The Robustness of NUMA Memory Management*

Richard P. LaRowe Jr.†
Carla Schlatter Ellis‡
Laurence S. Kaplan§

Abstract

The study of operating systems level memory management policies for nonuniform memory access time (NUMA) shared memory multiprocessors is an area of active research. Previous results have suggested that the best policy choice often depends on the application under consideration, while others have reported that the best policy depends on the particular architecture. Since both observations have merit, we explore the concept of policy tuning on an application/architecture basis.

We introduce a highly tunable dynamic page placement policy for NUMA multiprocessors, and address issues related to the tuning of that policy to different architectures and applications. Experimental data acquired from our DUnX operating system running on two different NUMA multiprocessors are used to evaluate the usefulness, importance, and ease of policy tuning.

Our results indicate that while varying some of the parameters can have dramatic effects on performance, it is easy to select a set of default parameter settings that result in good performance for each of our test applications on both architectures. This apparent robustness of our parameterized policy raises the possibility of machine-independent memory management for NUMA-class machines.

*This research was supported in part by NSF grants CCR-8721781 and CCR-8821809. LaRowe was partially supported by a Computer Measurement Group (CMG) Fellowship.

†Encore Computer Corp., 257 Cedar Hill Street, Marlborough, MA 01752. internet: `rlarowe@encore.com`

‡Department of Computer Science, Duke University, Durham, NC 27706. internet: `carla@cs.duke.edu`

§BBN Advanced Computers Inc., 70 Fawcett Street, Cambridge, MA 02238. internet: `lkaplan@bbn.com`

1 Introduction

Multiprocessor designs that can be classified as NUMA (nonuniform memory access time) architectures are becoming increasingly important with the drive to provide shared memory on a larger scale. Distributing the memory of a large scale multiprocessor among the processors so that each memory module can be considered close to some processor(s) while being distant from others offers clear price/performance advantages. The implication of this design decision is that the placement and movement of code and data become crucial to performance. Unfortunately, presenting the programmer with an explicit NUMA memory model results in a significant additional programming burden. The operating system can play a major role in managing placement through the policies and mechanisms of the virtual memory subsystem (e.g., by migrating and replicating shared pages).

OS-level NUMA memory management is an area of active research [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13]. This body of work has demonstrated that dynamic page placement is indeed effective. In fact, the effectiveness question has been the focus of many of the previous studies that took the approach of proposing an algorithm and evaluating its performance (e.g., by comparing it against some static, often random, page placement policy on a particular architecture). The next level of questioning involves exploring the policy design space to gain an understanding of the range of possible management options, of why certain policy alternatives behave as they do, and of how that behavior depends on the memory reference patterns of applications and architectural parameters. This has been the goal of our recent work.

In order to do comparative policy studies, we developed the DUnX kernel for the BBN GP1000 as a framework for implementing a wide assortment of dynamic page placement strategies. We initially viewed the policy design space in terms of distinct points; individual policies that captured various combinations of the large number of factors that we suspected might affect performance. Nearly fifty policies were tested, including (at least approximations of) most of the published policies, and the experimental results were reported in [9]. The DUnX policy space covered both pull- and push-based page movement, the collection and use of reference history information of various kinds and levels of detail, different mechanisms for triggering new placement decisions, different means of limiting excessive page movement (page bouncing), and different criteria for choosing between migration and replication operations.

One of the key observations made in those early experiments was that the application program under consideration often seemed to determine which placement policy delivered the best performance. For example, a policy that was eager to migrate and replicate pages performed poorly when faced with an application that had a substantial amount of active sharing. Similarly, conservative policies did not perform well when faced with applications exhibiting very neat sharing patterns that were ideal for page migration.

It is intuitively obvious that architectural features can influence the behavior of dynamic page placement policies as well. For example, in a system with very fast page transfers but incredibly slow remote reference times, migration and replication are very desirable. On the other hand, in a system in which page transfers are very slow, the desire to migrate pages is much less. Bolosky et al. consider the importance of architectural dependencies in [4]. The importance of page transfer speed for the policy implemented in the Platinum operating system on the BBN Butterfly Plus multiprocessor is discussed by Cox and Fowler [5].

The approach originally used to specify policies in the first version of the DUnX kernel favored the comparison of qualitatively different policies (e.g., evaluating the effect of different features). It became apparent that many of the remaining questions involved exploring the degree to which some of the techniques should be applied, raising the issue of policy tuning. Our experience also allowed us to prune and consolidate techniques. Thus, it appeared that our further investigation of policy issues could best be formulated in the context of a single parameterized policy and studied by varying the parameter settings and measuring the effect on performance. This insight led to the development of version two of DUnX, which supports such a tunable policy. This single policy seems to capture a fairly large region of the policy design space, including the most successful policies identified in our earlier experiments.

One of the potential benefits of a parameterized NUMA memory management policy, incorporated into a virtual memory subsystem that has been structured for portability (DUnX is a descendent of Mach [14]), may be an ability to tune for architectural differences. We investigate the premise that one policy may be used on different architectures for different applications and be tuned with relative ease to deliver acceptable performance. We address this issue by considering the effects that the differences between BBN's GP1000 and TC2000 architectures have on the tuning of policy parameters using DUnX implementations on both machines.

In the next section, we describe our parameterized policy and its implementation. In Section 3, we discuss our experimental methodology for investigating issues related to tuning, and in Section 4, we present the results of our experiments. Our results indicate that while varying some of the parameters can have dramatic effects on performance, it is easy to select a set of default parameter settings that result in good performance for each of our test applications on both architectures. We summarize and conclude in Section 5.

2 A Parameterized Dynamic Page Placement Policy

Our policy is a highly parameterized dynamic page placement policy that can move a page to a local frame upon demand (i.e., it is pull-based). It supports both migration and replication with the choice between the two operations based on reference history (the recent history of modifications made to the page). A directory-based invalidation scheme is used to ensure the coherence of replicated pages. The policy applies a freeze/defrost strategy (an idea adopted from [5]) to control page bouncing. This means that excessive page movement is controlled by freezing the page in place and forcing remote accesses. The freezing criterion is based on the time since the most recent invalidation of the page. Determining when to defrost a frozen page and trigger reevaluation of its placement is based on both time (by how often such decisions are made) and reference history (the recent remote/local usage). At the same time, choosing how to trigger new placement decisions is based on reference data (recent modification history).

Six parameters control the behavior of the policy. One set of parameters controls the frequencies at which certain events take place (defrost and triggering decision-points, reference data collection, and aging of usage counts). Other parameters set thresholds on the interpretation of reference data (e.g., defining "recent" history). These parameters are not intended to be orthogonal, but they do provide a means to systematically study the policy design space. With appropriate parameter settings, behavior of the policy can be adjusted to mimic a wide variety of freeze-based policies. In addition, we can use parameter tuning to study the effects of individual policy features on performance when faced with different application programs or different memory architectures. The parameters and their roles are summarized in Table 1, and are discussed in the following subsections.

The policy is comprised of two parts. The first defines

Param	*Role*
`freeze-window`	defines size of "recent invalidations" window for freezing decisions
`recent-mod`	controls the replication vs. migration decision
`scan-delay`	sets the rate of scanner daemons
`sample-passes`	adjusts the number of reference collection samples
`defrost-trigger`	remote count $\geq$ local count "successes" needed to defrost
`trigger-method`	controls the "invalidate all" vs. "invalidate remote" trigger decision

Table 1: Policy Parameter Summary

the behavior of the policy when faced with a page fault, and the second defines the behavior of the page scanner daemons used to trigger the reevaluation of earlier page placement decisions.

2.1 Fault Time Behavior

The policy specifies how a fault on a page that is already resident in some physical frame is to be handled. The chosen course of action depends primarily on the recent reference history for the page and the settings of the `freeze-window` and `recent-mod` policy parameters.

The processing of a fault on a page that is not currently replicated depends on whether the frame in which it resides is local or remote. If the frame is remote, the policy must decide between installing a remote mapping and migrating or replicating the page.

The first step involves determining whether the page should be *frozen* by checking to see whether the most recent invalidation of the page (due to a page migration or coherency fault) occurred within the past `freeze-window` milliseconds. If the page is frozen (either imposed just now or sometime in the past), the remote frame is used to service the fault. The `freeze-window` parameter essentially limits the rate at which invalidations of a page can occur. When `freeze-window` is set to zero, the policy behaves much like the caching policies used in the proposed distributed shared memory environments (e.g., [11, 12]). When `freeze-window` is set to infinity, a page may only be invalidated once (migrated once, or replicated until the first coherency fault occurs) before it is frozen. Values of `freeze-window` between these two extremes allow varying amounts of dynamic page placement activity (migration and replication). In essence, the `freeze-window` parameter controls the eagerness of the policy to migrate and replicate pages.

Once it is determined that the page can move, it is necessary to decide between migration and replication. The `recent-mod` parameter controls the replication versus migration decision. The policy checks to see if the page has been recently modified by comparing the modification history (maintained by the page scanners through aging of the hardware modification bits) to the `recent-mod` parameter. If the aged modification counter exceeds the `recent-mod` threshold or if a write reference triggered the current fault, the page is migrated to a local free frame. Otherwise, a local free frame is used to create a copy of the page. Write access to all copies is prohibited, allowing the fault handler to ensure data coherency. When `recent-mod` < 0, migration is always chosen in favor of replication (i.e., replication is not allowed). When `recent-mod` $= \infty$, replications are chosen over migrations on any fault not triggered by a write reference. If we assume that the goal of the policy is to replicate only pages being referenced in a read-only fashion, then we can characterize `recent-mod` as the point at which the policy concludes that the last modification of the page is far enough in the past that it can assume that the page is now being referenced in a read-only fashion.

The handling of a fault on a page that is already replicated requires that data coherence must be maintained. If a write memory reference triggered the fault, all but the copy eventually used to satisfy the fault must be invalidated. If no local copy of the page exists, one of the existing replicas is migrated to a local frame. If a read memory reference triggered the fault, then the policy either uses an existing replica, or creates an additional local one if none already exists. There is no need to check for freezing, since it is impossible for a page that should be frozen to be replicated.

2.2 Page Scanner Behavior

A dynamic page placement policy that relies solely on page faults to trigger placement decisions is severely limited, since even if it can detect when the placement of a page should be reevaluated, there is no way for the policy to gain control unless a page fault occurs. Our policy uses a page scanner on each processor node to trigger the reevaluation of earlier placement decisions.

The page scanners run every `scan-delay` seconds. Each time a scanner runs, it collects page reference and modification information for the frames on its processor. Separate local and remote reference counts are maintained, so that the policy can tell whether only local processes, only remote processes, or both local and remote processes are referencing each page.

After `sample-passes` runs in which a scanner simply collects data, it executes an additional phase in which it examines its frames to check for earlier placement decisions that ought to be reevaluated. In the DUnX implementation of the policy, the scanners also work to support the page replacement policy at this time.

The `scan-delay` and `sample-passes` parameters control the frequency that the scanners consider triggering the reevaluation of earlier placement decisions. Setting `scan-delay` to infinity essentially disables the scanner (making the other scanner parameter settings irrelevant). Values of `sample-passes` greater than zero allow the scanners to collect more accurate reference information (more samples) between checks on whether to trigger placement decision reevaluations. Of course, higher `sample-passes` values also increase the time, for a fixed `scan-delay`, between such checks.

The frames of primary interest are those containing frozen pages, for those are the pages which the policy forced some process(es) to use remotely. For each of the frames containing a frozen page, the policy compares the local and remote reference counts for that frame. If the remote reference count is as great as the local reference count, a counter stored with the frame data structure is incremented. Otherwise, the counter is reset to zero. Once the counter reaches `defrost-trigger`, the policy chooses to trigger the reevaluation of the earlier decision to use that frame.

The `defrost-trigger` parameter controls the eagerness of the scanner to trigger the reevaluation of earlier decisions. When `defrost-trigger` is set to zero, frozen pages are defrosted the first time the scanner considers doing so (this is the defrost strategy used by the Platinum policy [5]). Higher values ensure that reevaluations are triggered only when there is sufficient evidence that remote processes are using the page at least as much as local processes. Intuitively, when the cost of reevaluating a placement decision is high, higher `defrost-trigger` values would be more appropriate.

The policy triggers the reevaluation of earlier placement decisions in one of two ways, choosing between them by comparing the `trigger-method` parameter to the aged modification count. This is used as an indication of whether write access is likely in the near future. If a read access and, as a result, replication is to be expected, it is appropriate to invalidate remote mappings, causing page faults for all remote processes still using the page but not requiring local processes to incur extra faults. The more conservative triggering method involves invalidating all mappings (even those by local processes) to the frame. Both methods defrost the frame (clear the frozen bit). We suspect that in most cases, setting `trigger-method` equal to `recent-mod` is appropriate.

Operation	*GP1000*	*TC2000*
Cache Read	—	$0.061\mu s$
Local Read	$0.6\mu s$	$0.502\mu s$
Remote Read	$7.5\mu s$	$1.766\mu s$
Migration	$4.5ms$	$4.8ms$
Replication	$4.6ms$	$4.6ms$
Coherency Fault	$2.1ms$	$1.0ms$

Table 2: GP1000 and TC2000 Memory Operation Costs

3 Methodology

We study the effects of tuning our policy for different applications and architectures using implementations of DUnX on the BBN GP1000 and TC2000 shared memory multiprocessors and a collection of six parallel application programs obtained from various Butterfly user communities. In this section, we describe the differences in the two architectures under consideration, our workload collection, and our experimental approach.

The BBN GP1000 and TC2000 shared memory multiprocessors share many common features, yet also differ in several important aspects. Both architectures are Local/Remote NUMA multiprocessors based on a multistage Butterfly interconnection network. A Local/Remote architecture is one in which each memory module is physically associated with a processor. Processors can issue memory requests to any memory, but accesses to local memory are faster than remote. All remote memories are considered equally distant.

The most significant architectural differences between the GP1000 and TC2000 involve the microprocessors used by the machines (a Motorola 68020 in the GP1000 and a Motorola 88100 RISC processor in the TC2000) and the existence of data caches and interleaved memory on the TC2000. The TC2000 caches are not kept consistent via hardware, however, so typically only non-shared data are cached. In the current TC2000 DUnX implementation, we also allow replicated pages to be cached, using the software coherence mechanism in DUnX to ensure hardware cache consistency as well. Currently, other shared data are not cached, though we suspect that extensions to DUnX to support software consistency of the hardware caches for other shared data may prove effective. Also, in our experiments on the TC2000, we ignore the existence of interleaved memory. We suspect that the interleaved memory may prove useful for storing frozen pages, but we have not yet investigated this idea.

The most important difference between the two architectures, with respect to the performance of dynamic page placement, concerns the costs of different memory operations. In Table 2, the basic costs are summarized. It is important to note that the TC2000 used for our experiments consists of older processor nodes that run at 16 Mhz (as opposed to the 20 Mhz boards used in production models) and have only four megabytes of memory per node (as opposed to the typical sixteen megabytes). Thus, performance of a production TC2000 is likely better than that achieved in our experiments.

Several features of the data in Table 2 are worth pointing out. First, we see that the remote reference cost is significantly lower in the TC2000, resulting in a remote/local reference time ratio of just 3.5 as opposed to the 12.5 ratio on the GP1000. On the other hand, when pages are replicated the ratio of importance is the remote/cache ratio, which at 28.95, is much greater than the GP1000 remote/local ratio. The costs of migrating and replicating pages are similar for the two architectures, due to the fact that the TC2000 has no hardware support for block transfers whereas the GP1000 does. This implies that these costs *relative to processor speed* may seem as much as an order of magnitude greater in the TC2000. To see the significance of these cost differences, consider that if we ignore assorted contention factors, it generally pays to migrate a page when

$$at_l + M < at_r, \tag{1}$$

where a is the number of accesses the processor will make to the page, t_l is the local memory reference cost, t_r is the remote memory reference cost, and M is the page migration cost. Using the values in Table 2, the minimum a for which it pays to migrate a page on the GP1000 is 652, whereas for the TC2000, at least 3798 references are required. Using a similar expression for the page replication case, and accounting for the fact that replicated pages can be cached on the TC2000, we see that it pays to replicate a page on the GP1000 when at least 667 references to the page will be made, yet for the TC2000, 2698 references are required. This suggests that a more conservative approach to using migration and replication may be appropriate for the TC2000 than the GP1000. As we shall see, however, our results indicate that these architectural differences are not as significant to policy tuning as they first appear.

The workload used for our experimentation was developed independently from our project, in an effort to prevent unconscious attempts at making design decisions that might favorably affect our results. For most of the applications that comprise our workload collection (listed in Table 3), there are versions written in both UMA and NUMA styles. The exception is `msort`, for which we have no NUMA version. The UMA version does no NUMA-specific memory management, such as manually placing shared data pages or manually replicating read-only data structures. The NUMA version, on the other hand, is a highly-tuned implementation of the same program written to optimize memory reference locality. As one would expect, the NUMA version of an application is typically more complicated, less portable, and much more difficult to write. Preliminary experiments [9] have verified that dynamic page placement has an impact on the performance of the UMA version of these programs when run on a GP1000[1]

For each application in our workload collection, we began our study by "tuning" the parameter settings to achieve the best possible performance for that application on the GP1000 within the limits of the somewhat *ad hoc* tuning process. The search involved repeatedly picking a set of parameter values, monitoring behavior of the application with those settings, and then adjusting the settings appropriately. For example, if there appeared to be an excessive amount of page migration and replication activity, we might increase the `freeze-window` setting to see if performance would improve. The process requires a fair level of understanding of the role of the parameters, but our experience is that it is fairly easy to derive settings that deliver performance reasonably close to the best we have achieved for each application. Once we arrived at those parameter settings for an application, we designated them as the default settings for that application. The default settings are given in Table 3. We found that for all the applications in our workload collection, the default settings derived for the GP1000 also proved adequate for the TC2000. Thus, in all the experiments presented in this paper, the default settings for all but the parameter being varied were used. These settings are indicated in the figure captions.

In most of the figures in this paper, there are two heavy lines that mark the levels of performance obtained by the UMA and NUMA versions using static page placement (the upper line is the UMA result, and the lower the NUMA result). It must be noted that the static placement strategy is reasonably intelligent, placing pages in the memory of the faulting processor node during pagein, unless the data are explicitly bound to some other node. Each diamond in a plot is an experimental data point obtained with the UMA version of the program run under our dynamic policy with the corresponding parameter setting (multiple trials were done to check validity of data). The dotted lines connect the maximum and minimum levels of performance measured, and the thin solid line plots the mean values. In all plots in this paper, time on the y-axis is measured in elapsed seconds.

[1] Not included in this discussion are two other applications studied in [9] that were found not to benefit from dynamic placement, implying that the ability to disable such activity altogether might be valuable in a production system.

Prog	*Description*	`freeze-window`	`recent-mod`	`scan-delay`	`sample-passes`	`defrost-trigger`	`trigger-method`
`gauss`	simulates gaussian elimination with integer arithmetic	125ms	2^{16}	1.5s	0	4	2^{16}
`hh3d`	simulates electrical conduction in cardiac tissue	312.5ms	2^{16}	10s	0	4	2^{16}
`psolu`	solves $Ax = b$ for sparse A using block chaotic relaxation	187.5ms	2^{16}	10s	0	1	2^{16}
`hough`	computes hough transforms	31.25ms	2^{16}	10s	0	1	2^{16}
`msort`	merge sort of an integer array	50ms	0	10s	0	0	0
`wave`	solves the wave equation on a square grid with periodic boundary	250ms	2^{16}	10s	10	4	2^{16}

Table 3: Experimental Workload Collection

4 Evaluation and Results

In this section, we investigate the effects of varying the policy parameters. The goal of these experiments is to determine the importance and sensitivity of policy parameter tuning to the performance of our test applications on the two architectures considered.

4.1 Effect of Varying `recent-mod`

Intuitively, it seems that the `recent-mod` parameter, which controls the migration versus replication decision, should be important in balancing the costs of unnecessary coherency faults against missed opportunities for local accesses. For applications that use a significant amount of read-only sharing, page replication is highly desirable and should be chosen over migration whenever possible. On the other hand, a strong preference for replication when writing activity is high incurs the overhead of coherency faults.

The results of varying `recent-mod` for the `gauss` application on the GP1000 are given in Figure 1 (note that we also vary `trigger-method` so that it remains equal to `recent-mod`). It is clear that the higher the preference for replication over migration, the better the measured performance. Since the primary mode of sharing in `gauss` involves reading pivot rows of the matrix, which are never modified once they become pivot rows, the value of replication is not surprising. Figure 1 shows that when replication is always chosen over migration on read faults (`recent-mod` $= 2^{16}$), performance of the UMA version of `gauss` is nearly as good as with the NUMA version of the program. Intermediate `recent-mod` values result in performance better than without replication (the `recent-mod` < 0 case), but fail to take advantage of some potential page replications that further improve performance in the `recent-mod` $= 2^{16}$ case. Results of `recent-mod` experiments with the `psolu`, `hough`, and `wave` applications on the GP1000 resemble those obtained for the `gauss` program.

On the other hand, the `msort` application is an example of an application that does not benefit when replication is preferred over migration on the GP1000. The data are shown in Figure 2. This behavior is explained by the fact that there is no read-only sharing in `msort` which can benefit from page replication. As a result, when `recent-mod` $= 2^{16}$, we see a slight (barely noticeable) performance degradation due to using replication/coherency fault pairs to effectively migrate pages, which is a bit more costly than simply migrating the page. The `hh3d` results are similar to those of `msort` (though there is some slight improvement in `hh3d` performance with replication).

These data provide little evidence supporting the usefulness of intermediate `recent-mod` values. We believe that this is because the cost of incorrectly choosing replication over migration (a coherency fault) is not that significant in our DUnX implementation on the GP1000. The importance of avoiding coherency faults is simply not great enough to overcome the cost of avoiding or delaying desirable page replications.

Avoiding coherency faults may be more important for architectures in which processing of a coherency fault is very expensive. The results for the `hough` application on the TC2000 begin to illustrate this behavior. The GP1000 results with `hough` look very similar to the `gauss` results shown in Figure 1. On the TC2000, however, we see in Figure 3 that the importance of increas-

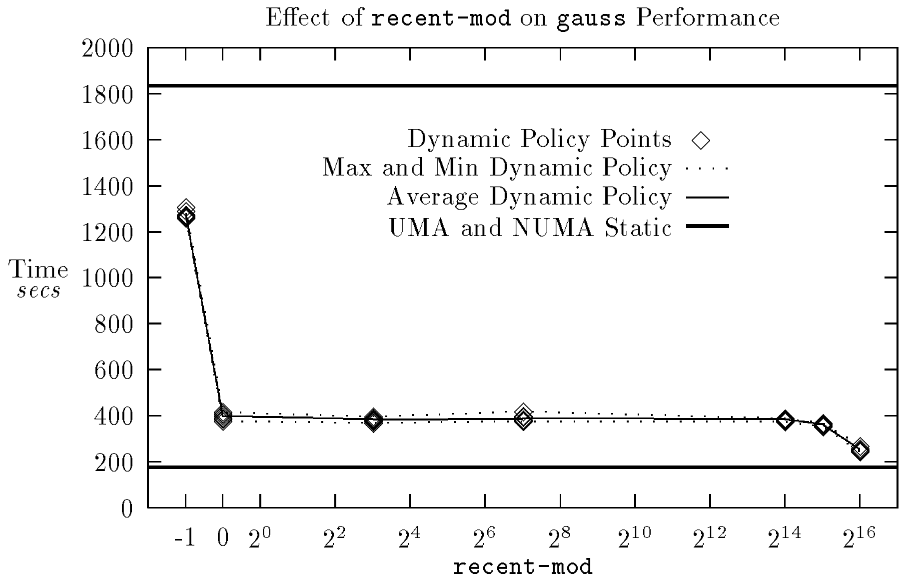

Figure 1: Effects of `recent-mod` on `gauss`/GP1000 Performance

(`freeze-window` $= 125ms$, `scan-delay` $= 1.5s$, `sample-passes` $= 0$, `defrost-trigger` $= 4$, `trigger-method` $=$ `recent-mod`)

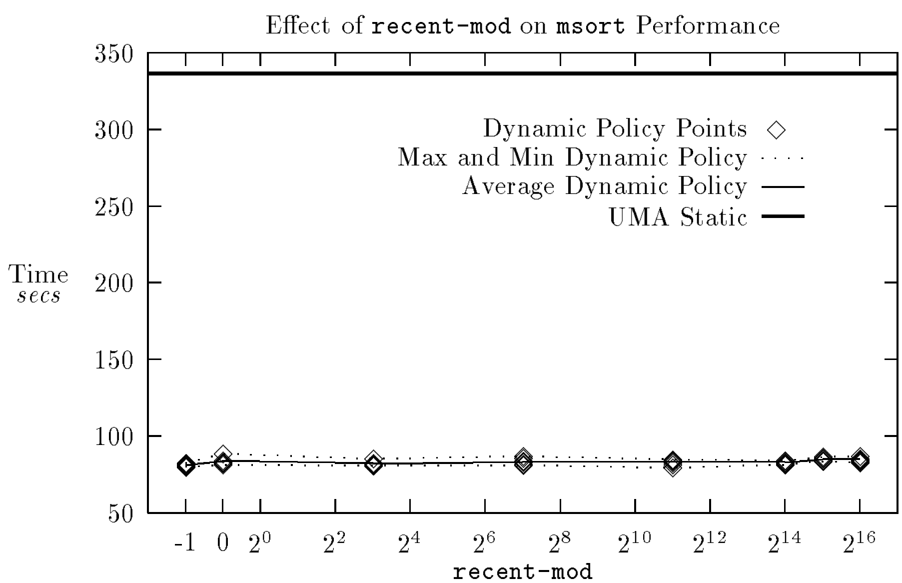

Figure 2: Effects of `recent-mod` on `msort`/GP1000 Performance

(`freeze-window` $= 50ms$, `scan-delay` $= 10s$, `sample-passes` $= 0$, `defrost-trigger` $= 0$, `trigger-method` $=$ `recent-mod`)

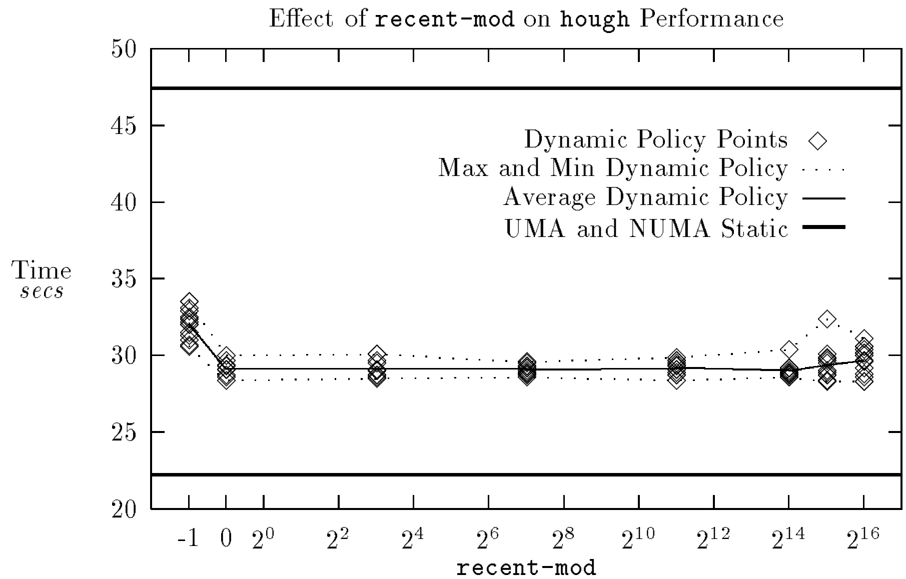

Figure 3: Effects of `recent-mod` on `hough`/TC2000 Performance

(`freeze-window` $= 31.25ms$, `scan-delay` $= 10s$, `sample-passes` $= 0$, `defrost-trigger` $= 1$, `trigger-method` = `recent-mod`)

ing `recent-mod` values is not nearly as great as in the GP1000 case. In fact, when replication is always preferred over migration (`recent-mod` $= 2^{16}$), some performance degradation is evident. This is due to the fact that relatively speaking, coherency faults are more expensive on a TC2000 than on a GP1000. The absolute time to process a coherency fault is greater on the GP1000, but the TC2000 is able to do a significant amount more processing in the time it takes for a coherency fault to complete. Since coherency faults are relatively more expensive, the cost associated with mistakenly replicating a page rather than migrating it is more important, and a more conservative approach to selecting replication (i.e., `recent-mod` $< 2^{16}$) is more appropriate. In an architecture in which the relative cost of processing a coherency fault is even greater, the intermediate `recent-mod` values may be essential to obtaining the best performance.

The experiments varying the `recent-mod` parameter for the rest of our workload on the TC2000 yielded qualitatively similar results to those obtained on the GP1000 (absolute performance changed since the TC2000 is a much faster machine). One might anticipate that an increasing preference for replication over migration would deliver increased benefits since the remote to cache access time improvement supplants the more modest remote to local improvement possible with migration. It appears, however, that the overhead of additional coherency faults can overwhelm that effect.

4.2 Effect of Varying `freeze-window`

The `freeze-window` parameter essentially controls the imposition of freezing to limit the amount of dynamic page placement activity. When `freeze-window` is set to zero, there is no limit on the frequency of page migrations and coherency faults, and for most of our applications on both machines, the page bouncing problem sets in, resulting in incredibly poor performance. In order to prevent such situations, higher `freeze-window` values must be used. However, if `freeze-window` is set too high, the prevention or delay (until defrost) of desirable migrations and replications becomes a real possibility.

The results of our `freeze-window` experiments with `psolu` on a TC2000, shown in Figure 4, are typical. The plot shows that performance with lower `freeze-window` values suffers relative to higher values. If we let `freeze-window` go to zero, performance degrades to the point that we have never been able to let the computation complete. An important characteristic of the `psolu` `freeze-window` results is that once the `freeze-window` setting is "high enough," further increases have little effect on performance. This is true of the `hough`, `gauss`, `hh3d`, and `msort` results as well, on both the GP1000 and the TC2000. This suggests that the potential problem of delaying desirable operations is not a concern for these applications on these architectures, either because the effects of such delays are negligible (e.g., delays are short since defrost comes soon), or because there are few such operations. We suspect that a combination of

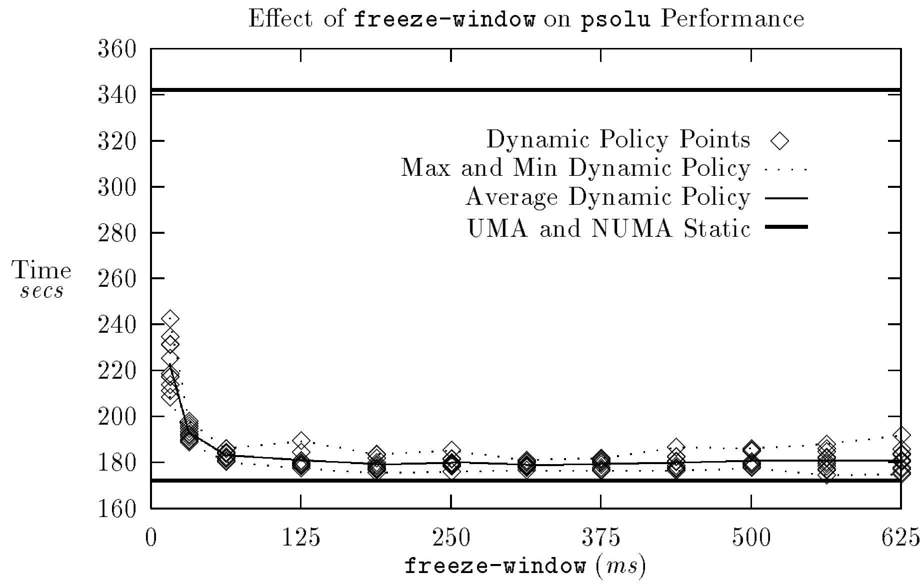

Figure 4: Effects of `freeze-window` on `psolu`/TC2000 Performance

(`scan-delay` $= 10s$, `recent-mod` $= 64k$, `sample-passes` $= 0$, `defrost-trigger` $= 1$, `trigger-method` $= 64k$)

the two factors is the reason.

The results of our GP1000 `freeze-window` experiments with the `wave` application differ and are shown in Figure 5. We see that we can easily pick a "best" `freeze-window` setting for `wave` running under the DUnX parameterized policy on a GP1000. Values lower than the minima (around $250ms$) allow too much dynamic placement activity, whereas higher values prevent desirable migrations and replications. This result indicates that when performance is best, at least some shared pages are migrated or replicated more than once before being frozen, since otherwise performance would not degrade with further increases in `freeze-window`. Even though this application exhibits a best setting for `freeze-window`, the wide range of acceptable values suggests that the task of tuning to an appropriate `freeze-window` value is not exceptionally difficult. In fact, a range of `freeze-window` settings from about $100ms$ to $300ms$ work well for any of our six applications on either architecture. This is at first somewhat surprising since the TC2000 is so much faster than the GP1000, however, it is apparent that `freeze-window` is not so sensitive to architectural differences to necessitate significant tuning changes for the degree of difference that exists between the two machines.

Behavior of the `wave` application on the GP1000 with respect to the `freeze-window` parameter provides an interesting opportunity to study architectural dependencies on `freeze-window` parameter settings, since we can see whether and how the `freeze-window` minima shifts with changes in the architecture. To study the effects of varying the costs of migrations, replications, and coherency faults while holding the memory reference costs constant, we use a simple modification to the DUnX kernel that allowed us to introduce an arbitrary amount of delay into the handling of such operations. Though this does not provide the ability to consider systems in which these operations are faster, it does let us investigate hypothetical systems in which they are slower. For example, consider the data of Figure 6 where five curves are plotted representing the average performance of `wave` in each of five hypothetical systems (experimentally modeled using our GP1000 DUnX modifications). The best overall performance is obtained with the basic DUnX GP1000 costs (i.e., replication (R) cost of $4.6ms$, migration (M) cost of $4.5ms$, and coherency fault (C) cost of $2.1ms$). As these basic costs are increased, we see a slight shift in the `freeze-window` minima towards higher values, as well as a "flattening" of the curves. The shift of the minima is not at all surprising, since as the costs of dynamic placement operations increases, one would expect that more conservative control over dynamic placement would be appropriate. The "flattening" of the `freeze-window` curves is a result of the fact that some dynamic operations prevented by higher `freeze-window` values when operation costs are low become undesirable as those operation costs increase. The number of references needed to justify the dynamic operations (a in equation (1)) increases as the costs for those operations is increased. For example, plugging

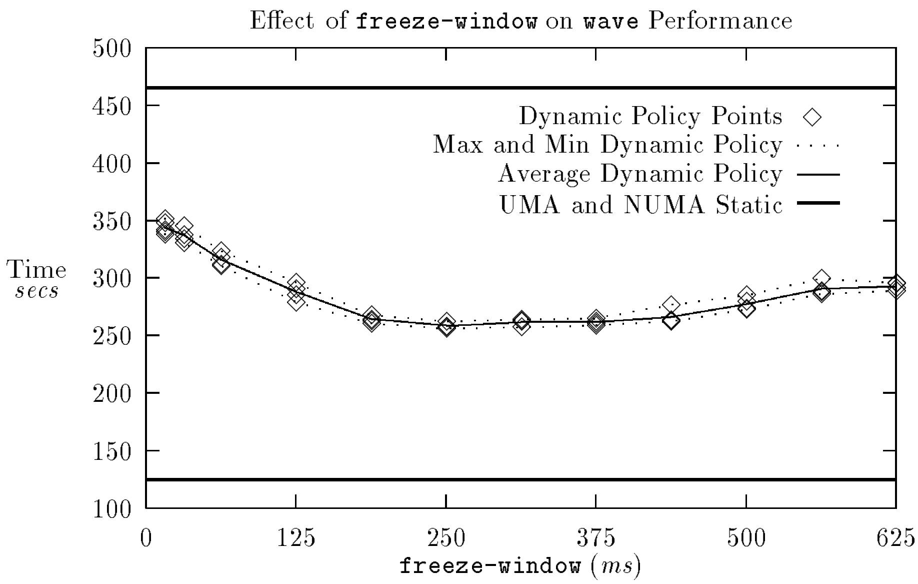

Figure 5: Effects of `freeze-window` on `wave`/GP1000 Performance

(`scan-delay` $= 10s$, `recent-mod` $= 64k$, `sample-passes` $= 10$, `defrost-trigger` $= 4$, `trigger-method` $= 64k$)

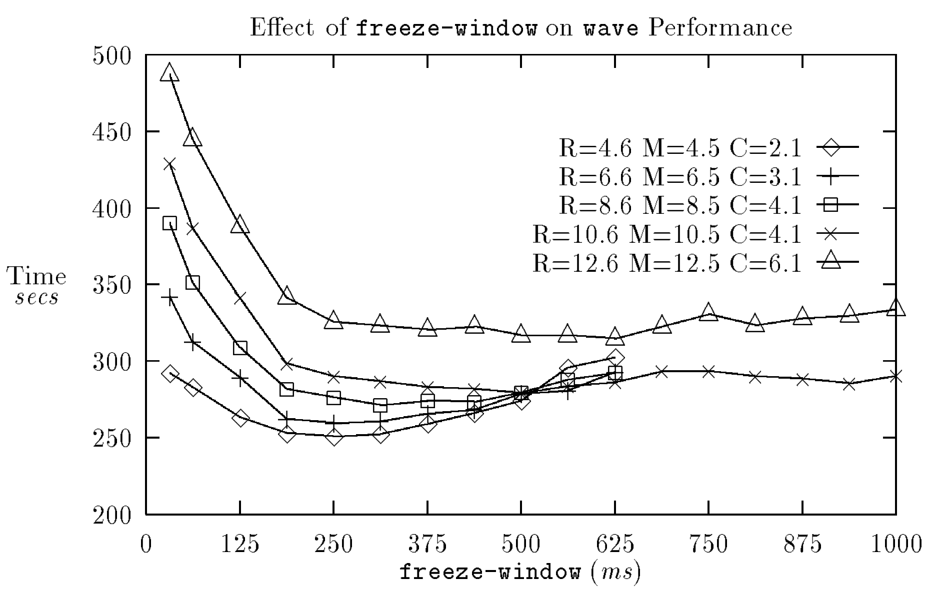

Figure 6: Effects of `freeze-window` and Dynamic Operation Times on `wave` Performance

(`scan-delay` $= 10s$, `recent-mod` $= 64k$, `sample-passes` $= 10$, `defrost-trigger` $= 4$, `trigger-method` $= 64k$)

the cost factors used in the experiments of Figure 6 into equation (1), we see that the minimum a value needed to justify a migration increases from 666, to 956, 1246, 1536, and finally to 1826. An interesting question concerns how decreasing the basic operation costs would affect performance. If the trend in Figure 6 continued in the opposite direction, it would indicate that as those costs decreased, tuning of the `freeze-window` parameter for the `wave` application might play a more significant role than in our base system.

Performance of the `wave` application on the TC2000 with respect to the `freeze-window` parameter might have been predicted from the Figure 6 data, since for the TC2000, the minimum a for which migration would be justified is 3798, which is greater than any of those considered in the Figure 6 experiments. Based on this, one would expect the performance of `wave` on the TC2000 to resemble that of our other applications, such as the `psolu` results shown in Figure 4. As expected, the results of the TC2000 `wave` `freeze-window` experiments, not included here due to space limitations, are similar to the `psolu` results.

4.3 Effect of Varying `scan-delay`

The rate at which the page scanners are run is controlled by the `scan-delay` parameter. An important function of the scanners is to defrost frozen pages. Since defrosting a page is only desirable when migrating or replicating that page would improve performance, the need to defrost a correctly frozen page is likely to come only after some sort of memory reference phase change. Thus, the appropriate `scan-delay` setting for an application depends partially on the phase change behavior of the application.[2] If `scan-delay` is set very low (with reasonable `sample-passes` and `defrost-trigger` settings), defrost operations may occur more frequently than desired, causing additional overhead in the form of extra page faults, migrations, replications, coherency faults, and freezing operations. Low `scan-delay` values also incur more overhead just running the scanners. On the other hand, if `scan-delay` is set too high, a page correctly frozen in one phase of an application may end up incorrectly placed for a long time in later program phases.

For some application/architecture combinations, such as `hh3d` on a GP1000 (Figure 7), and to a lesser extent, `hh3d` on a TC2000, performance is best when `scan-delay` $= \infty$. Note that in the plots in which `scan-delay` is varied, the final point along the x-axis is infinity. The data in Figure 7 show that small `scan-delay` values can be costly, and that at higher values, performance is significantly better. For most of our other applications on either machine, the savings associated with not using the scanners (i.e., setting `scan-delay` $= \infty$) are not as great as in the GP1000 `hh3d` case. For example, Figure 8 shows the effects of `scan-delay` on the `hough` application on a TC2000.

Only one of our application/architecture combinations, `hough` on a GP1000, performs better with the scanners than without. This is shown in Figure 9, where `scan-delay` values around $12s$ give the best performance. This is an example in which defrosting is necessary to achieve the best performance possible.

The results of our `scan-delay` experiments indicate that scanner overhead is an important factor, but that the overhead can be minimized by using larger `scan-delay` values. For at least one application/architecture combination (`hough` on a GP1000), the scanners provide a beneficial service. We expect that there exist many other real applications for which this is true on both architectures, each with its own optimal `scan-delay` value, though our results do not support this intuition.

The data of Figures 8 and 9 are interesting in another way. In Figure 9, we note that using the parameterized policy on the GP1000, performance of the UMA version of `hough` is better than that obtained with the hand-tuned NUMA version of `hough`, whereas on the TC2000 (Figure 8), this is not the case. This is because on the GP1000, the costs associated with dynamically achieving better page placements through migration and replication are low enough that the benefits outweigh those costs. On the TC2000, the costs are relatively higher so the overall benefit of the improved placements are not as great. The NUMA version of `hough` may not have the best placement, but the (static) placement it has is extremely cheap to obtain. We suspect that if the TC2000 had support for page-size block transfers, thus decreasing the costs of migrations and replications, performance of our policy in DUnX would be better than the NUMA version just as it is on the GP1000.

4.4 Other Parameters

The `sample-passes` parameter also plays a role in controlling the rate at which defrost operations are possible. Intuitively, higher `sample-passes` values may delay some desirable defrost operations, but with a constant `scan-delay` value, the accuracy of the reference information used to make decisions about defrosting pages is better (due to more samples of the hardware reference bits). The greater number of samples used when making defrost decisions should improve the selection of pages to defrost and result in fewer unwanted defrosts. Higher `sample-passes` values should provide enhanced performance for applications that exhibit active sharing, yet have identifiable predominate users of each page.

[2] In a recent paper [10] we demonstrated that "phase change hints" inserted by the programmer or compiler can also be used to trigger the defrosting of frozen pages.

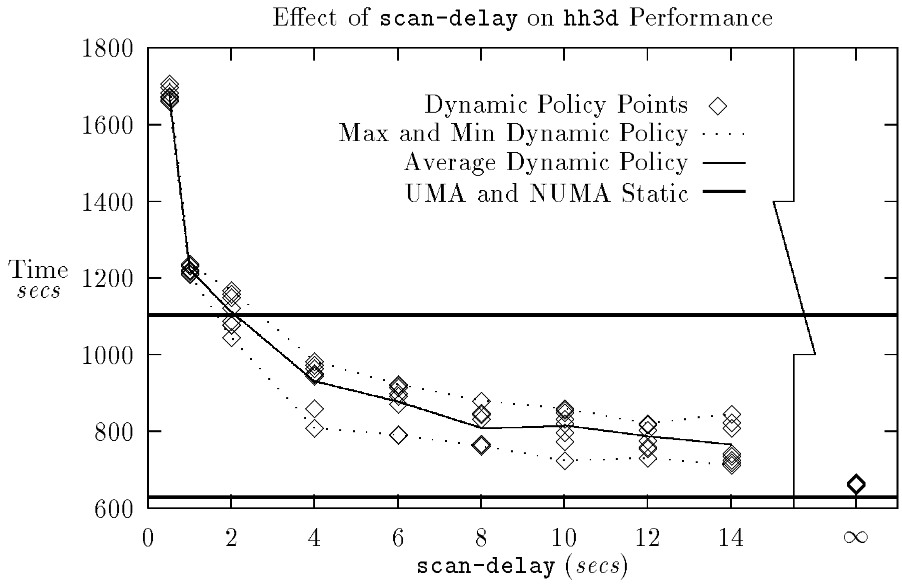

Figure 7: Effects of `scan-delay` on `hh3d`/GP1000 Performance

(`freeze-window` $= 312.5ms$, `recent-mod` $= 64k$, `sample-passes` $= 0$, `defrost-trigger` $= 4$, `trigger-method` $= 64k$)

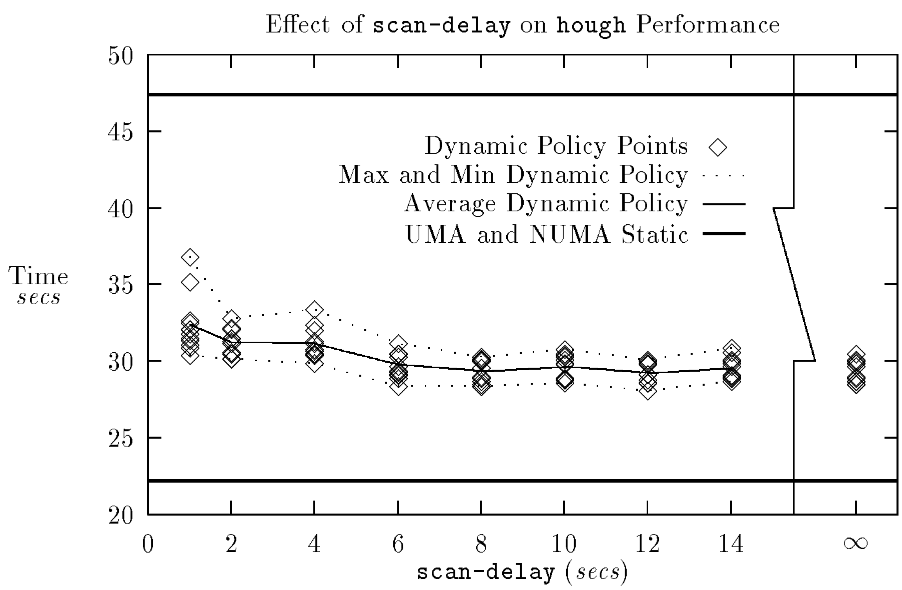

Figure 8: Effects of `scan-delay` on `hough`/TC2000 Performance

(`freeze-window` $= 312.5ms$, `recent-mod` $= 64k$, `sample` $-$ `passes` $= 10$, `defrost-trigger` $= 4$, `trigger-method` $= 64k$)

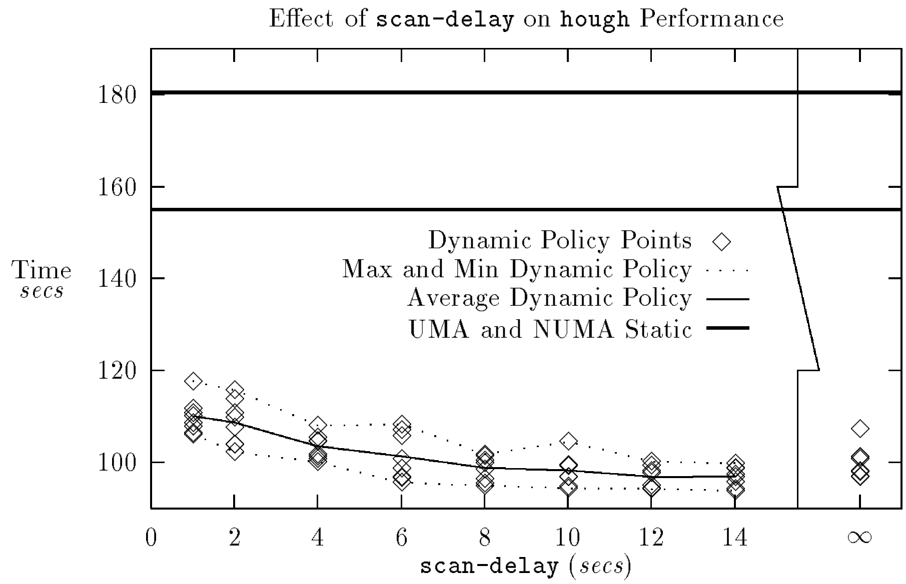

Figure 9: Effects of `scan-delay` on `hough`/GP1000 Performance

(`freeze-window` $= 31.25ms$, `recent-mod` $= 64k$, `sample-passes` $= 0$, `defrost-trigger` $= 1$, `trigger-method` $= 64k$)

In our experiments, we have looked at `sample-passes` values ranging from zero to ten. For our application suite, varying `sample-passes` appears to have only minor effects on performance on either the GP1000 or the TC2000. These results suggest that there may be little use in continuing to support the `sample-passes` parameter.

The defrost decision-making process is partially controlled by the `defrost-trigger` parameter. When `defrost-trigger` $= 0$, reference information is not used to make defrost decisions; the scanners simply defrost every frozen page. Higher `defrost-trigger` values are more conservative, deciding to defrost only when it appears very clear that a migration or replication of the page is appropriate.

In our experiments, we considered `defrost-trigger` values ranging from zero to seven. For nearly all of our applications, the `defrost-trigger` parameter had only minor effects on performance on either architecture, generally resulting in slightly better performance for higher values but rarely resulting in a major performance improvement.

Similarly, maintaining a separate `trigger-method` parameter, decoupled from the `recent-mod` value, can not be justified by our data.

4.5 General-Purpose Settings

The results of the experiments presented thus far in this paper suggest that tuning of our policy parameters is not as important as we initially expected. While it is possible to choose especially bad settings for some of the parameters (e.g., `freeze-window` and `recent-mod`), our results suggest that a wide range of possible settings would likely prove acceptable for any of the applications on either architecture.

To test this hypothesis, we selected a set of default parameter values based on our experimental results. We compared performance of our six applications with their respective tuned parameter settings to the performance obtained with the default settings selected when run on a GP1000. The default parameter settings we selected are as follows: `freeze-window` $= 150ms$, `recent-mod` $= 2^{16}$, `scan-delay` $= 10s$, `sample-passes` $= 0$, `defrost-trigger` $= 1$, and `trigger-method` $= 2^{16}$. We should note that a different version of the DUnX kernel was used for this experiment than for the results presented previously, so the numbers reported in this subsection should not be directly compared to earlier results. We used the same tuned policy parameter settings reported in Table 3.

The results are reported in Table 4. For each of the two parameter settings (the application's tuned values and the system defaults), we report the upper and lower bounds of a 99% confidence interval determined using the Student-t distribution and a sample size of twenty. The mean value is also reported for each case. Finally, in the rightmost column of the table, we report the percentage improvement (calculated with reference to the

Program	*Tuned*			*Default*			*%Improve*
	Lower	*Mean*	*Upper*	*Lower*	*Mean*	*Upper*	
`gauss`	220.033s	234.12s	248.207s	232.735	248.795	264.855	5.9%
`hh3d`	772.688s	787.53s	802.372s	788.799	800.81	812.821	1.6%
`psolu`	565.906s	569.53s	573.154s	561.23	564.86	568.49	-0.8%
`hough`	100.539s	101.1s	101.661s	99.9675	100.225	100.482	-0.9%
`msort`	72.9939s	75.235s	77.4761s	77.6131	78.515	79.4169	4.2%
`wave`	267.083s	276.57s	286.057s	264.901	267.75	270.599	-3.3%

Table 4: Comparison of Default and Tuned Performance.

mean values) obtained through application-specific policy tuning.

The first important point to notice about the results is that in all but two cases (`hough` and `msort`) the confidence intervals overlap, indicating that the differences are statistically insignificant. Thus it is clear that for the other four applications, per-application tuning provides little additional benefit. Even for the two applications for which the differences are statistically significant, the differences are quite small. The improved performance of `msort` with the tuned settings could have been predicted based on the fact that `msort` performed best when `recent-mod` is negative. However, the improvement of just 4.2% is not overwhelming. The `hough` results indicate that performance is slightly *better* with the default settings than with the tuned settings. This is an artifact of our ad hoc tuning process, in which in this case, we selected parameter values that are not quite optimal.

In any case, the results of this subsection demonstrate the primary conclusion of this paper. NUMA memory management is robust. Even with a highly parameterized policy, a single set of default parameter values can easily come within 5% of the tuned performance.

5 Conclusions

Both intuition and previous studies have suggested that NUMA memory management policy depends on the memory reference patterns of applications and on the target architectures. We have attempted to experimentally explore these dependencies, determine which of the many possible factors that might affect behavior are the most important, and assess the potential utility of policy tuning as a way to accommodate different memory features and workloads. To investigate these issues of policy tuning, we developed a version of the DUnX kernel for the BBN GP1000 and TC2000 multiprocessors that supports a highly parameterized policy.

Our experiments have provided evidence to support the following conclusions:

1. The results again confirm the effectiveness of dynamic placement policies, shown by previous studies, in that the measured performance of the UMA versions of the workload programs running with appropriate tunings often approaches the performance of the hand-tuned NUMA versions.

2. A policy as highly parameterized as the one used in our experiments does not appear necessary. The most interesting parameters appear to be `recent-mod` and `freeze-window`. In setting `recent-mod`, the most important consideration is to enable some amount of replication. For `freeze-window`, the setting must be high enough to avoid bouncing. Beyond those concerns, there are a wide range of acceptable values from which to choose. Several of the parameters have only a negligible effect on performance and can be dropped from the list.

3. The data indicate that our NUMA memory management policy is robust. We have shown that it is easy to find a "general-purpose" set of parameter values for the GP1000 that delivers good performance across our entire suite of test programs. Although experiments with the system default settings were not also performed on the TC2000, the similarity of other GP1000 and TC2000 results suggests that a single set of default tunings may be successfully applied to that architecture as well.

Our experience leads us to believe that a reasonably simple parameterized policy may form the basis for the development of machine-independent memory management subsystems for the class of Local/Remote NUMA machines.

Acknowledgements

The authors wish to thank the SOSP program committee and outside reviewers for their helpful suggestions for improving this paper.

References

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

[2] David Black and Daniel Sleator. Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Carnegie-Mellon University, November 1989.

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

[4] W. Bolosky, M. Scott, R. Fitzgerald, R. Fowler, 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] 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.

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

[7] 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.

[8] R. P. LaRowe Jr. *Page Placement for Nonuniform Memory Access Time (NUMA) Shared Memory Multiprocessors.* PhD thesis, Duke University, 1991.

[9] R. P. LaRowe Jr. and C. S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. Technical Report CS-1990-10, Duke University, April 1990. To Appear in ACM Transactions on Computer Systems.

[10] R. P. LaRowe Jr., J. T. Wilkes, and C. S. Ellis. Exploiting operating system support for dynamic page placement on a NUMA shared memory multiprocessor. In *Proceedings of the Symposium on the Principles and Practice of Parallel Programming*, pages 122–132, April 1991.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, 1986.

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

[13] J. Ramanathan and L. M. Ni. Critical factors in NUMA memory management. In *Proceedings of the Eleventh International Conference on Distributed Computer Systems*, may 1991.

[14] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proc. ASPLOS-II*, October 1987.