those applications that were run while both processors were busy, i.e., if one processor finished its list before the other processor, the remaining applications on the second list were not executed (since the workload was no longer multiprogrammed). The beginning of the execution of each list was also slightly staggered to simulate one processor starting with more local memory than the other. Each set of lists was executed under first-touch placement, and then under cache aware placement, and enough random lists were generated so that between 5 and 10 execution times for each application under each policy were recorded.

Figure 5.12 shows the recorded execution times for the applications following several runs of this experiment. Each application in the graph has a total of six bars associated with it. These six bars are divided into two groups of three, the left group being associated with execution under our cache aware placement policy, and the right group associated with execution under first-touch placement. In each of these groups of three, the leftmost bar indicates the fastest execution time for all instances of the application in question during the experimental runs. The middle bar shows the average execution time of all instances. Finally, the rightmost bar indicates the slowest execution time out of all runs of the application. All of these bars represent normalized execution time with respect to the average execution time of each application under first-touch placement (the middle bar of the right side group in for each application).

As these results show, our new placement policy fared well in this multiprogrammed experiment. The six applications from the first two groups we identified in the previous section on single application workloads (*apsi*, *buk*, *hydro2d*, *mxm*, *tomcatv*, and *turb3d*) showed a decrease in average execution time ranging from 11% to 36%. Those applications belonging to the third group (*applu mgrid*, *swim*, *wave*) experienced more modest improvements (approximately 5%) or showed no noticeable change.

To ensure the statistical significance of these results, Table 5.4 shows the raw mean execution times for each application (in seconds) under each policy with an associated 95% confidence interval on the difference between those two means. These confidence intervals were computed using the t-test [27]. According to this test, the two mean execution times are statistically different with 95% confidence if the confidence interval for the difference between these two means does not include zero. Using this test, there is a statistically significant difference between first-touch and cache aware placement for six of the applications. These include five of the applications that we reported improvements on

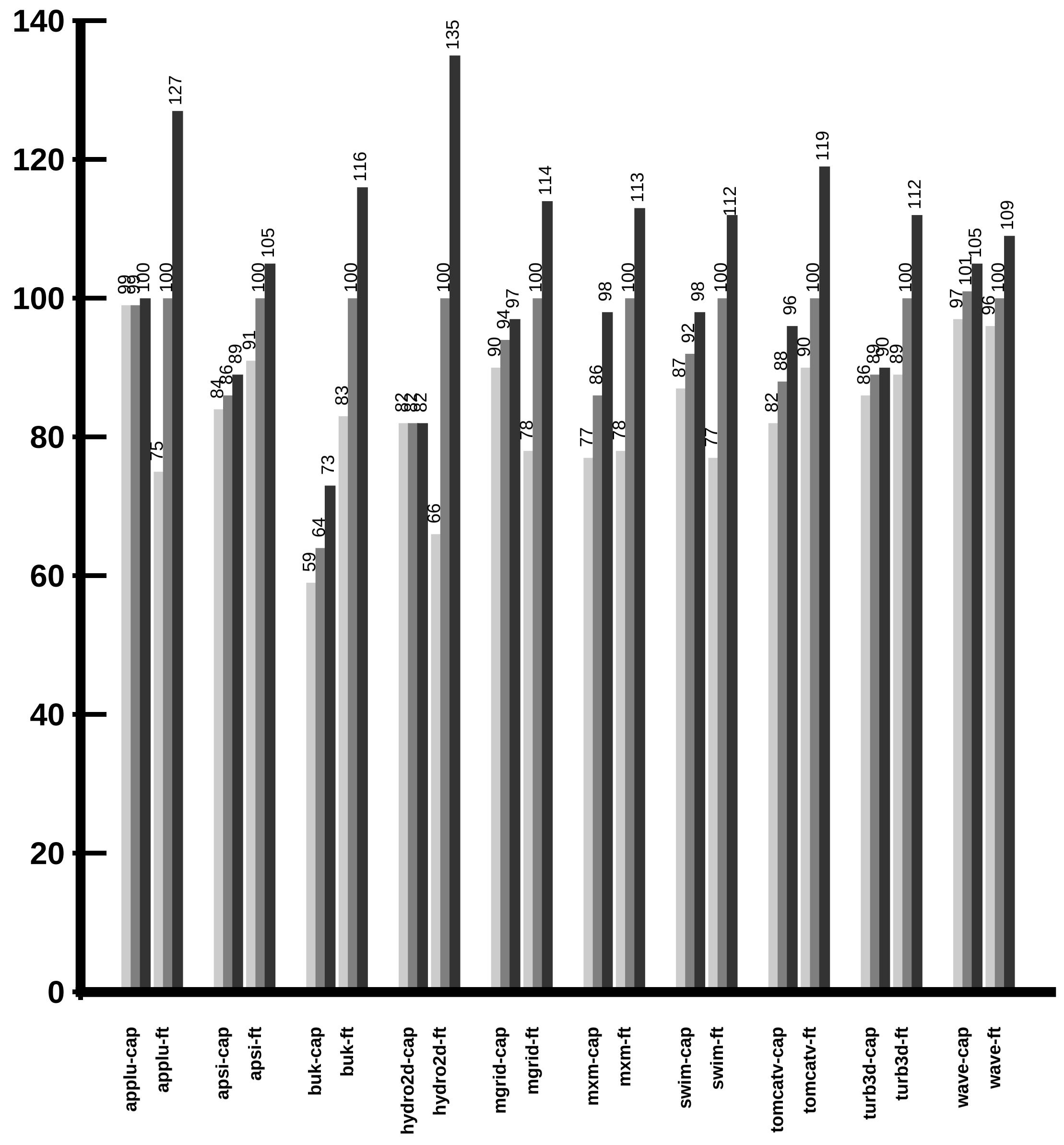


Figure 5.12: The results of our multiprogrammed experiment. The data for each application is shown in two sets of three bars, representing execution under cache aware placement (left side set) and first-touch placement. For each set, the three bars represent (from left to right): minimum execution time, average execution time, and maximum execution time. For each application, all times have been normalized to the average execution time under first-touch placement.

Application	FT mean	CAP mean	difference	95% CI	statistically different
applu	733	727	6	±72	no
apsi	1040	918	122	±19	yes
buk	832	534	298	±84	yes
hydro2d	789	646	143	±183	no
mgrid	1226	1158	68	±82	no
mxm	666	542	124	±36	yes
swim	259	239	20	±18	yes
tomcatv	522	465	57	±24	yes
turb3d	448	396	52	±17	yes
wave	2147	2166	-19	±28	no

Table 5.4: This table shows the 95% confidence interval for the difference in mean execution time between first-touch placement and cache aware placement for each application. The first two columns show the mean execution time under each policy in seconds. The third column shows the difference in these means. The fourth column shows the 95% confidence interval for the difference in the two means (calculated using the t-test). The means are statistically different with a 95the confidence interval does not include zero.

above (*apsi*, *buk*, *mxm*, *tomcatv*, and *turb3d*). A sixth application, *swim*, also passes this test, although by a slim margin. However, although the mean execution time for *hydro2d* under cache aware placement is 18% less than execution under first-touch placement, the high variance in the recorded first-touch placement times prevents us from declaring that the two means are statistically different with 95% confidence.

Although we were unable to run multiprogrammed workloads with our simulator, we can use some of the insights gained from the simulated results of our single application experiment in Section 5.4 to better understand our results. In that experiment, the second group of applications consisting of *apsi*, *tomcatv* and *turb3d*, showed no improvement under cache aware placement because their poorly cached regions are initially accessed before their well cached regions (such as in the hypothetical example in Figure 3.4(c)), meaning that these poorly cached regions get placed in local memory under first-touch placement.

However, in our multiprogrammed experiment, all three applications experienced performance improvements. As we discussed in Section 3.2.2, this opportunity for improvement arises from the greater competition for local memory present in a multiprogrammed environment. For example, as we discussed in the previous section, *apsi* has a single user region that is poorly cached and placed in local memory under cache aware placement. In a single application environment, first-touch placement can also place this region in local memory because there is no other competition for that memory. In a multiprogrammed environment, a situation such as is shown in Figure 3.5 can occur, with *apsi* taking the role of application B in the figure. As is shown in part (b) of that figure, cache aware placement can free some of the local memory so that the single *apsi* region can be placed in local memory, where as under first-touch placement, this might not be the case.

In addition to the improvement in performance under cache aware placement, we also observe that there appears to be less variance in the performance of each application under cache aware placement. In almost all runs using first-touch placement, a processor whose first running application was allocated a majority of the local memory pages tended to hold onto that local memory for the duration of execution of its entire list of applications. The only way for processes running on the other processor to gain access to this local memory was to begin execution at the same time that the process running on the other processor was ending. At this point, the ending process would free up its local memory pages, allowing the newly starting process to allocate them. Since this happened infrequently, the most common scenario was that an application would perform very well if it were started on the processor that initially lay claim to the most local memory, but would do extremely poorly if it were started on the other processor since most of its memory would be allocated on a remote node.

In the cache aware case, this extreme variance in the amount of local memory available to a newly started process did not occur. Since most applications would offload some of their pages to remote memory and not dominate the majority of local memory frames, a newly starting process would typically find that there was always some local memory available for its use. This resulted in the more even sharing of memory between the two processors under cache aware placement, and less variance in the performance of applications under this policy.

5.5.1 Statistical Approximation of Memory in Use

As described in Section 4.5.3, our placement policy needs to have an idea of the memory usage at each node in the system when deciding which node to allocate the pages of a region to. Although we used the number of allocated pages to measure usage at each node in our experiments, such an approach was only possible due to the absence of page replacement in the system. In this section, we present the results of an experiment that show the effectiveness of using a statistics-based approach for describing memory usage in an environment that includes page replacement. This method, described in Section 4.5.3, approximates the memory usage by unmapping a set number of random memory pages in a node at set intervals, and extrapolating the fraction of these pages that are remapped during that interval to approximate the fraction of total memory pages at the node that are in use.

In this experiment, we wrote a benchmark program that reads and writes user configurable amounts of memory. This *working set* of memory in use was varied over the execution of the program to see how accurately the statistical approximation of usage could track the actual usage. The graph in Figure 5.13 shows the results of a run of this test program. The solid line represents the actual working set as a fraction of total memory allocated. The dotted line shows the approximation made by the sampling method. The sampling method unmapped a total of 1% of the allocated pages during each sampling period of 100 ms. As we can see, the approximation of memory in use tracks very closely to the actually memory in use, and follows changes to the working set very rapidly. These results indicate that a statistical-based approach to describing memory usage could be viable in a system employing page replacement.

5.5.2 Summary

Cache aware placement assumes that memory access behaviour to regions is relatively uniform to justify making cache predictions and placement decisions on a region granularity. We have shown that such uniformity exists in the vast majority of regions in our application suite. We have also demonstrated that cache aware placement is effective in reducing remote memory accesses, and thus, application execution times in both single and multiprogrammed workloads. In the latter case, even some applications that do not experience improvement when run in isolation can benefit from cache aware placement. In the multiprogrammed case, applications in our workloads experience improvements of up to 35%.

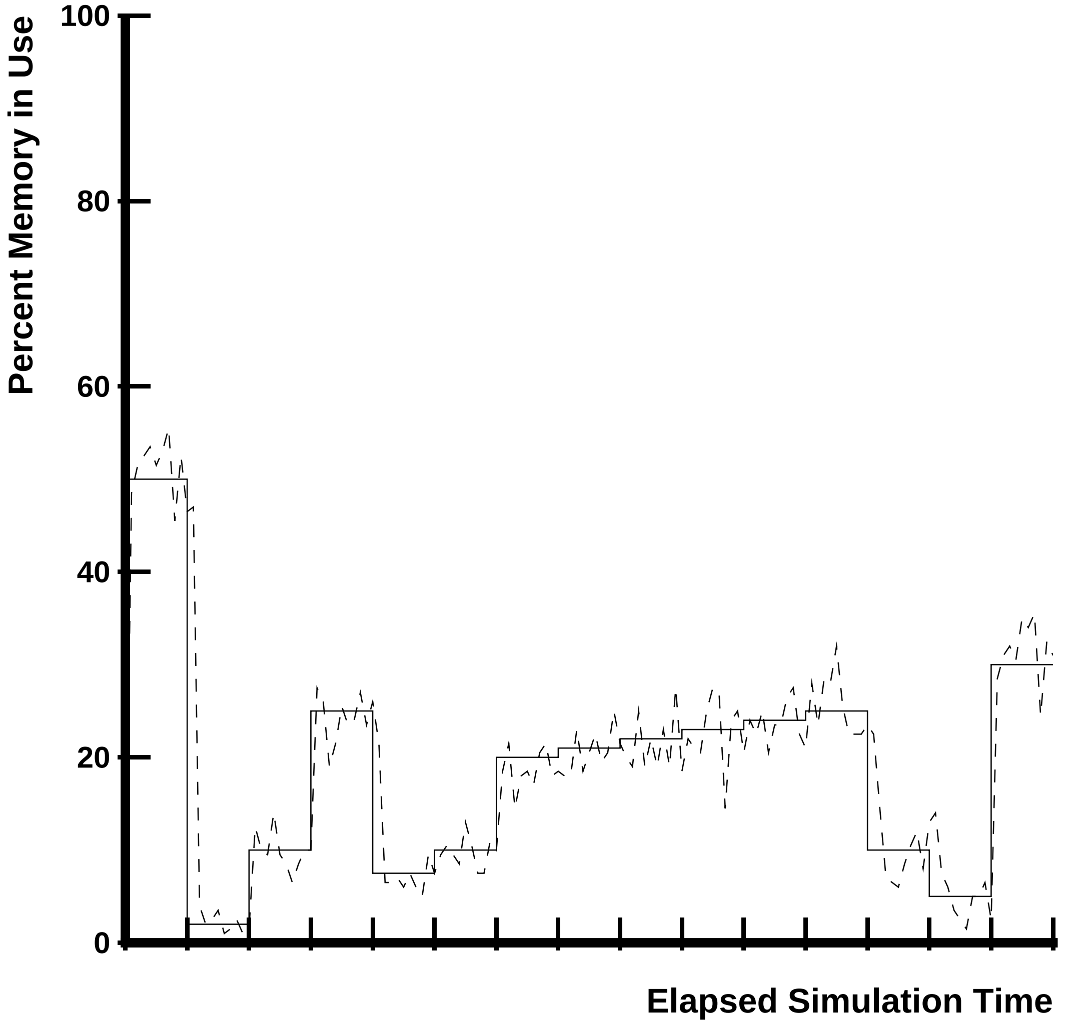


Figure 5.13: Approximating the memory in use by statistical sampling. The solid line represents the actual memory in use, while the dashed line is the approximation made by the sampling method.

Chapter 6

Conclusions

The goal of this dissertation has been to describe the design of a new page placement policy for CC-NUMA multiprocessors called *cache aware placement*. It has also been our goal to analyze and compare the performance of well known benchmark applications under cache aware placement against performance under *first-touch placement*, the existing standard for these types of systems.

The need for a new placement policy has arisen due to the recent trend toward using CC-NUMA multiprocessors as centrally managed, general purpose compute servers, rather than in their traditional role as highly specialized compute platforms running specially written multithreaded applications. Existing memory management policies like first-touch placement were developed to improve the performance of these specialized applications when run in isolation, with the memory needs of each thread not exceeding the amount of local memory available. Such an environment is not commonly found when these systems are used as general purpose compute servers.

The shortcoming of first-touch placement in situations where an application must allocate some pages in remote memory (due to insufficient free frames at the local node) is that local memory frames are allocated to page faults on a first come first served basis. We have argued that this ordering can sometimes result in highly accessed pages being placed in remote memory resulting in a larger percentage of time spent waiting for memory accesses to complete and slower execution times. This effect can be seen in our comparison of first-touch placement against a placement policy that uses a priori knowledge of memory accesses to minimize the number of remote accesses by an application. For some applications, we showed that the difference in execution time between using first-touch placement and a best case placement can be as much as 30%.

Cache aware placement is a new placement policy that makes more efficient use of local memory than first-touch placement by prioritizing the allocation of local memory frames to regions of pages that are predicted to have high cache miss rates. Additionally, to reduce pressure on local memory, cache aware placement places those pages that are predicted to have low cache miss rates in remote memory. The effectiveness of this policy relies on an accurate method of predicting the future cache behaviour of regions of memory. Our prediction method divides user allocated memory into contiguous virtual address segments called *regions*. The access patterns and cache behaviour for each region are inferred from their initial page fault ordering, and we have presented results that demonstrate a correlation between low cache miss rates and sequential page faults.

The use of cache aware placement with our suite of benchmark applications has led to substantial performance improvements for both single program and multiprogrammed workloads. In the single application case, we have experienced execution time reductions of between 6% and 27% for three of the ten applications in our test suite, with the remaining seven applications showing no significant change in performance. For multiprogrammed workloads, we found that cache aware placement improved the performance of some of the applications that exhibited no benefit in a single application environment, due to the higher demand for local memory in this environment. Under multiprogrammed conditions, our experiments showed that cache aware placement can achieve reductions in execution time ranging from 11% to 36% for six of the ten applications in our test suite.

6.1 Future Work

Two types of applications that we did not examine in this dissertation are multithreaded applications and pointer-based integer applications, both of which have characteristics that affect the application of cache aware placement. Multithreaded applications add the complication of shared memory between threads. Faced with shared pages by multiple threads on different processors, any static placement policy such as first-touch or cache aware placement will encounter the same limitation: there no longer exists a single "local" node where a page can be placed to eliminate remote accesses. Such a limitation can be addressed by use of a dynamic policy like page migration to move pages between sharers, which we have previously suggested could compliment a static placement policy such as ours.

However, a drawback to using migration to deal with multithreaded applications is that it negates one of our goals for cache aware placement: that no specialized hardware or compiler support is used. One possible avenue of research along these lines might be to examine whether the need for hardware monitoring of cache misses in a migration system could be obviated by our cache prediction mechanism. As discussed in Chapter 2, some migration systems rely on hardware monitoring of cache misses to trigger migration, since older schemes, such as triggering migration on TLB misses, do not reliably identify hot pages. One possible approach to eliminating the need for hardware monitoring might be to trigger a migration decision upon a TLB miss, and allow only those pages that are predicted to have poor cache behaviour to be migrated.

Unlike the scientific applications we have examined, most integer applications rely on pointer-based data structures rather than arrays, making the identification of regions more difficult. This may make some form of compiler analysis or user supplied hints necessary (in fact, compiler analysis may only become tractable with user supplied information given how difficult compiler analysis of pointer-based applications can be). The dynamic nature of memory allocation in these applications may also require a reevaluation of the definition of a region, since the pages of a dynamically allocated data structure are not likely to form a contiguous segment of the virtual address space. This has implications for how we correlate low and high miss rate regions with page fault ordering, calling into question whether a dichotomy of sequential and non-sequential page fault orderings makes sense for pointer-based data structures.

Another avenue of experimentation might be to incorporate cache aware placement into an environment that also utilizes a dynamic placement policy like page migration. In particular, it would be interesting to examine how often the page migration system is called upon to migrate "incorrectly" placed pages, or conversely, how many migrations are eliminated by "correctly" placing pages at allocation time using cache aware placement. Such an experiment might be especially interesting for environments where processes or threads allocate more memory than is available at a processing node, as most migration systems do not allow migration to a node that has no available frames (i.e., there is no mechanism to migrate pages "away" to make room in local memory). Cache aware placement could compliment page migration in these situations by only placing the most heavily accessed pages in local memory, leaving room in the local node for the page migration system to decide which, if any, of the remotely placed pages need to be migrated there.

Finally, a more involved avenue of research might examine how to incorporate memory management concerns into the scheduler. For example, it may be beneficial to discourage the scheduler from running a process on a node with very low available memory. Managing the allocation of both memory and cpu resources in the scheduler could require great complexity. However, such a unified approach may ultimately be an effective way of dealing with locality issues.

Appendix A

User Region Memory Reference Distributions

The following graphs show the per page memory reference data for the user regions in all test applications referred to in Section 5.3.

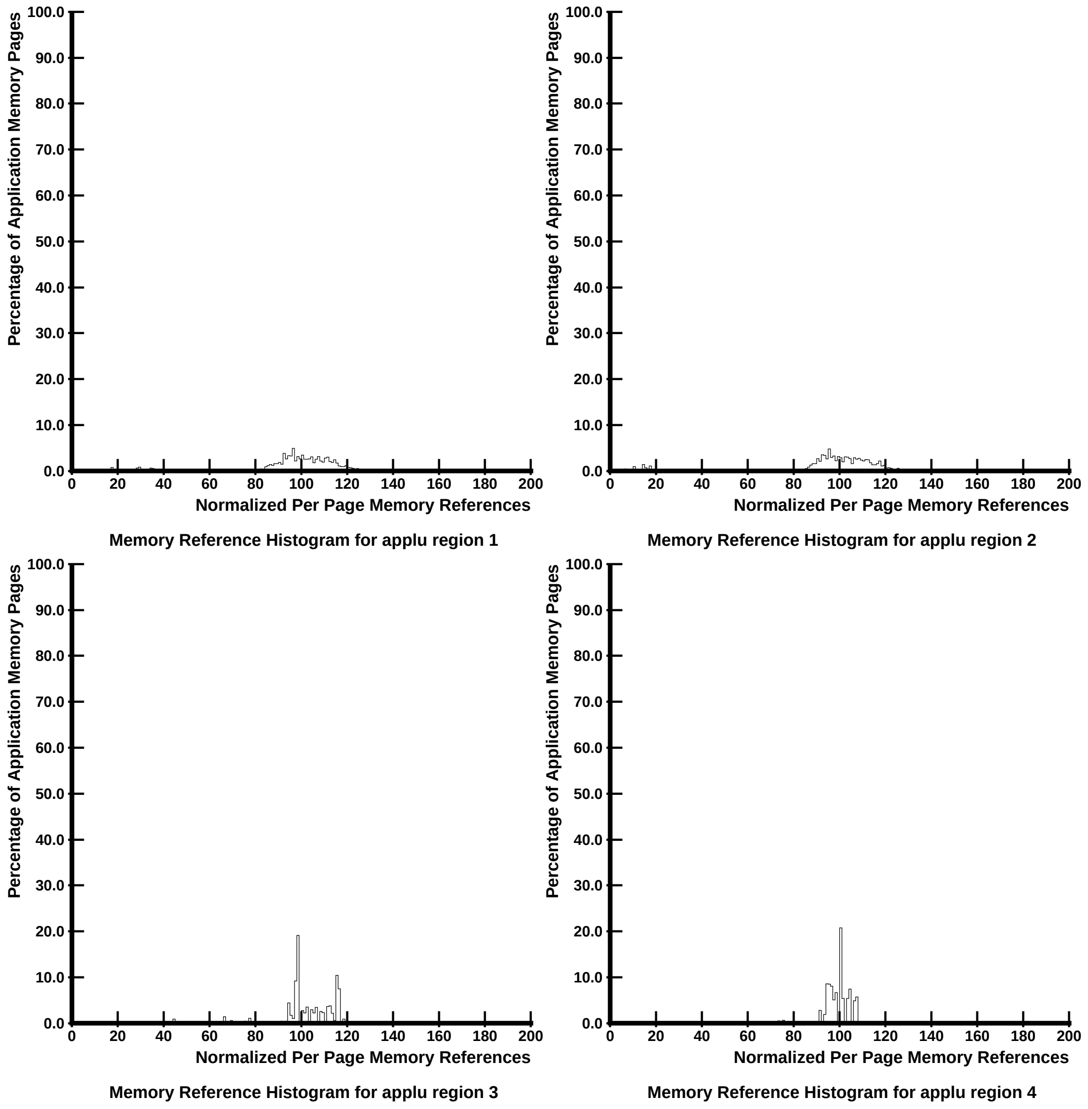
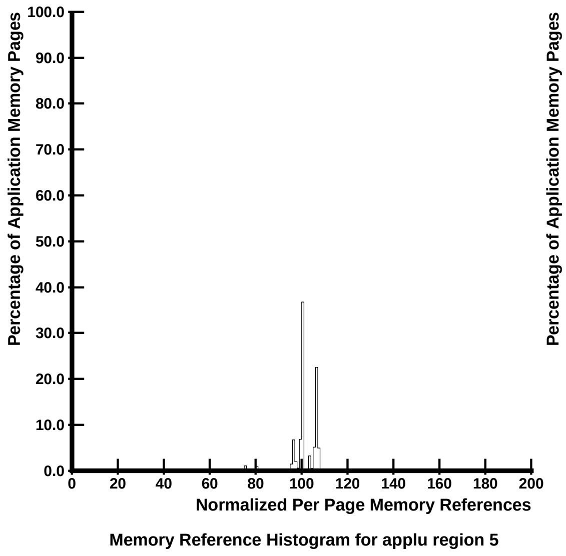
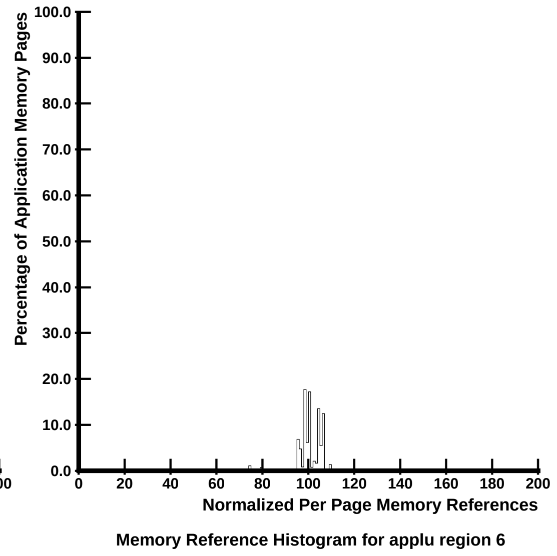


Figure A.1: Distribution of per page memory reference counts over all pages for applu regions 1-4. Regions 2, 3, and 4 are sequential.





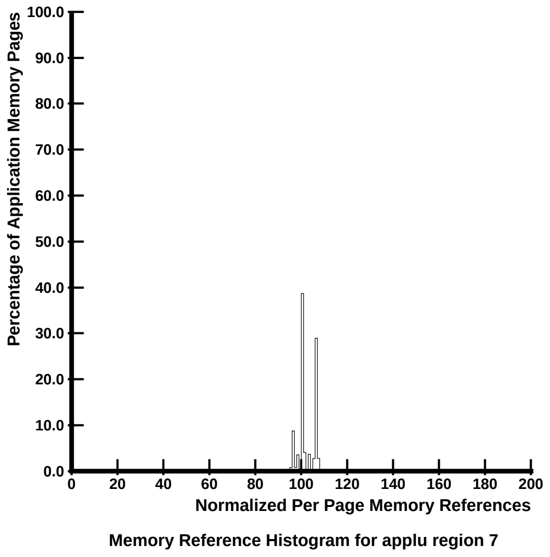


Figure A.2: Distribution of per page memory reference counts over all pages for applu regions 5-7. Regions 5, 6, and 7 are sequential.

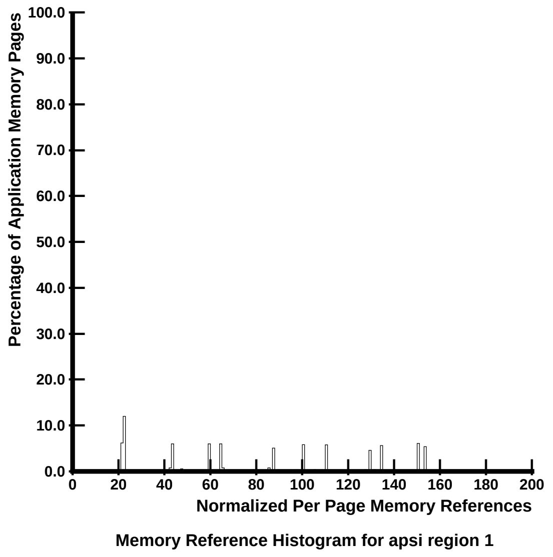


Figure A.3: Distribution of per page memory reference counts over all pages for apsi region 1. Region 1 is non-sequential.

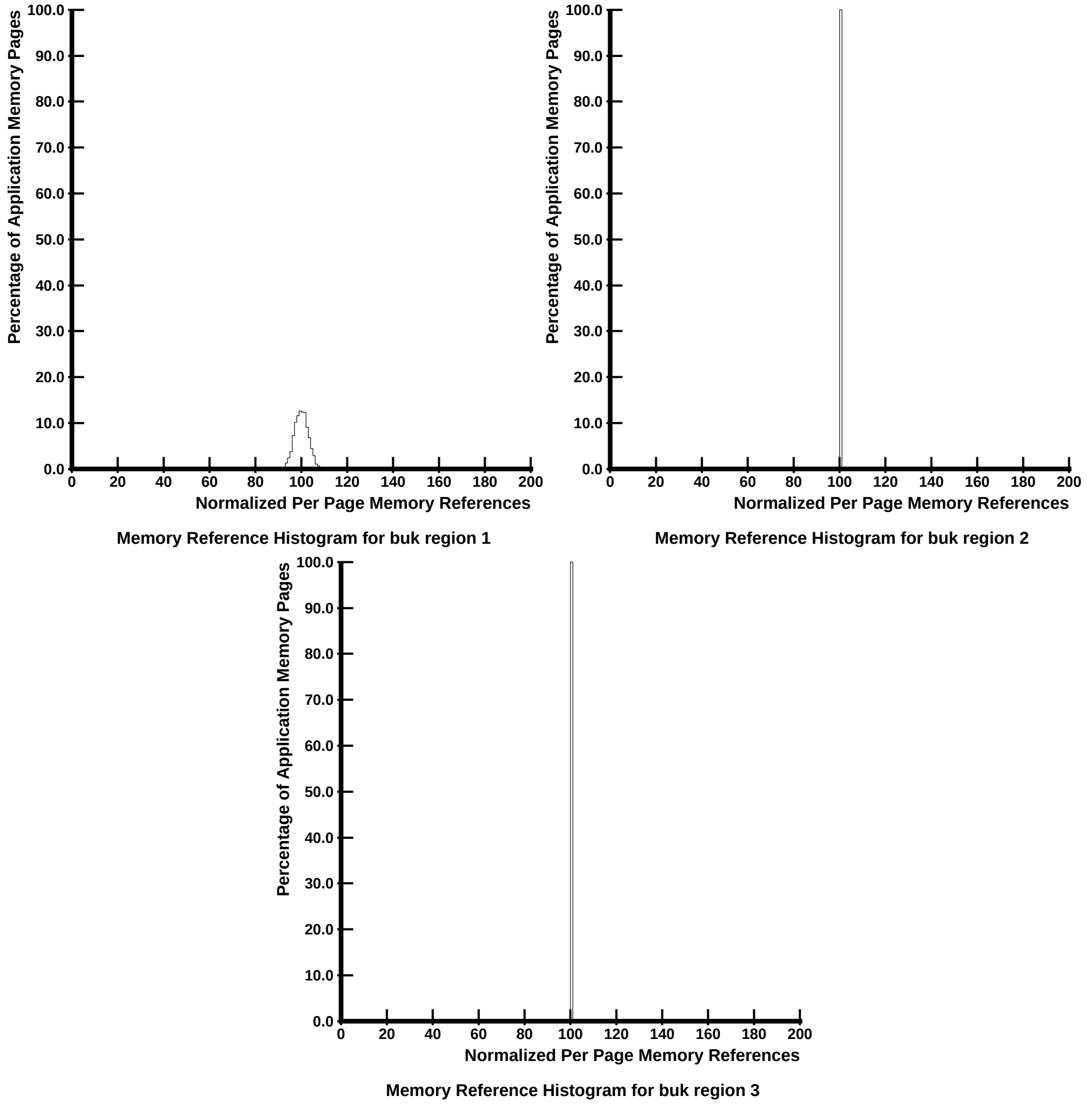


Figure A.4: Distribution of per page memory reference counts over all pages for buk regions 1-3. Regions 2 and 3 are sequential.

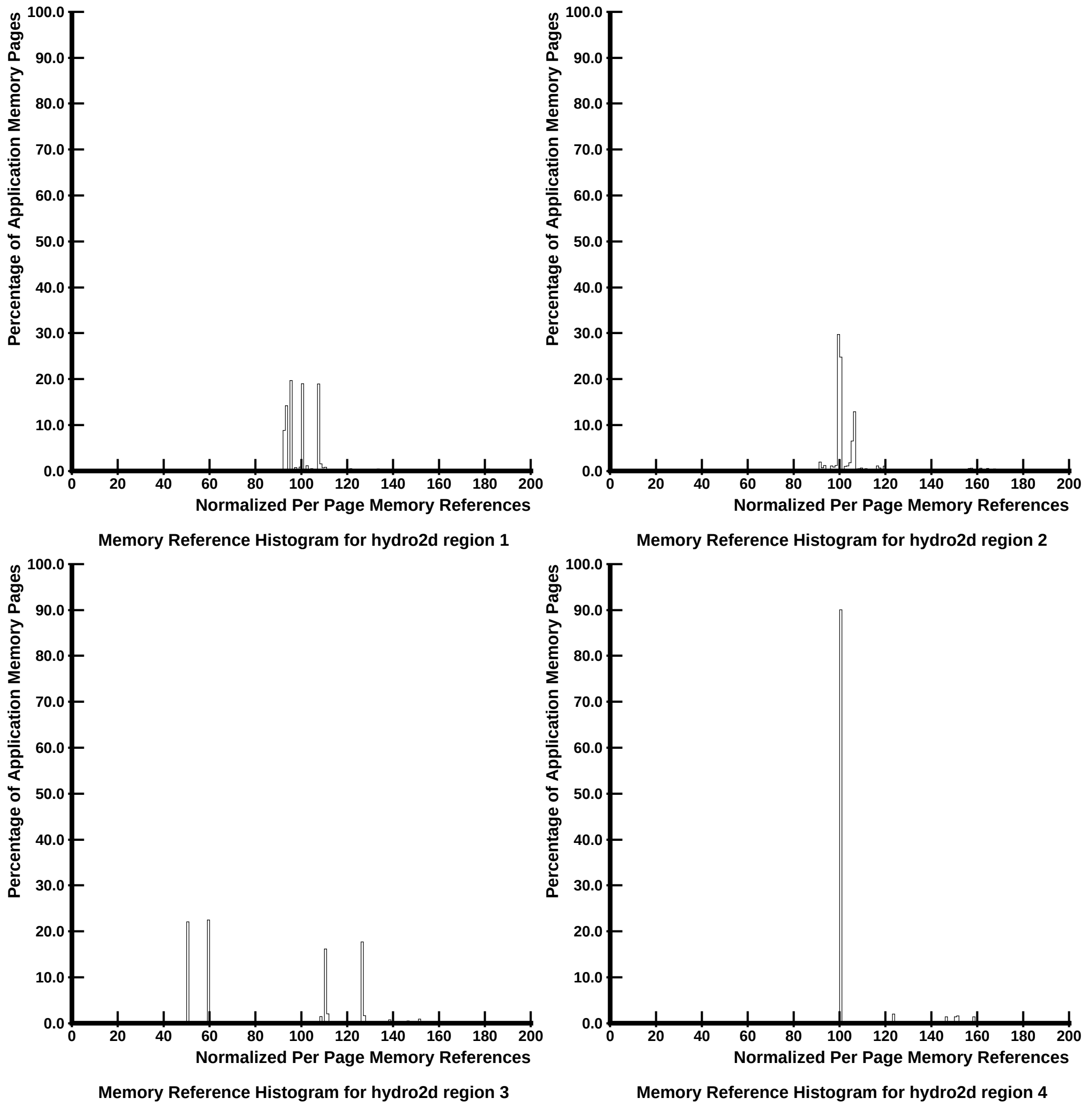


Figure A.5: Distribution of per page memory reference counts over all pages for hydro2d regions 1-4. Regions 1, 2, and 4 are sequential.

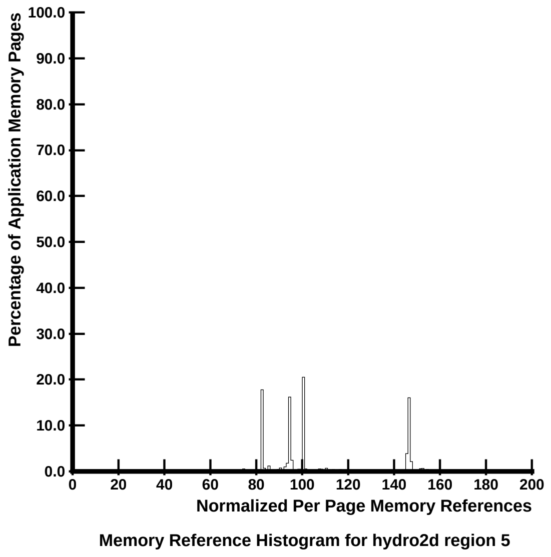
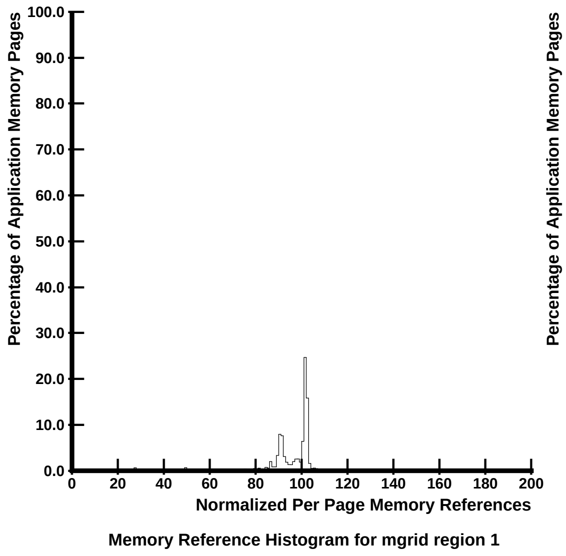
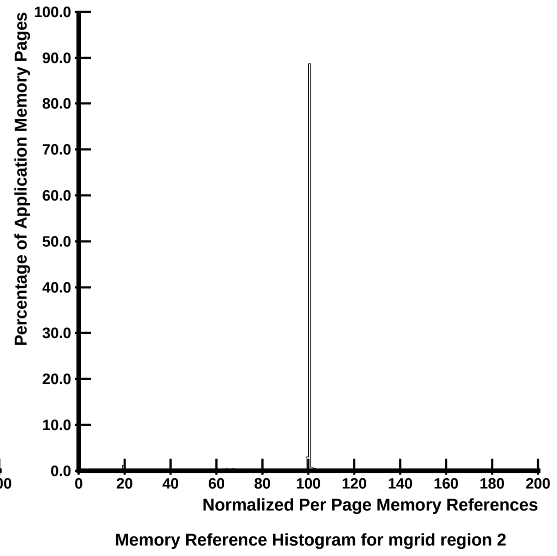


Figure A.6: Distribution of per page memory reference counts over all pages for hydro2d regions 5. Region 5 is non-sequential.





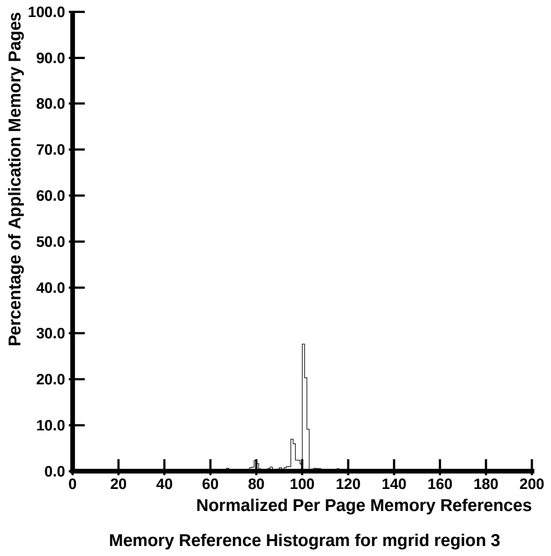


Figure A.7: Distribution of per page memory reference counts over all pages for mgrid regions 1-3. Regions 1, 2, and 3 are sequential.

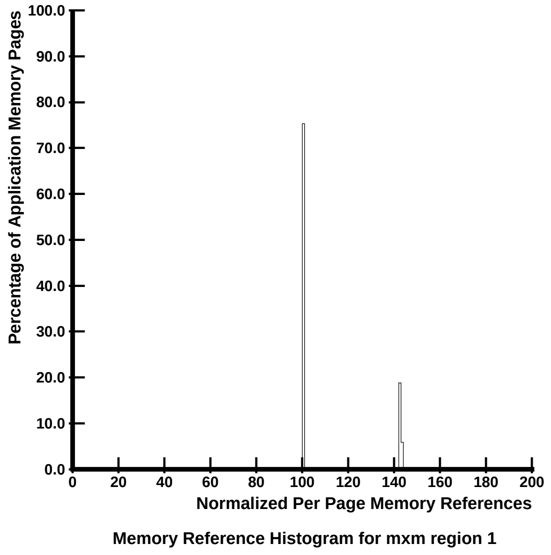


Figure A.8: Distribution of per page memory reference counts over all pages for mxm region 1. Region 1 is sequential. Regions 2 and 3 not shown (see explanation of simulator limitation in Section 4.3.

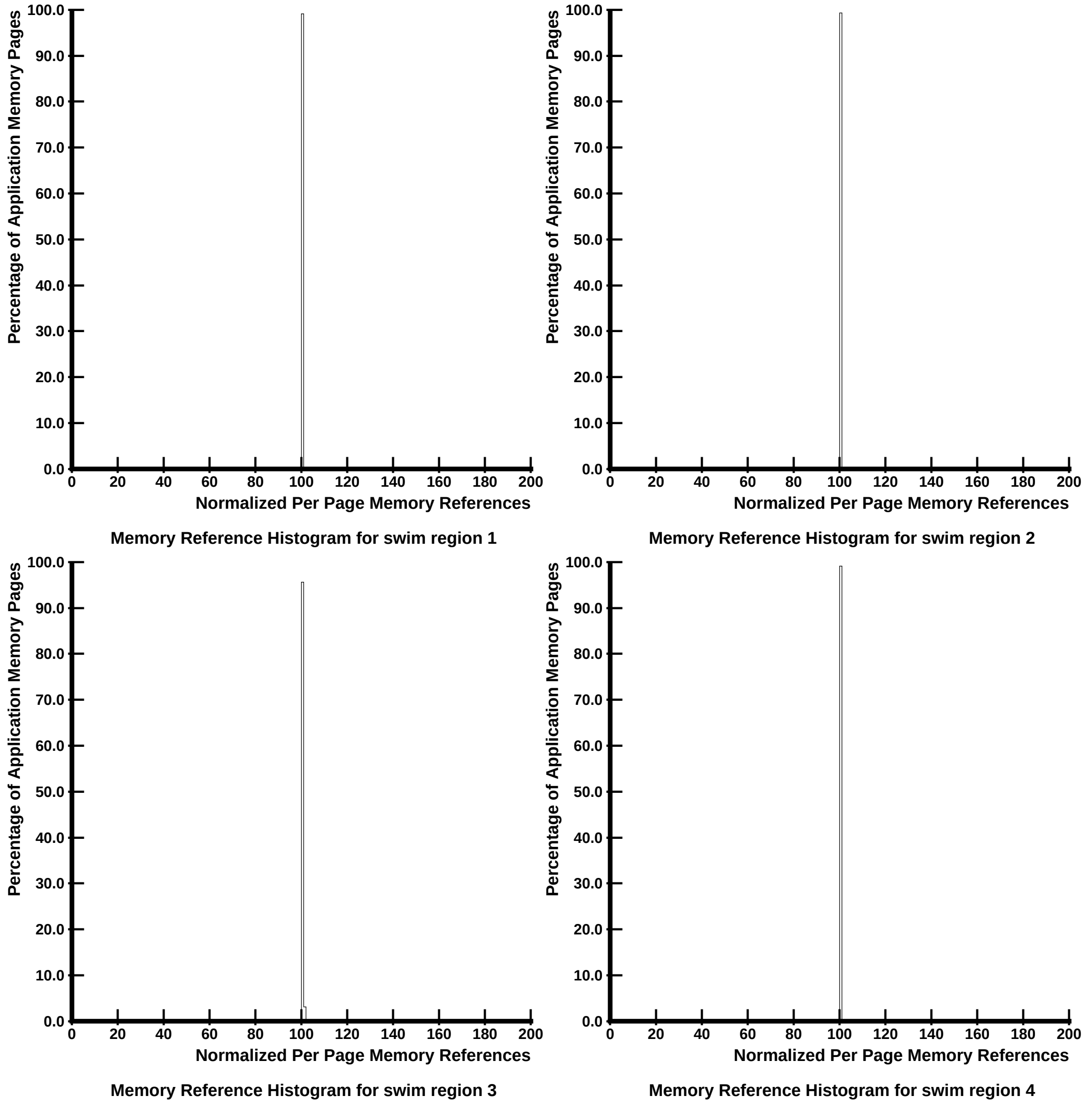


Figure A.9: Distribution of per page memory reference counts over all pages for swim regions 1-4. Regions 1, 2, 3, and 4 are sequential.

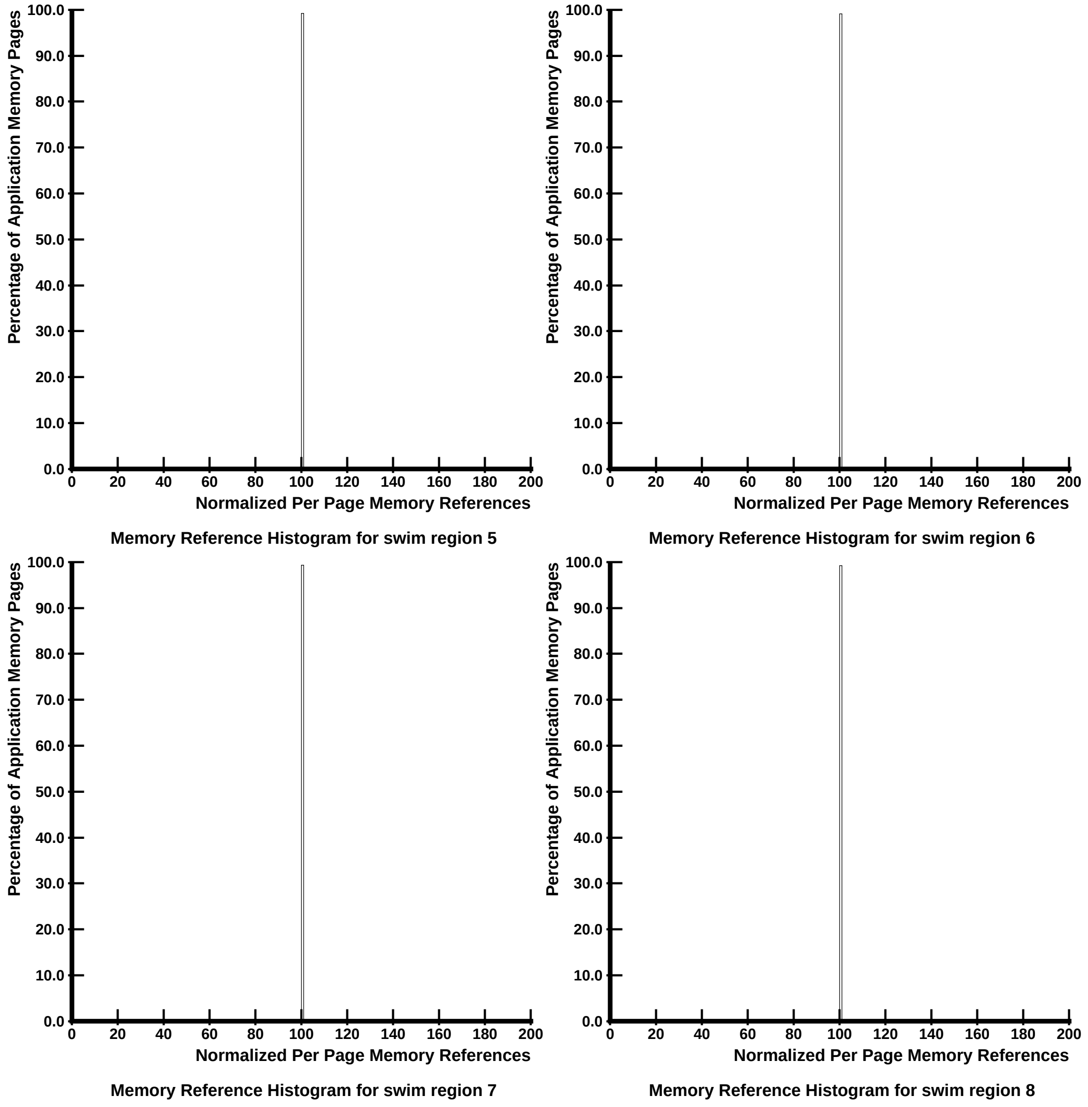


Figure A.10: Distribution of per page memory reference counts over all pages for swim regions 5-8. Regions 5, 6, 7, and 8 are sequential.

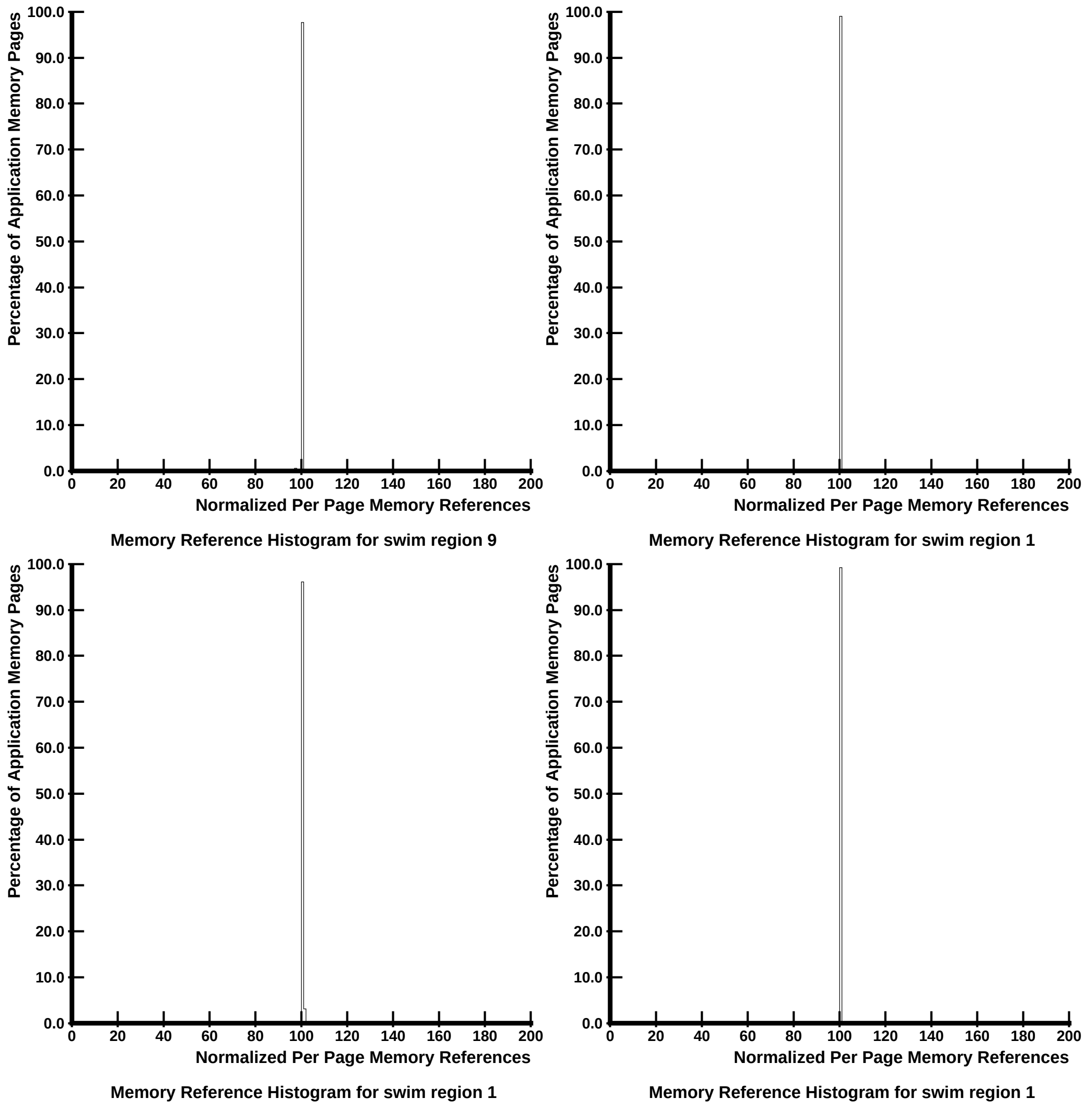
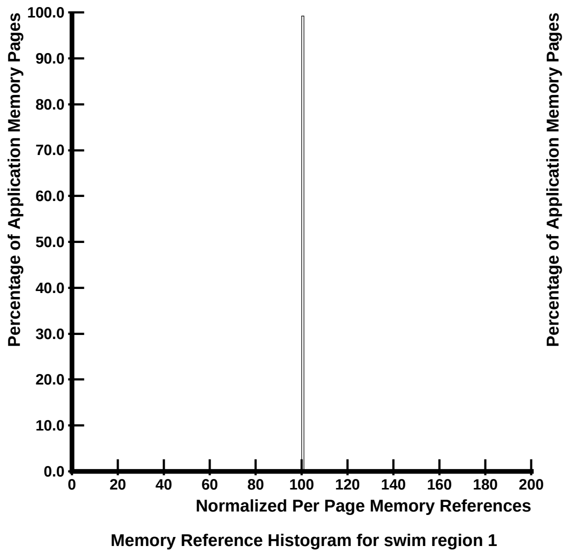


Figure A.11: Distribution of per page memory reference counts over all pages for swim regions 9-12. Regions 9, 10, 11, and 12 are sequential.



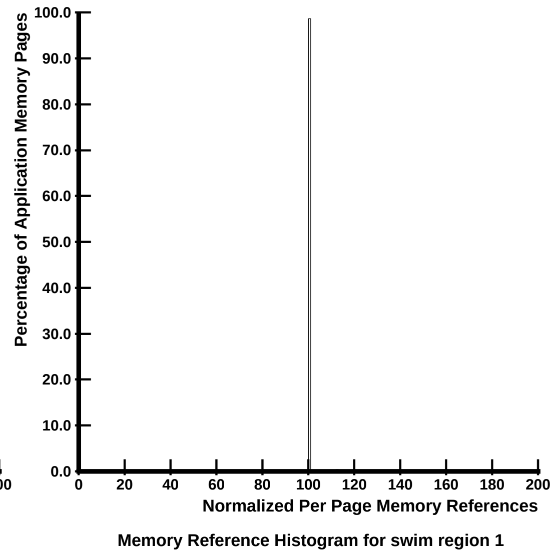


Figure A.12: Distribution of per page memory reference counts over all pages for swim regions 13-14. Regions 13 and 14 are sequential.

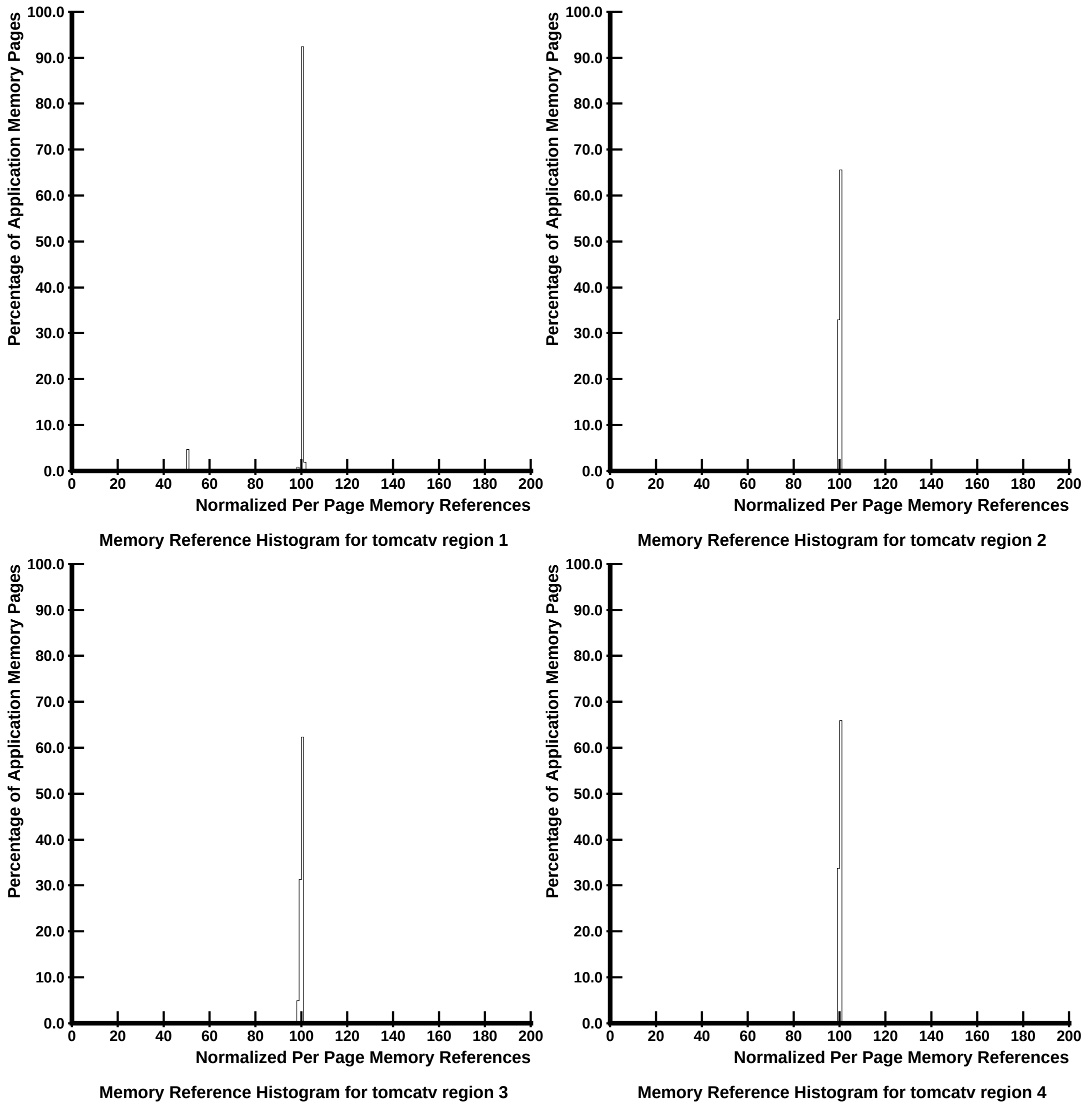
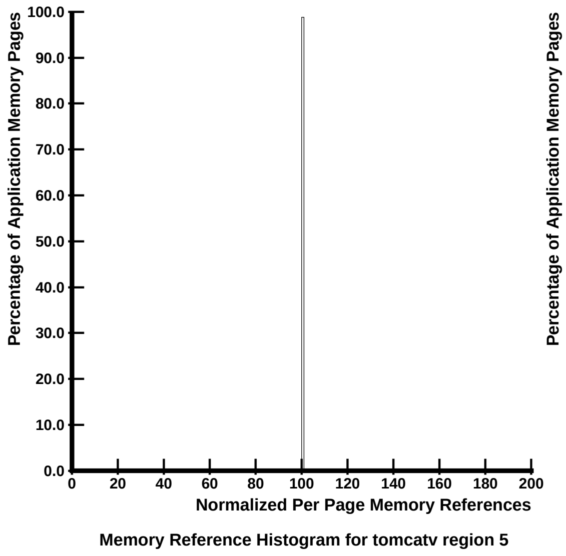
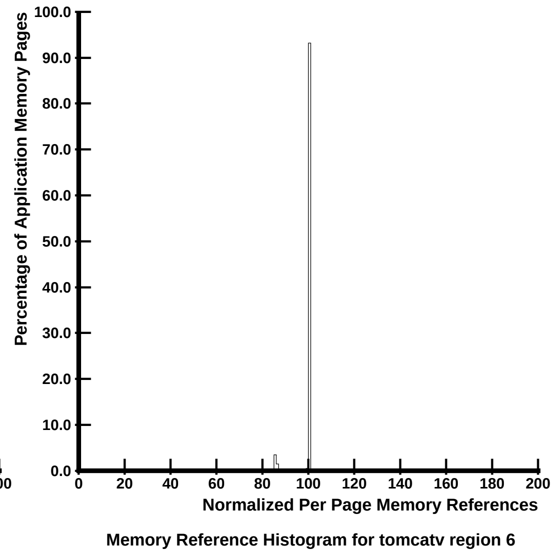


Figure A.13: Distribution of per page memory reference counts over all pages for tomcatv regions 1-4. Regions 1 and 4 are sequential.





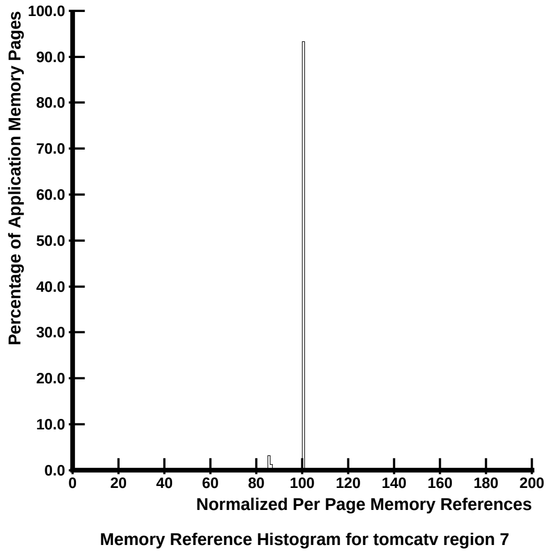


Figure A.14: Distribution of per page memory reference counts over all pages for tomcatv regions 5-7. Regions 5, 6, and 7 are sequential.

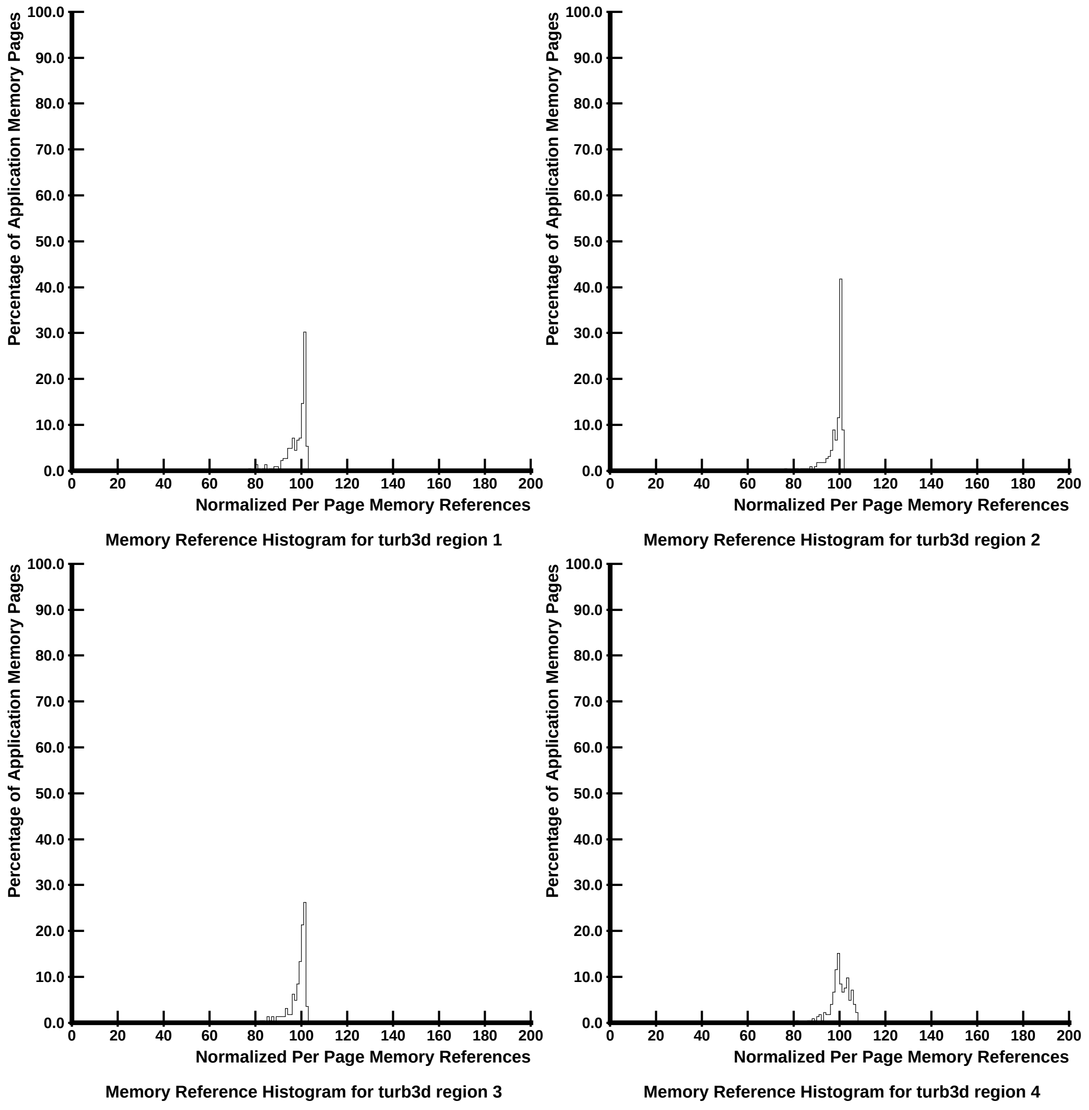
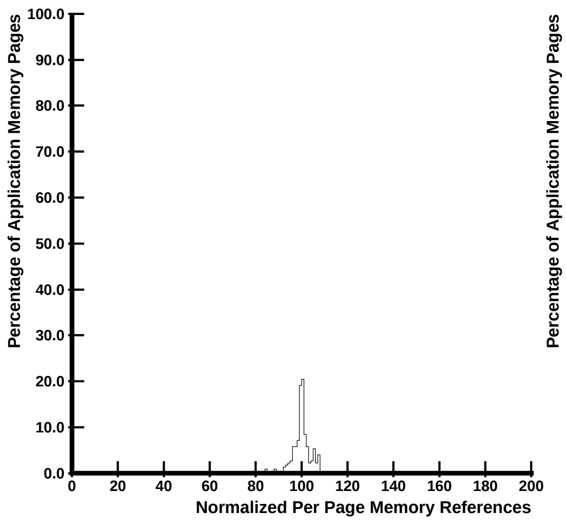
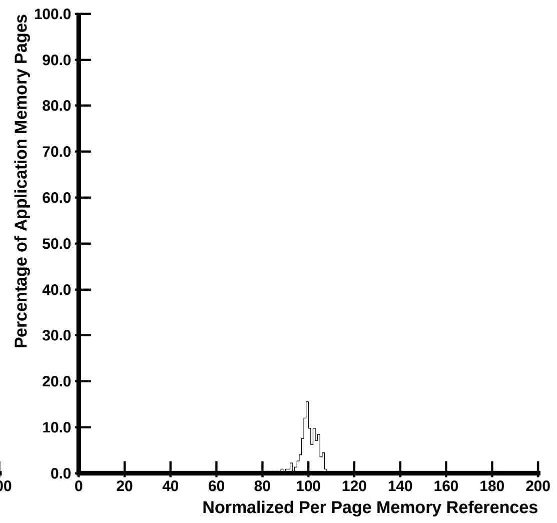


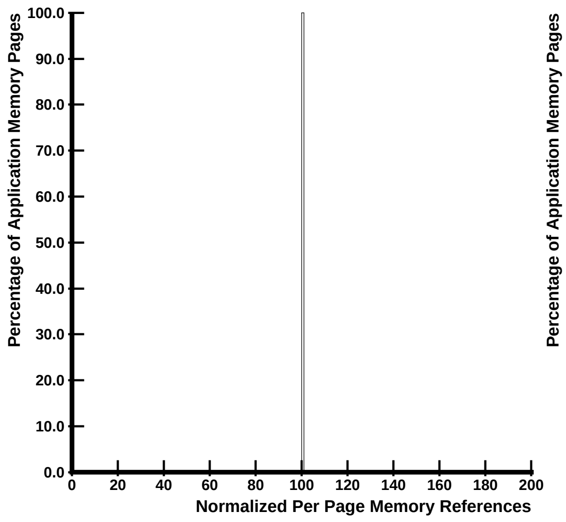
Figure A.15: Distribution of per page memory reference counts over all pages for turb3d regions 1-4. No sequential regions.



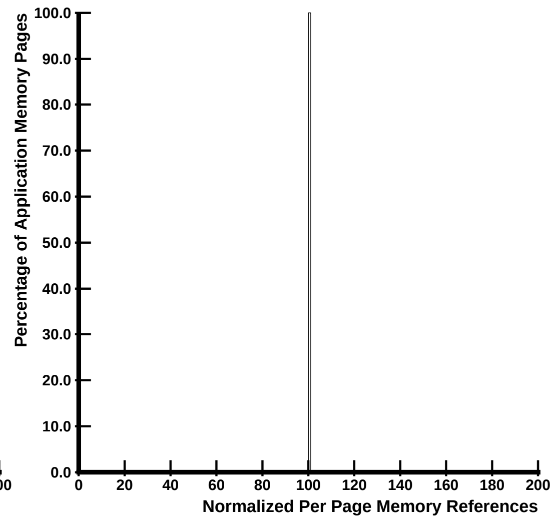
Memory Reference Histogram for turb3d region 5



Memory Reference Histogram for turb3d region 6

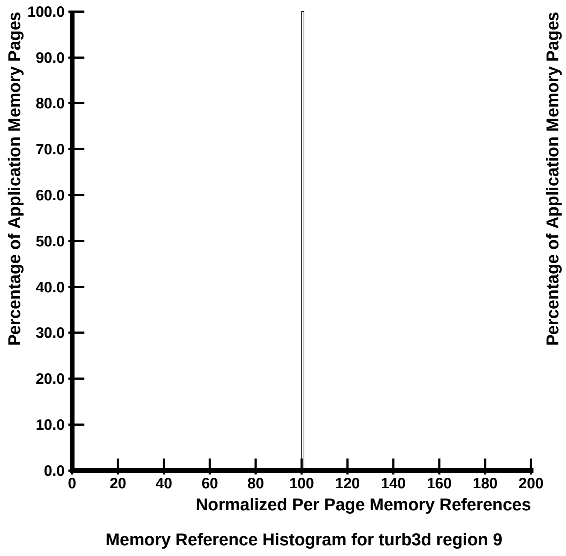


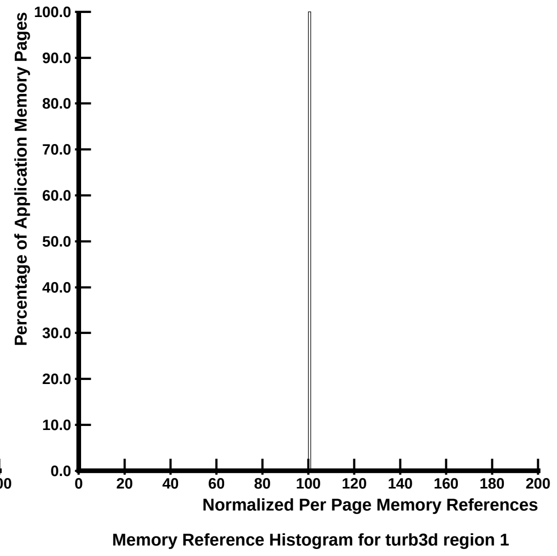
Memory Reference Histogram for turb3d region 7

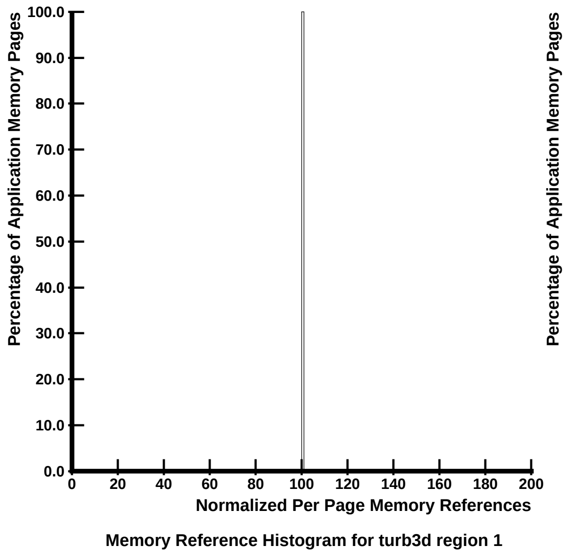


Memory Reference Histogram for turb3d region 8

Figure A.16: Distribution of per page memory reference counts over all pages for turb3d regions 5-8. Regions 7 and 8 are sequential.







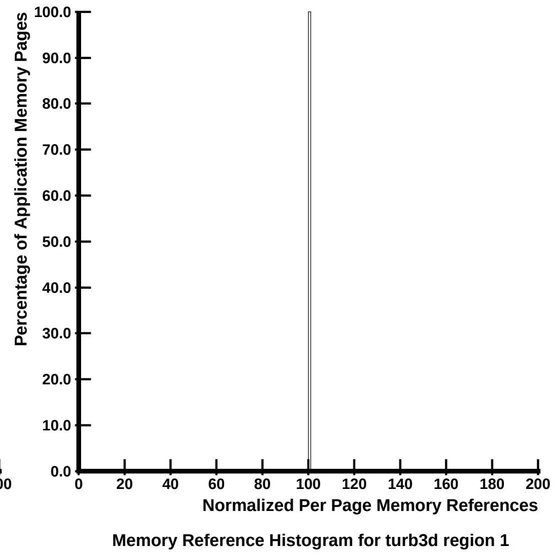


Figure A.17: Distribution of per page memory reference counts over all pages for turb3d regions 9-12. Regions 9, 10, 11, and 12 are sequential.

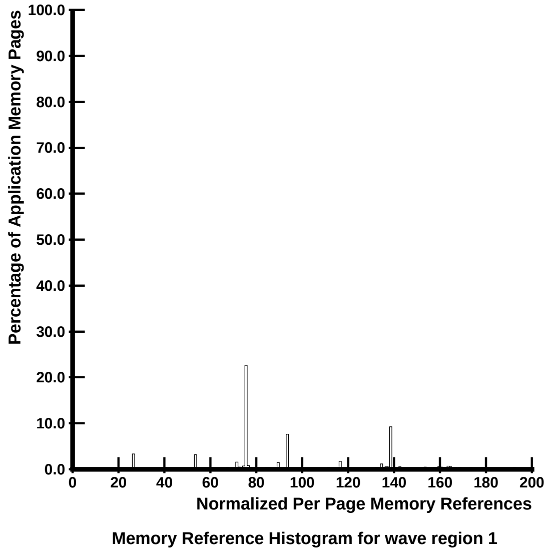


Figure A.18: Distribution of per page memory reference counts over all pages for wave region 1. Region 1 is sequential.

Appendix B

User Region Cache Miss Rate Distributions

The following graphs show the per page L2 cache miss rate data for the user regions in all test applications referred to in Section 5.3.

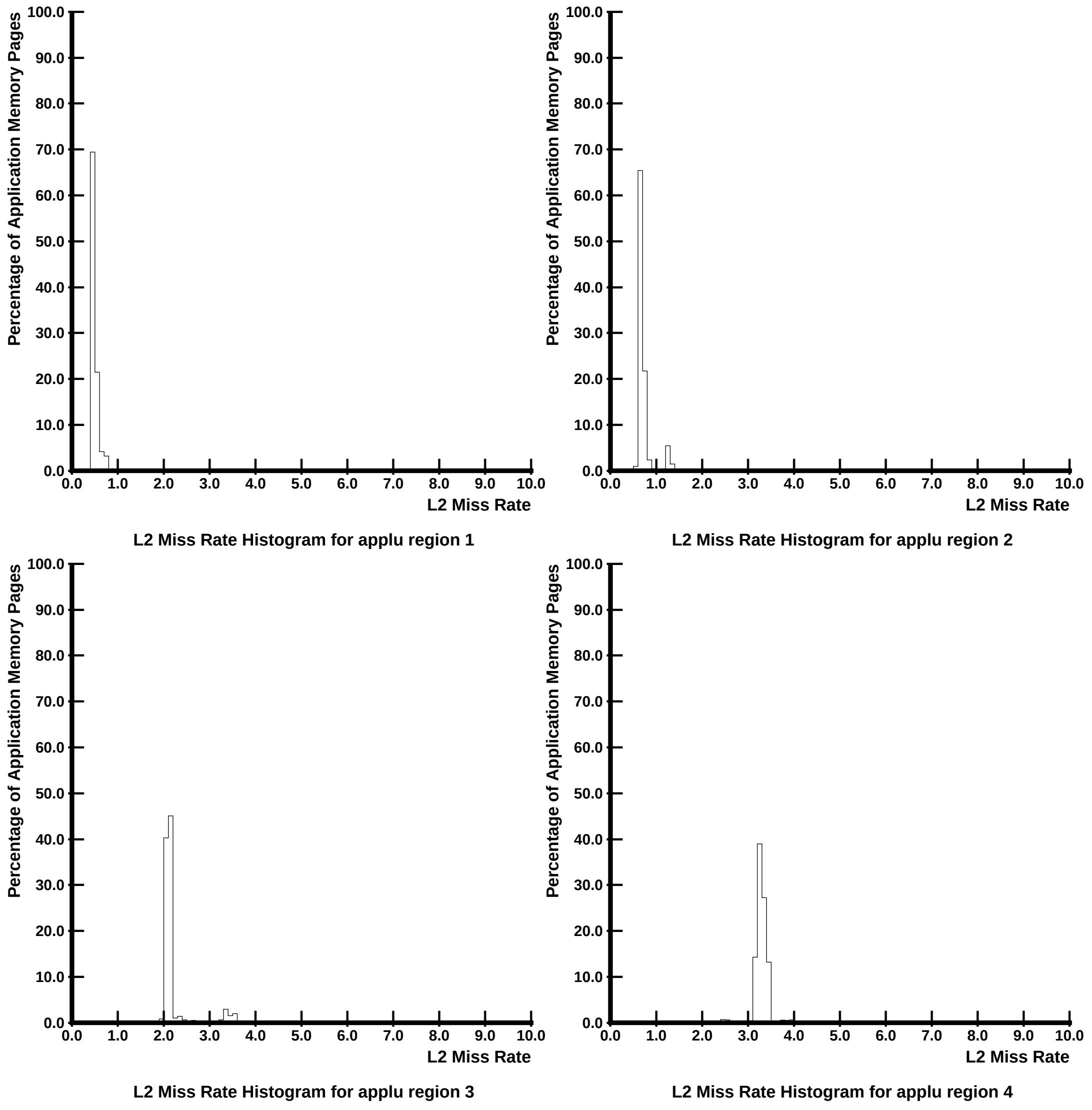
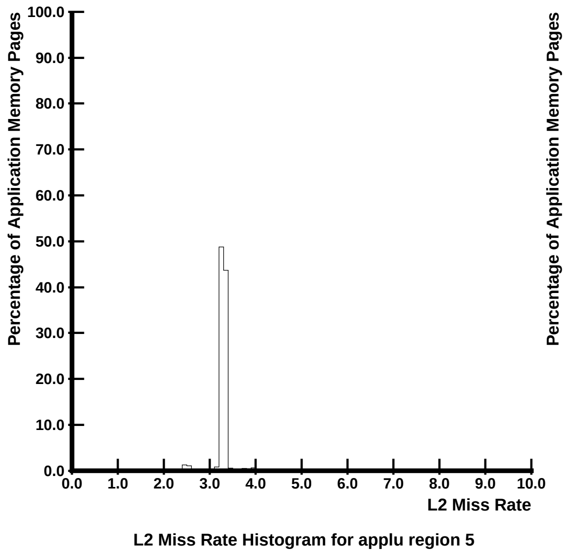
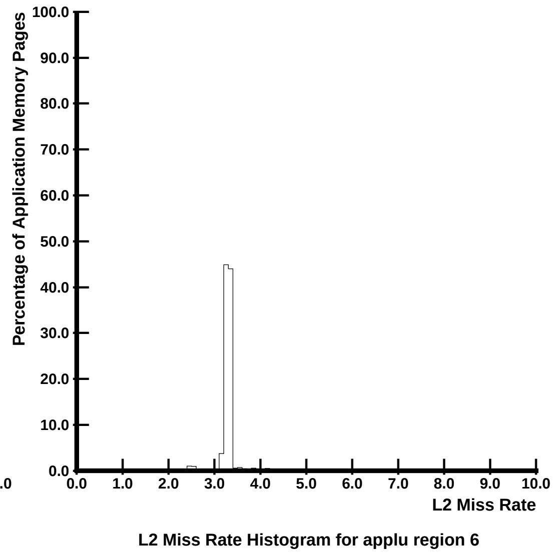


Figure B.1: Distribution of per page L2 miss rate over all pages for applu regions 1-4. Regions 2, 3, and 4 are sequential.





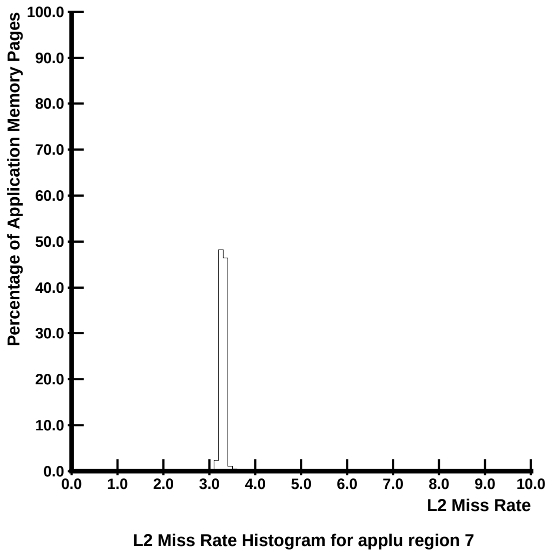


Figure B.2: Distribution of per page L2 miss rate over all pages for applu regions 5-7. Regions 5, 6, and 7 are sequential.

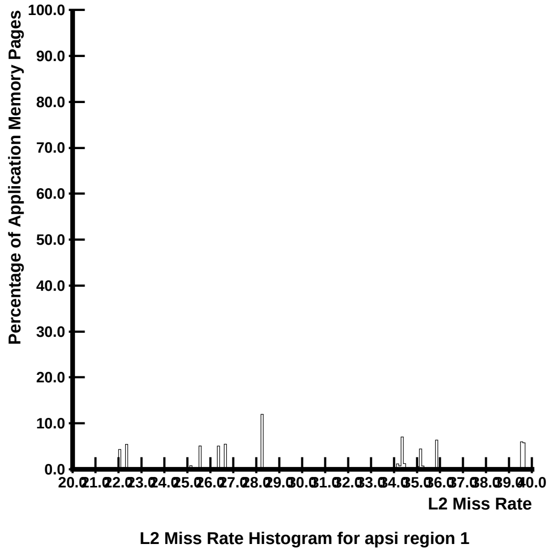


Figure B.3: Distribution of per page L2 miss rate over all pages for apsi region 1. Region 1 is non-sequential.

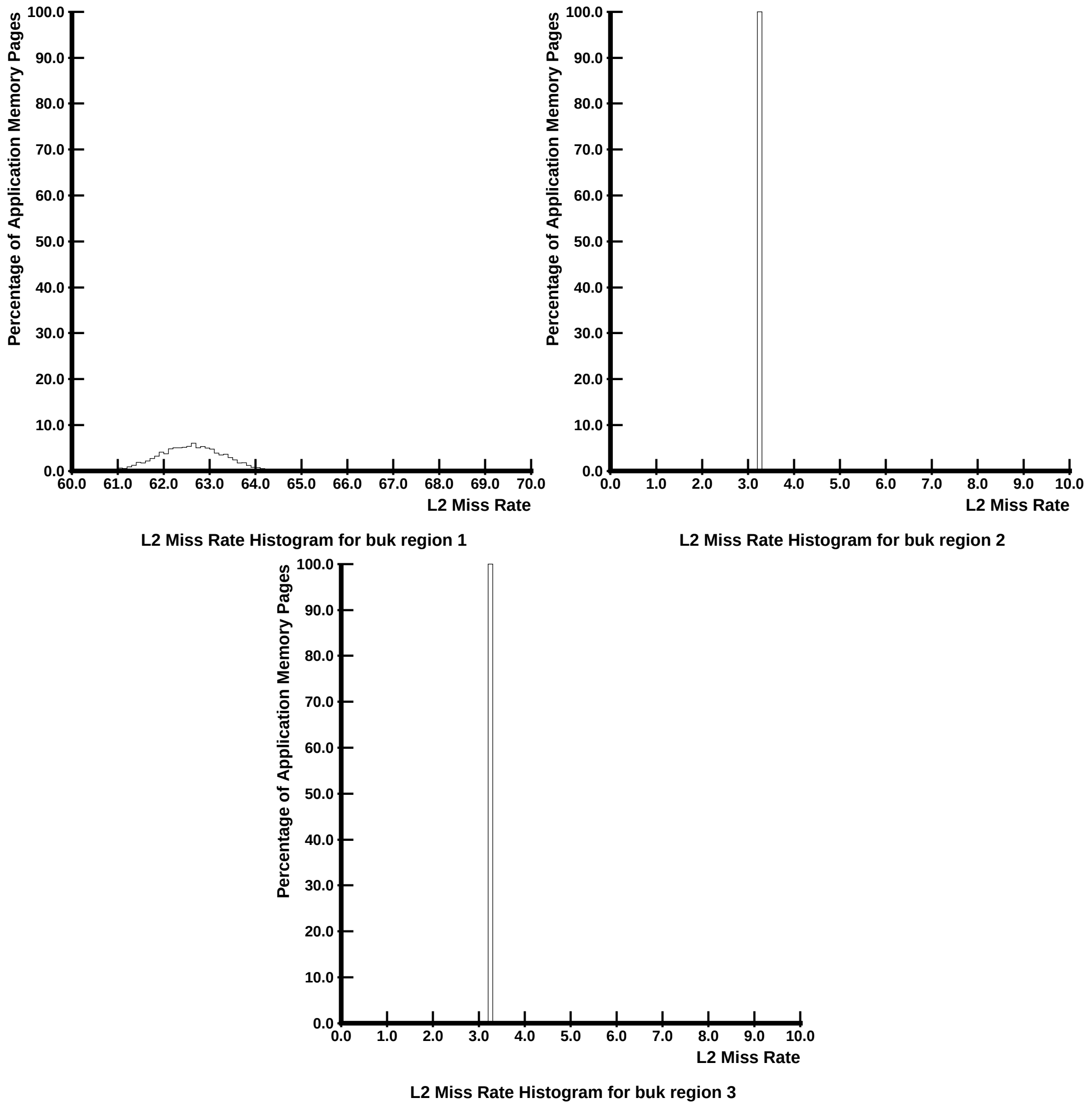


Figure B.4: Distribution of per page L2 miss rate over all pages for buk regions 1-3. Regions 2 and 3 are sequential.

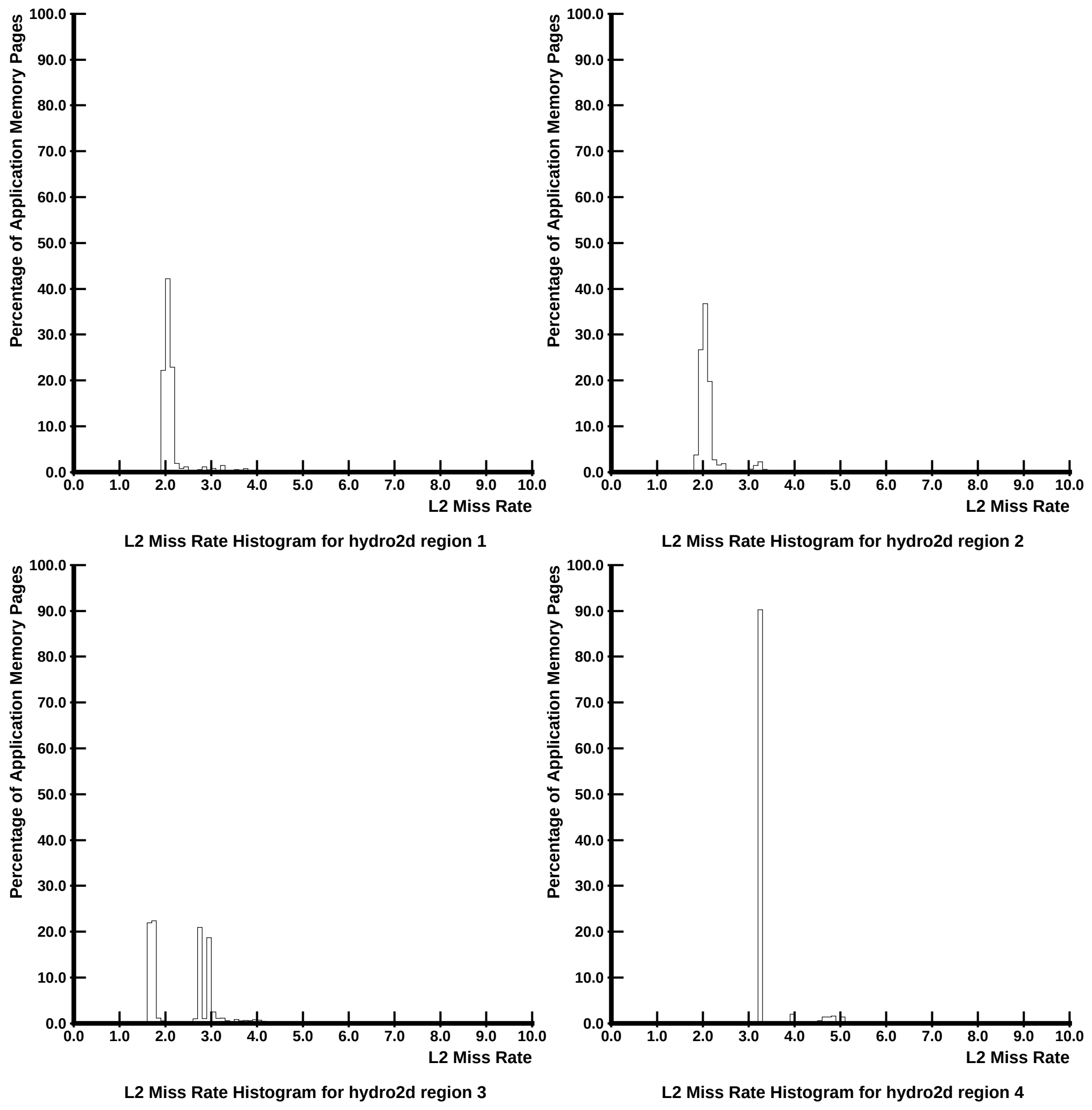


Figure B.5: Distribution of per page L2 miss rate over all pages for hydro2d regions 1-4. Regions 1, 2, and 4 are sequential.

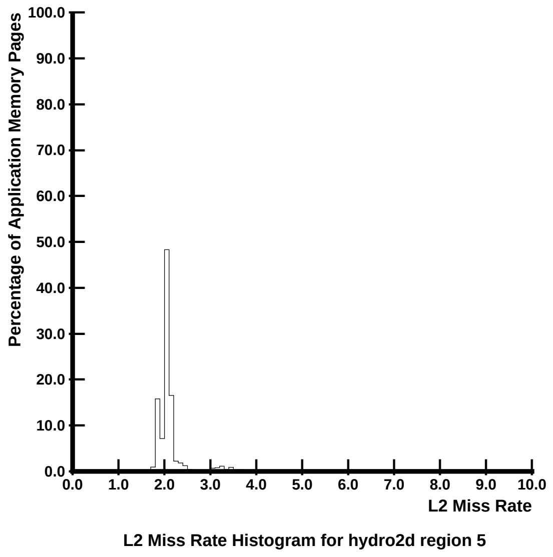


Figure B.6: Distribution of per page L2 miss rate over all pages for hydro2d regions 5. Region 5 is non-sequential.

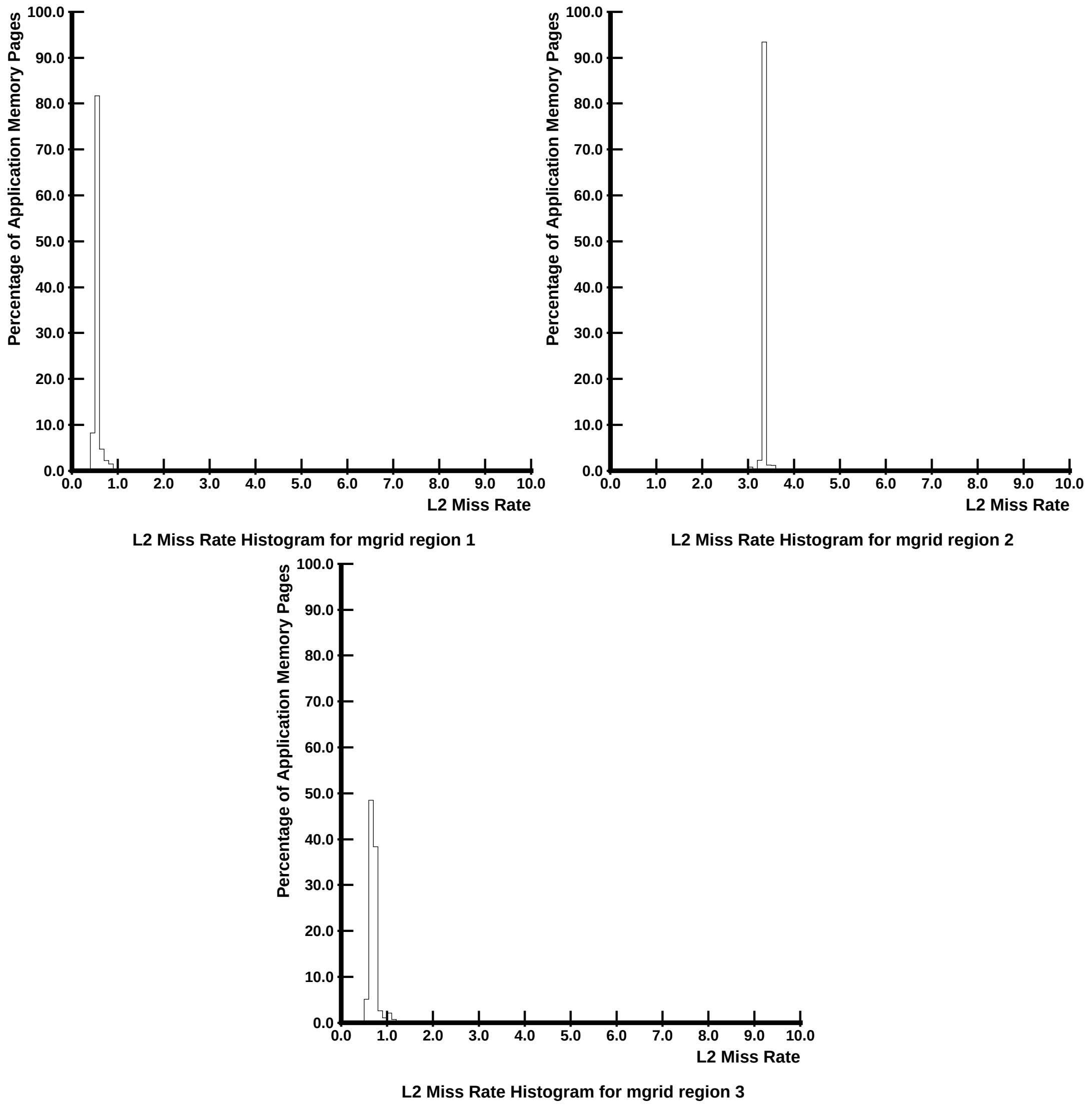


Figure B.7: Distribution of per page L2 miss rate over all pages for mgrid regions 1-3. Regions 1, 2, and 3 are sequential.

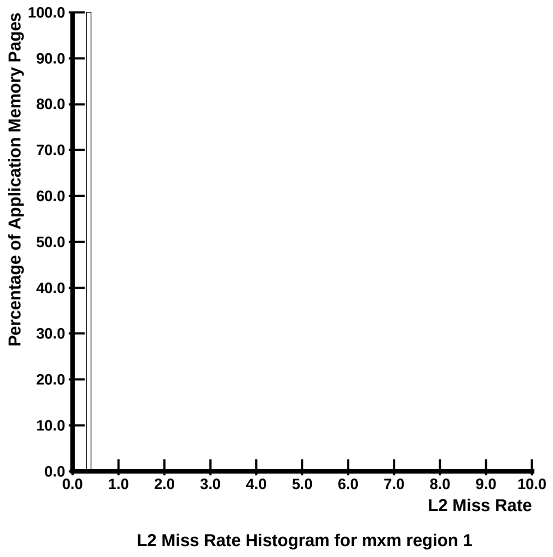


Figure B.8: Distribution of per page L2 miss rate over all pages for mxm region 1. Region 1 is sequential. Regions 2 and 3 not shown (see explanation of simulator limitation in Section 4.3.

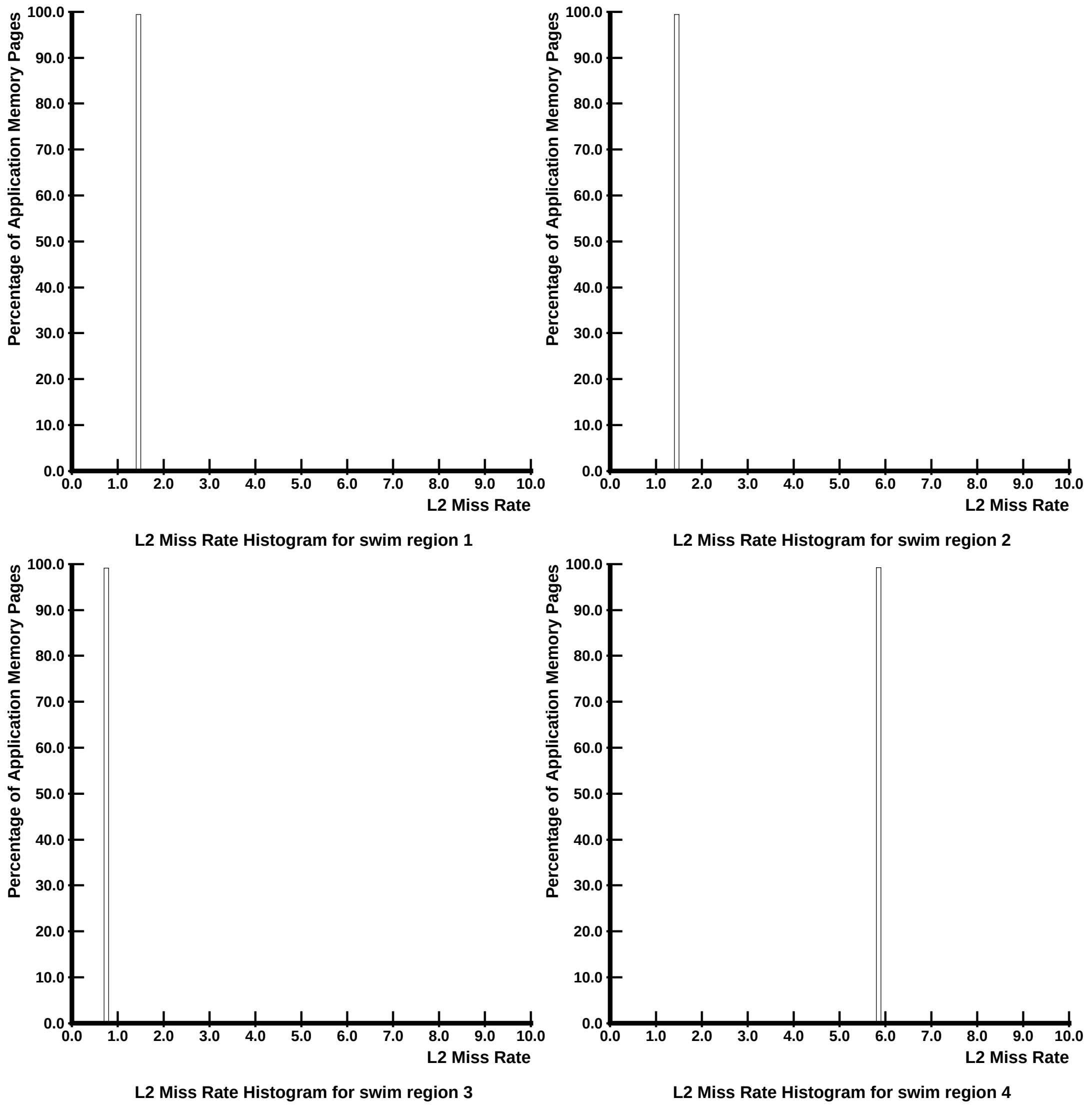


Figure B.9: Distribution of per page L2 miss rate over all pages for swim regions 1-4. Regions 1, 2, 3, and 4 are sequential.

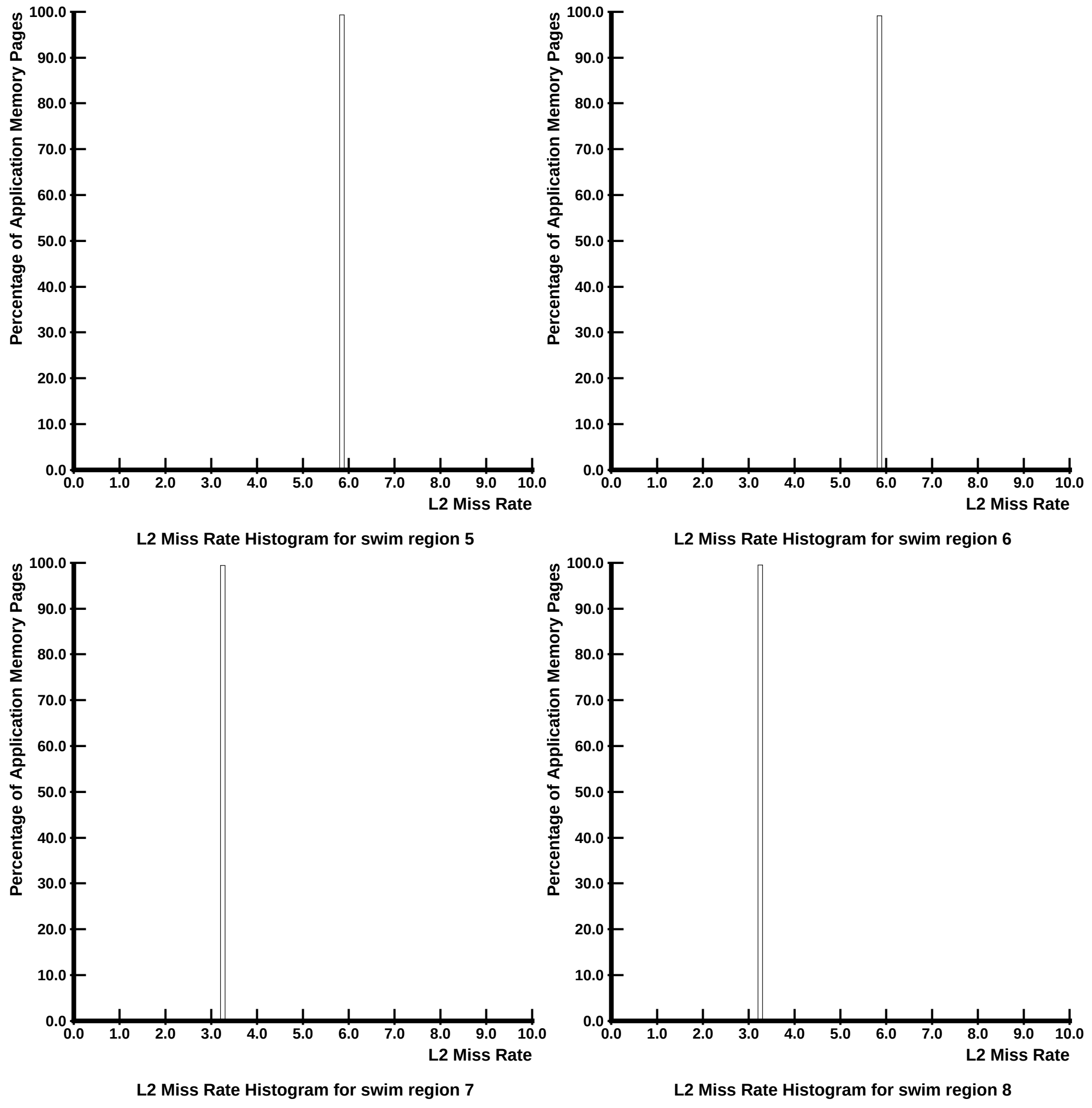
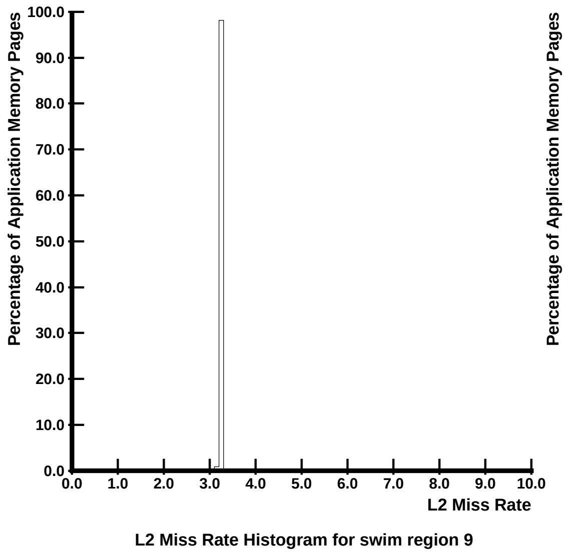
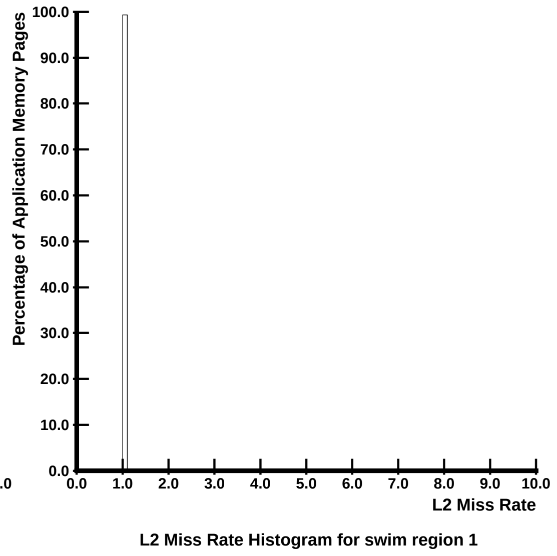
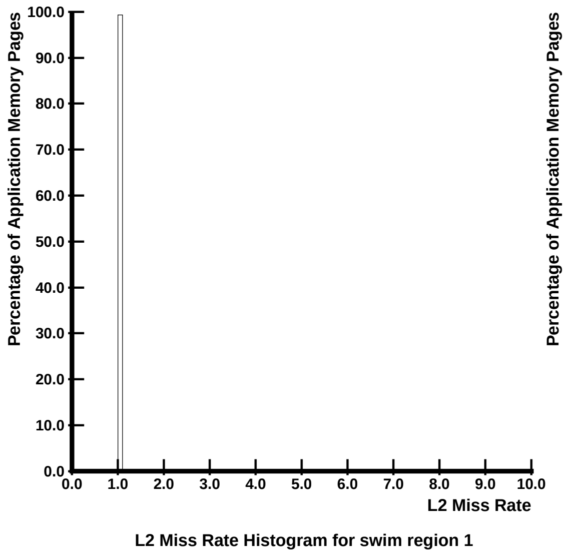


Figure B.10: Distribution of per page L2 miss rate over all pages for swim regions 5-8. Regions 5, 6, 7, and 8 are sequential.







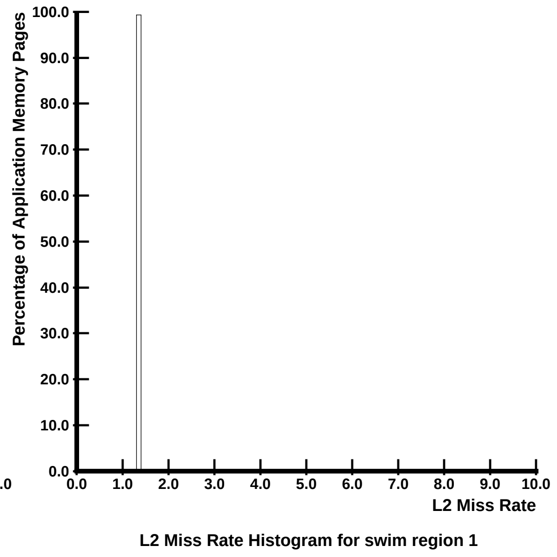
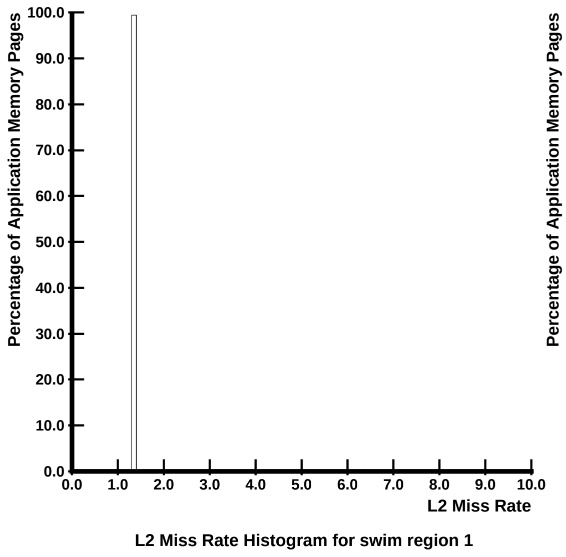


Figure B.11: Distribution of per page L2 miss rate over all pages for swim regions 9-12. Regions 9, 10, 11, and 12 are sequential.



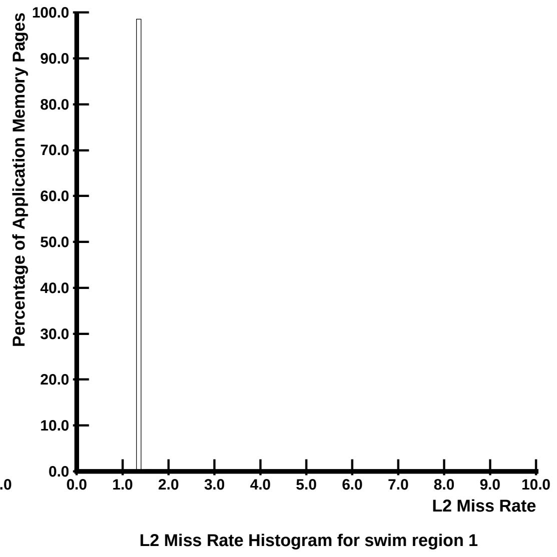


Figure B.12: Distribution of per page L2 miss rate over all pages for swim regions 13-14. Regions 13 and 14 are sequential.

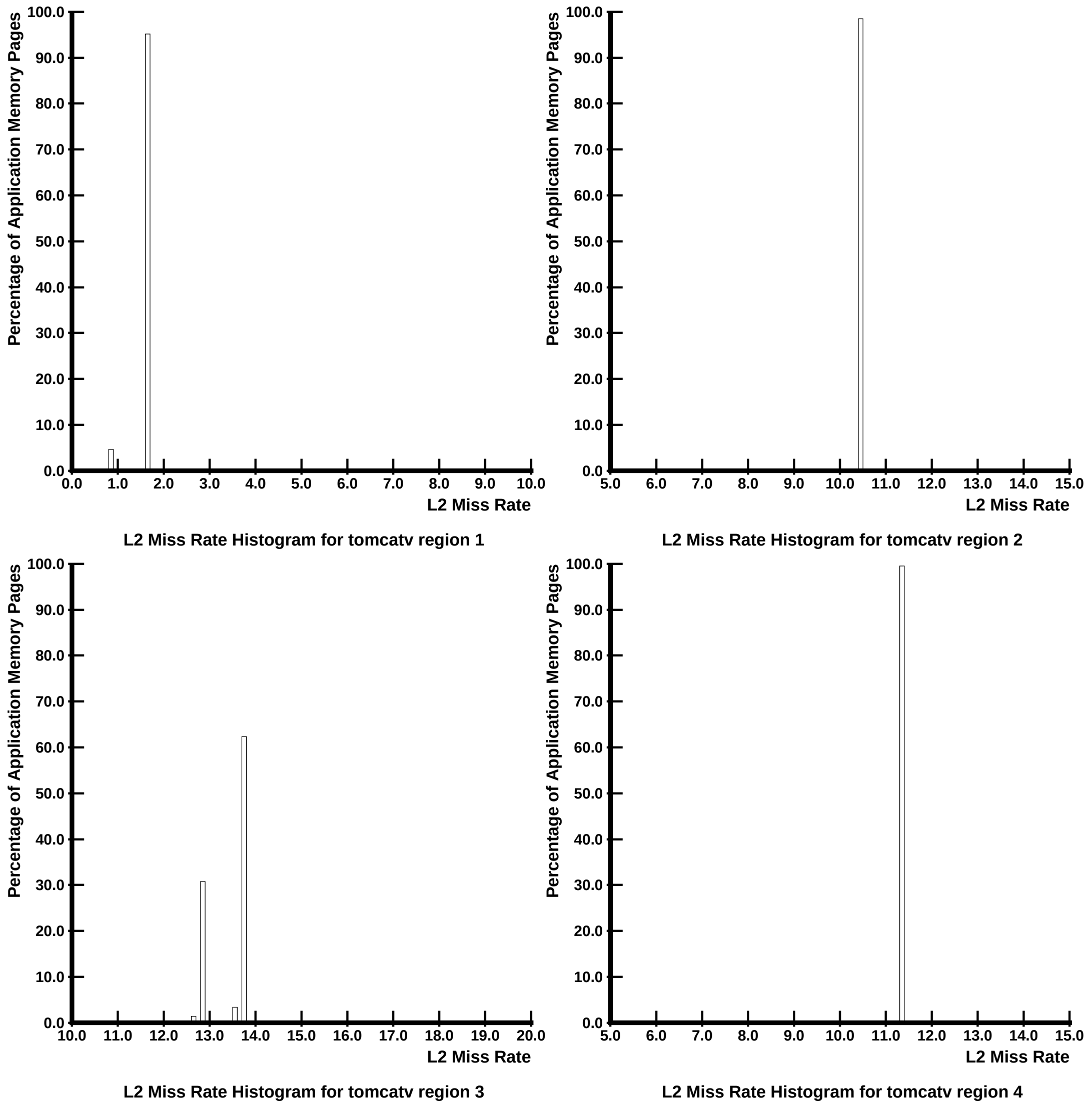


Figure B.13: Distribution of per page L2 miss rate over all pages for tomcatv regions 1-4. Regions 1 and 4 are sequential.

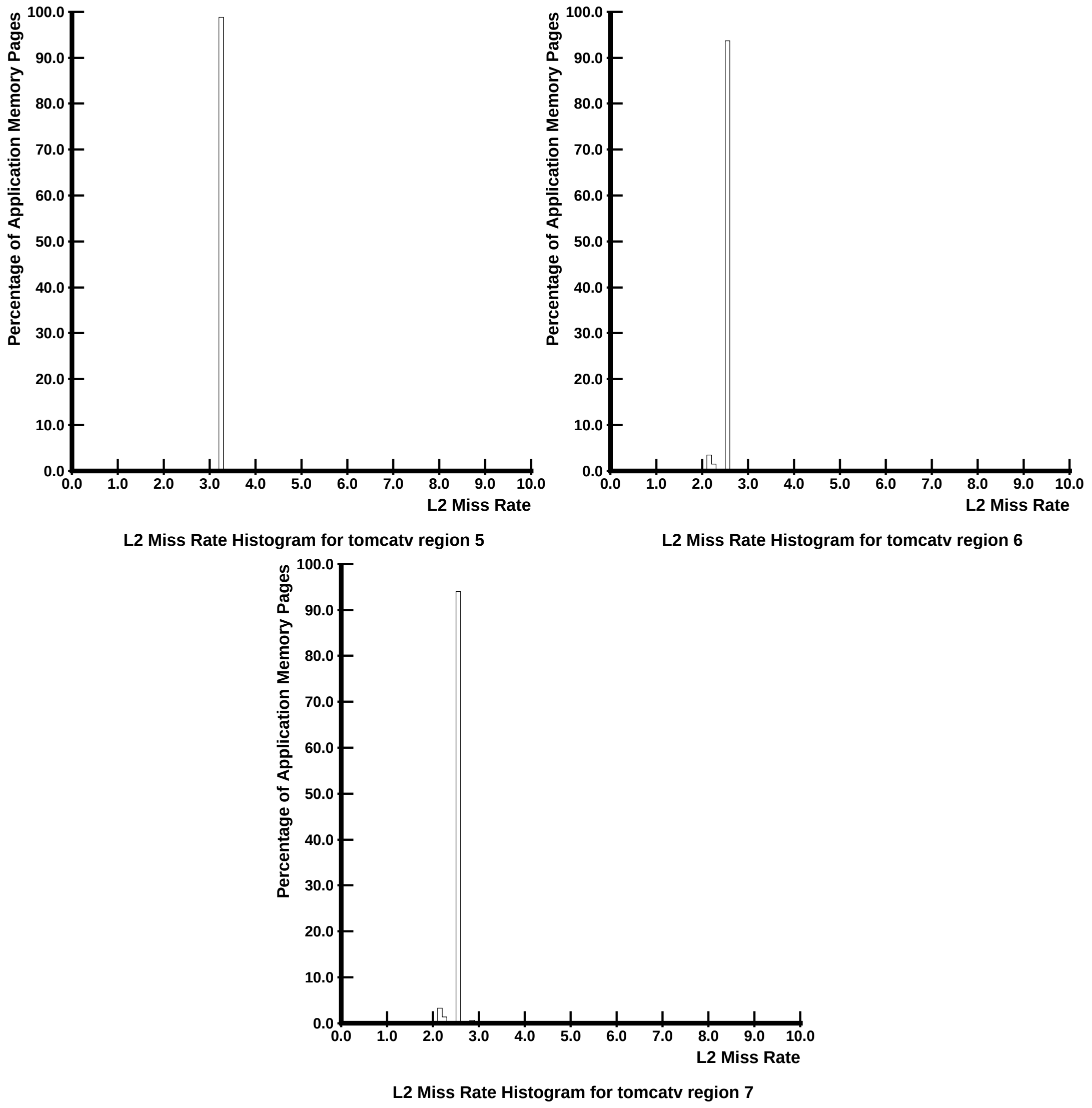
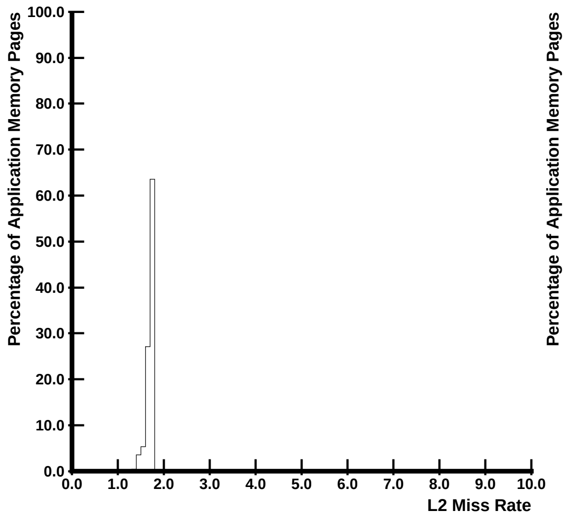
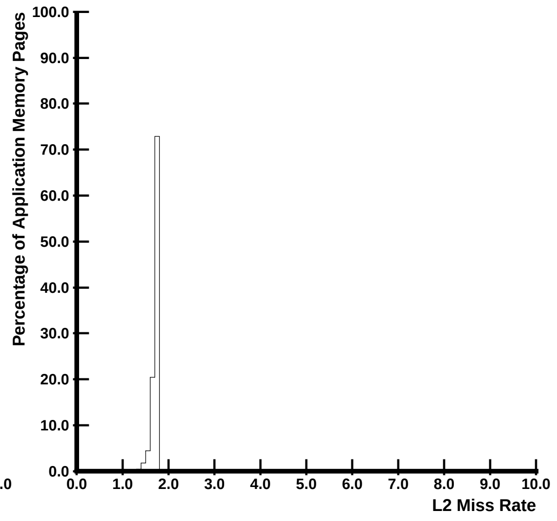


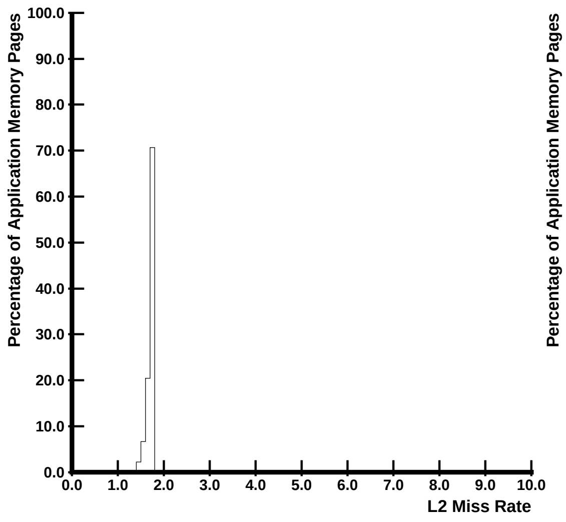
Figure B.14: Distribution of per page L2 miss rate over all pages for tomcatv regions 5-7. Regions 5, 6, and 7 are sequential.



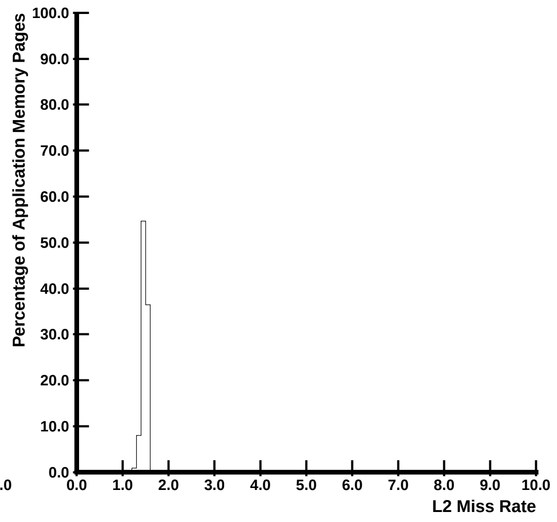
L2 Miss Rate Histogram for turb3d region 1



L2 Miss Rate Histogram for turb3d region 2



L2 Miss Rate Histogram for turb3d region 3



L2 Miss Rate Histogram for turb3d region 4

Figure B.15: Distribution of per page L2 miss rate over all pages for turb3d regions 1-4. No sequential regions.

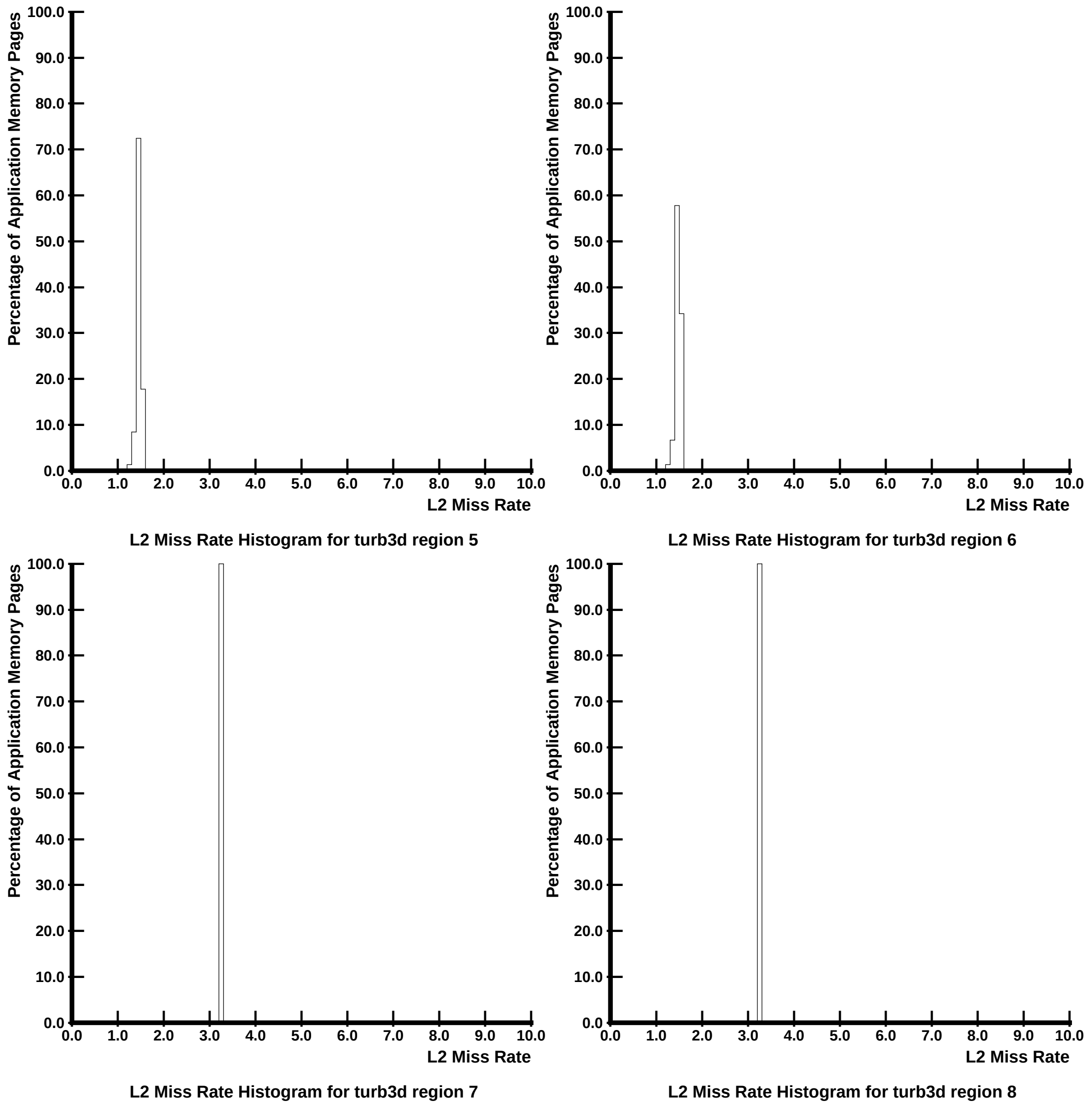


Figure B.16: Distribution of per page L2 miss rate over all pages for turb3d regions 5-8. Regions 7 and 8 are sequential.

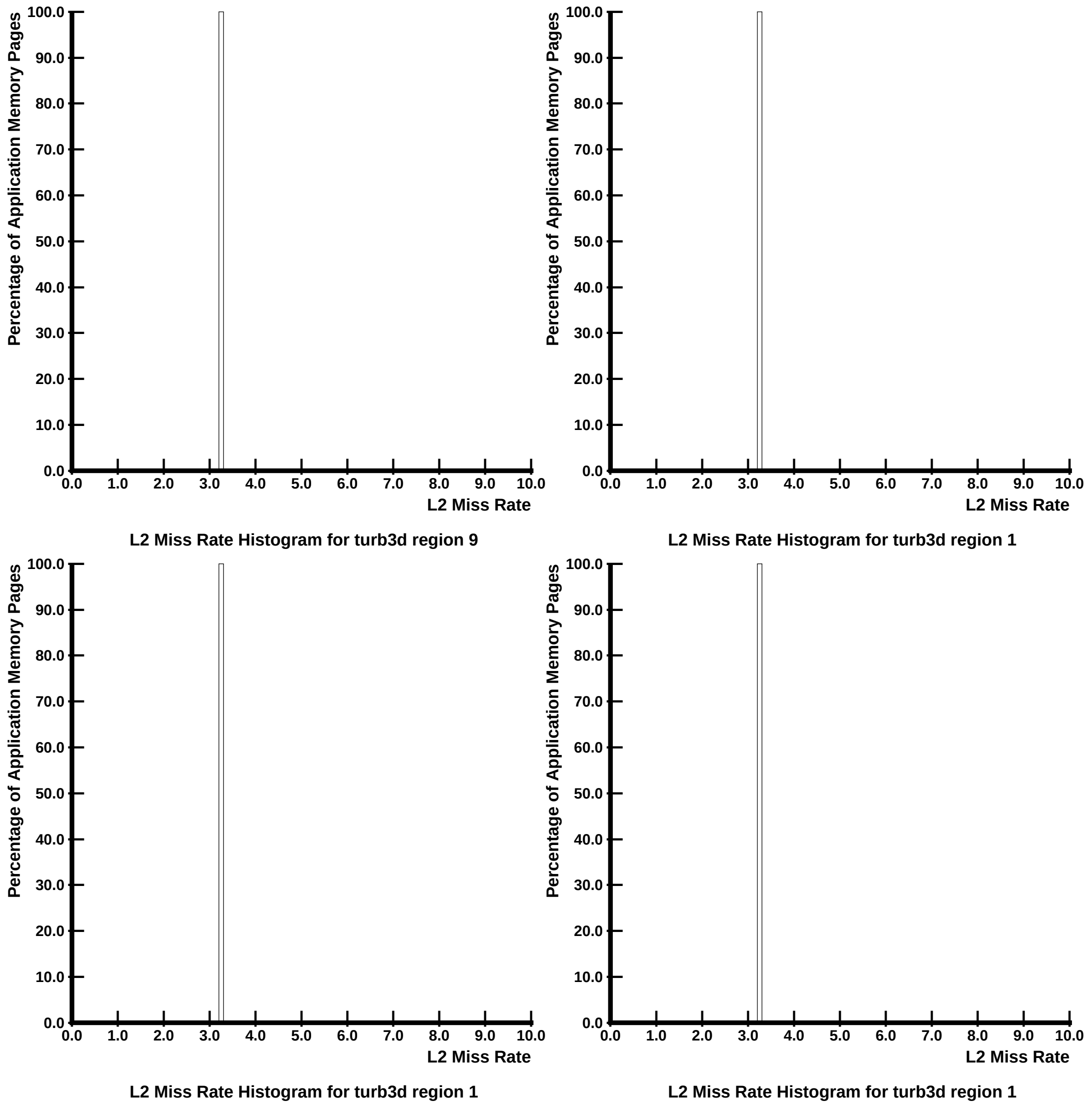


Figure B.17: Distribution of per page L2 miss rate over all pages for turb3d regions 9-12. Regions 9, 10, 11, and 12 are sequential.

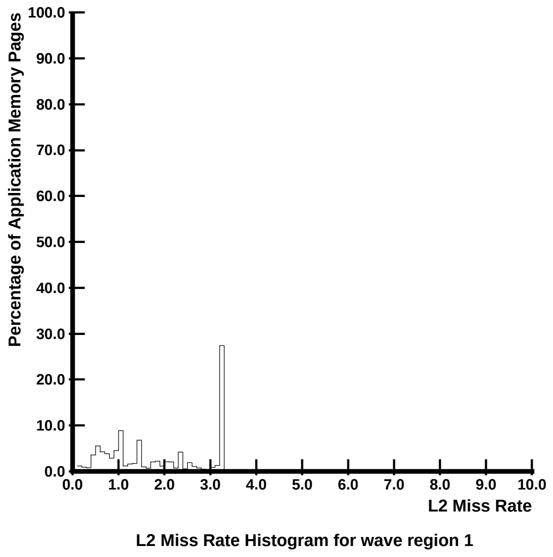


Figure B.18: Distribution of per page L2 miss rate over all pages for wave region 1. Region 1 is sequential.

Bibliography

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[2] O. Babaoglu and W. Joy. Converting a swap-based system to do paging in architecture lacking page-reference bits. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 78–86, December 1981.

[3] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[4] W. Bolosky, M. Scott, R. Fitzgerald, and A. Cox. Numa policies and their relationship to memory architecture. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[5] W. Bolosky, M. Scott, R. Fitzgerald, and A. Cox. Numa policies and their relationship to memory architecture. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[6] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX SEDMS IV Conference*, 1993.

[7] Tim Brecht. On the importance of parallel application placement in numa multiprocessors. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 1–18, September 1993.

[8] Ray Bryant and John Hawkes. Linux scalability for large numa systems. In *Proceedings of the Linux Symposium*, July 2003.

[9] P. Cao, E. Felten, and K. Li. Application-controlled file caching policies. In *Proceedings of the Usenix summer 1994 Technical Conference*, pages 171–182, 1994.

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

[11] Alan E. Charlesworth. The sun fireplane system interconnect. *IEEE Micro*, 22(1):36–45, January 2002.

[12] Hewlett-Packard Company. Hp 9000 n4000 enterprise server using hp-ux 11.00 64-bit and informix extended parallel server 8.30fc2: Tpc-h full disclosure report. Technical report, Hewlett-Packard Company, May 2000.

[13] Standard Performance Evaluation Corporation. The spec95 benchmark suite, http://www.specbench.org.

[14] 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.

[15] David E. Culler and Jaswinder Pal Singh with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Incorporated, San Francisco, California, 1999.

[16] Fredrik Dahlgren and Josep Torrellas. Cache-only memory architectures. *Computer*, pages 72–79, June 1999.

[17] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.

[18] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of alphaserver GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[19] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *Proceedings of ACM SIGMETRICS*, pages 115–126, 1997.

[20] Alexander Grbic. Hierarchical directory controllers in the NUMAchine multiprocessor. Master's thesis, Dept. of Electrical and Computer Engineering, University of Toronto, 1996.

[21] Alexander Grbic, Steve Brown, Steve Caranci, et al. Design and implementation of the NUMAchine multiprocessor. In *Proc. of the 35th Design Automation Conference*, pages 66–69, 1998.

[22] Robin Grindley. *The NUMAchine Multiprocessor: Design and Analysis.* PhD thesis, Dept. of Electrical and Computer Engineering, University of Toronto, 1999.

[23] Robin Grindley, Tarek Abdelrahman, Steve Brown, Steve Caranci, Derek Devries, Ben Gamsa, Alex Grbic, Mitch Gusat, Robert Ho, Orran Krieger, Guy Lemieux, Kelvin Loveless, Narig Manjikian, Paul McHardy, Sinisa Srbljic, Michael Stumm, Zvonko Vranesic, and Zjelko Zilic. The NUMAchine multiprocessor. In *Proc. of the International Conference on Parallel Processing*, pages 69–79, 2000.

[24] E. Hagersten, A. Landin, and S. Haradi. Ddm – a cache-only memory architecture. *Computer*, pages 44–54, September 1992.

[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] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report 9202001, Kendall Square Research, Boston, February 1992.

[27] Raj Jain. *The Art of Computer Systems Performance Analysis.* Wiley, 1991.

[28] Dongming Jiang and Jaswinder Pal Singh. A methodology and an evaluation of the SGI Origin2000. In *Proc. of the Joint Intl. Conference on Measurement and Modeling of Computer Systems*, pages 171–181, 1998.

[29] T. Joe. *COMA-F: A Non-Hierarchical Cache Only Memory Architecture.* PhD thesis, Dept. of Electrical Engineering, Stanford University, 1995.

[30] T. Joe and J. Hennessy. Evaluating the memory overhead required for coma architecture. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 82–93, April 1994.

[31] R. P. LaRowe Jr., C. S. Ellis, and L. S. Kaplan. The robustness of numa memory management. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 137–151, October 1991.

[32] R. P. LaRowe Jr., C. S. Ellis, and L. S. Kaplan. The robustness of numa memory management. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 137–151, October 1991.

[33] Kendall Square Research. *KSR1 Principles of Operation*. Waltham, MA, 1991.

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

[35] Daniel Lenoski, James Laudon, Truman Joe, et al. The DASH prototype: Implementation and performance. In *Proc. of the 19th Intl. Symposium on Computer Architecture*, pages 92–103, 1992.

[36] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Computing*, pages 125–132, August 1988.

[37] D. B. Loveman. High performance fortran. *IEEE Parallel and Distributed Technology*, pages 25–42, February 1993.

[38] T. Lovett and R. Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proc. of the 23rd Intl. Symposium on Computer Architecture*, pages 308–317, 1996.

[39] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory sys-

tems. Technical Report TR 535, Department of Computer Science, University of Rochester, 1994.

[40] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguade. A case for user-level dynamic page migration. In *Proc. of the 14th ACM International Conference on Supercomputing*, pages 119–130, 2000.

[41] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The s3.mp scalable shared memory multiprocessor. In *International Conference on Parallel Processing*, August 1995.

[42] E. O'Neil and C. Shaefer. The argot strategy iii: the bbn butterfly multiprocessor. In *Proceedings of Supercomputing, Vol.II: Science and Applications*, pages 214–227, 1989.

[43] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple coma. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 276–285, January 1995.

[44] SGI. *Memory Management Control Programmer's Manual (IRIX 6.5).* SGI, 2002.

[45] Abraham Siberschatz and Peter Baer Galvin. *Operating Systems Concepts.* Addison-Wesley Pub Co., 1997.

[46] Jaswinder Pal Singh, Truman Joe, Anoop Gupta, and John L. Hennessy. An empirical comparison of the kendall square research KSR-1 and stanford DASH multiprocessors. In *Supercomputing*, pages 214–225, 1993.

[47] Yannis Smaragdakis, Scott Kaplan, and Paul R. Wilson. EELRU: Simple and effective adaptive page replacement. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 122–133, 1999.

[48] Sinisa Srbljic. An adaptive coherence protocol using write invalidate and write update mechanisms. *CIT, Journal of Computing and Information Technology*, 4(3):187–197, September 1996.

[49] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.

[50] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[51] Andrew Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[52] Igor Tartalja and Veljko Milutinovic. *The Cache Coherence Problem in Shared Memory Multiprocessors: Software Solutions*. IEEE Computer Society Press, 1996.

[53] Josep Torrellas and David Padua. The Illinois aggressive coma multiprocessor project (I-ACOMA). In *Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computing*, 1996.

[54] J.E. Veenstra. MINT tutorial and user manual. Technical Report 452, Computer Science Department, University of Rochester, May 1993.

[55] Ben Verghese. *Resource Management Issues for Shared-Memory Multiprocessors*. PhD thesis, Dept. of Electrical Engineering, Stanford University, 1998.

[56] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, October 1996.

[57] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. The hector multiprocessor. *Computer*, 24(1), 1991.

[58] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 115–126, December 2002.

[59] S.C. Woo, M. Ohara, E. Torrie, J. P. Shingh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Intl. Symposium on Computer Architecture*, pages 24–36, 1995.

[60] Z. Zhang and J. Torrellas. Reducing remote conflict misses: NUMA with remote cache versus COMA. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, January 1997.