A Case for NUMA-aware Contention Management on Multicore Systems

Sergey Blagodurov
Simon Fraser University

Sergey Zhuravlev
Simon Fraser University

Mohammad Dashti
Simon Fraser University

Alexandra Fedorova
Simon Fraser University

Abstract

On multicore systems, contention for shared resources occurs when memory-intensive threads are co-scheduled on cores that share parts of the memory hierarchy, such as last-level caches and memory controllers. Previous work investigated how contention could be addressed via scheduling. A contention-aware scheduler separates competing threads onto separate memory hierarchy domains to eliminate resource sharing and, as a consequence, to mitigate contention. However, all previous work on contention-aware scheduling assumed that the underlying system is UMA (uniform memory access latencies, single memory controller). Modern multicore systems, however, are NUMA, which means that they feature non-uniform memory access latencies and multiple memory controllers.

We discovered that state-of-the-art contention management algorithms fail to be effective on NUMA systems and may even *hurt* performance relative to a default OS scheduler. In this paper we investigate the causes for this behavior and design the first contention-aware algorithm for NUMA systems.

1 Introduction

Contention for shared resources on multicore processors is a well-known problem. Consider a typical multicore system, schematically depicted in Figure 1, where cores share parts of the memory hierarchy, which we term *memory domains*, and compete for resources such as last-level caches (LLC), system request queues and memory controllers. Several studies investigated ways of reducing resource contention and one of the promising approaches that emerged recently is contention-aware scheduling [23, 10, 16]. A contention-aware scheduler identifies threads that compete for shared resources of a memory domain and places them into different domains. In doing so the scheduler can improve the worst-case performance of individual applications or threads by as much as 80% and the overall workload performance by as much as 12% [23].

Unfortunately studies of contention-aware algorithms focused primarily on UMA (Uniform Memory Access) systems, where there are multiple shared LLCs, but only a single memory node equipped with the single memory controller, and memory can be accessed with the same latency from any core. However, new multicore systems increasingly use the Non-Uniform Memory Access (NUMA) architecture, due to its decentralized and scalable nature. In modern NUMA systems, there are multiple memory nodes, one per memory domain (see Figure 1). Local nodes can be accessed in less time than remote ones, and each node has its own memory controller. When we ran the best known contention-aware schedulers on a NUMA system, we discovered that not only do they not manage contention effectively, but they sometimes even *hurt performance* when compared to a default contention-unaware scheduler (on our experimental setup we observed as much as 30% performance degradation caused by a NUMA-agnostic contention-aware algorithm relative to the default Linux scheduler). The focus of our study is to investigate (1) why contention-management schedulers that targeted UMA systems fail to work on NUMA systems and (2) devise an algorithm that would work effectively on NUMA systems.

Why existing contention-aware algorithms may hurt performance on NUMA systems: Existing state-of-the-art contention-aware algorithms work as follows on NUMA systems. They identify threads that are sharing a memory domain and hurting each other's performance and migrate one of the threads to a different domain. This may lead to a situation where a thread's memory is located in a different domain than that in which the thread is running. (E.g., consider a thread being migrated from core C1 to core C5 in Figure 1, with its memory being located in Memory Node #1). We refer to migrations that may place a thread into a domain remote from its mem-

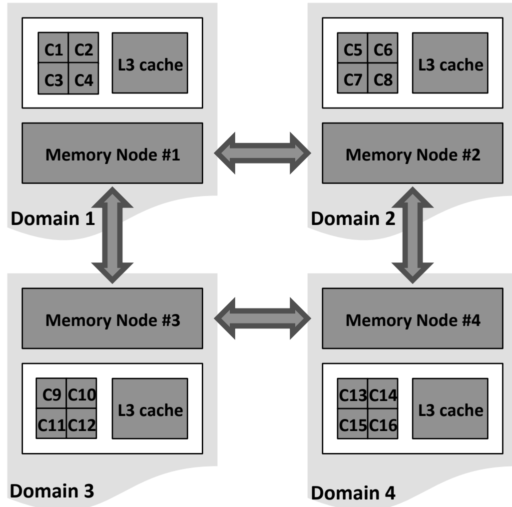


Figure 1: A schematic view of a system with four memory domains and four cores per domain. There are 16 cores in total, and a shared L3 cache per domain.

ory *NUMA-agnostic migrations*.

NUMA-agnostic migrations create several problems, an obvious one being that the thread now incurs a higher latency when accessing its memory. However, contrary to a commonly held belief that remote access latency – i.e., the higher latency incurred when accessing a remote domain relative to accessing a local one – would be the key concern in this scenario, we discovered that NUMA-agnostic migrations create other problems, which are far more serious than remote access latency. In particular, NUMA-agnostic migrations fail to eliminate contention for some of the key hardware resources on multicore systems and create contention for additional resources. That is why existing contention-aware algorithms that perform NUMA-agnostic migrations not only fail to be effective, but can substantially hurt performance on modern multicore systems.

Challenges in designing contention-aware algorithms for NUMA systems: To address this problem, a contention-aware algorithm on a NUMA system must migrate the memory of the thread to the same domain where it migrates the thread itself. However, the need to move memory along with the thread makes thread migrations costly. So the algorithm must minimize thread migrations, performing them only when they are likely to significantly increase performance, and when migrating memory it must carefully decide which pages are most profitable to migrate. Our work addresses these challenges.

The contributions of our work can be summarized as follows:

- We discover that contention-aware algorithms known to work well on UMA systems may actually *hurt* performance on NUMA systems.
- We identify NUMA-agnostic migration as the cause for this phenomenon and identify the reasons why performance degrades. We also show that remote access latency is not the key reason why NUMA-agnostic migration hurt performance.
- We design and implement *Distributed Intensity NUMA Online* (DINO), a new contention-aware algorithm for NUMA systems. DINO prevents superfluous thread migrations, but when it does perform migrations, it moves the memory of the threads along with the threads themselves. DINO performs up to 20% better than the default Linux scheduler and up to 50% better than Distributed Intensity, which is the best contention-aware scheduler known to us [23].
- We devise a page migration strategy that works online, uses Instruction-Based Sampling, and eliminates on average 75% of remote accesses.

Our algorithms were implemented at user-level, since modern operating systems typically export the interfaces for implementing the desired functionality. If needed, the algorithms can also be moved into the kernel itself.

The rest of this paper is organized as follows. Section 2 demonstrates why existing contention-aware algorithms fail to work on NUMA systems. Section 3 presents and evaluates DINO. Section 4 analyzes memory migration strategies. Section 5 provides the experimental results. Section 6 discusses related work, and Section 7 summarizes our findings.

2 Why existing algorithms do not work on NUMA systems

As we explained in the introduction, existing contention-aware algorithms perform NUMA-agnostic migration, and so a thread may end up running on a node remote from its memory. This creates additional problems besides introducing remote latency overhead. In particular, NUMA-agnostic migrations fail to eliminate *memory controller contention*, and create additional *interconnect contention*. The focus of this section is to experimentally demonstrate why this is the case.

To this end, in Section 2.1, we quantify how contention for various shared resources contributes to performance degradation that an application may experience as it shares the hardware with other applications. We show that memory controller contention and interconnect contention are the most important causes of performance degradation when an application is running remotely from its memory. Then, in Section 2.2 we use

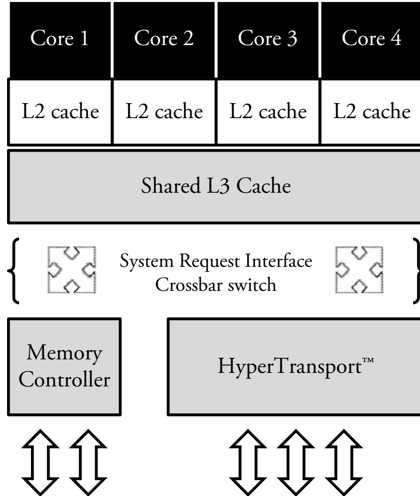


Figure 2: A schematic view of a system used in this study. A single domain is shown.

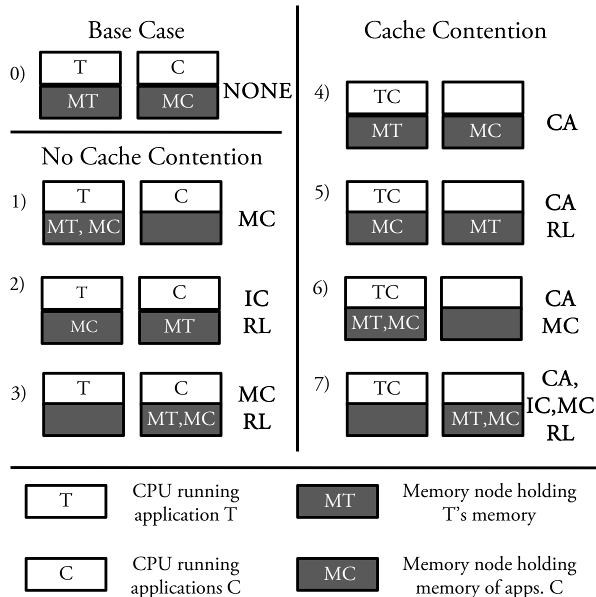


Figure 3: Placement of threads and memory in all experimental configurations.

this finding to explain why NUMA-agnostic migrations can be detrimental to performance.

2.1 Quantifying causes of contention

In this section we quantify the effects of performance degradation on multicore NUMA systems depending on how threads and their memory are placed in memory domains. For this part of the study, we use benchmarks from the SPEC CPU2006 benchmark suite. We perform experiments on a Dell PowerEdge server equipped with four AMD Barcelona processors running at 2.3GHz, and 64GB of RAM, 16GB per domain. The operating system is Linux 2.6.29.6. Figure 2 schematically represents the architecture of each processor in this system.

We identify four sources of performance degradation that can occur on modern NUMA systems, such as those shown in Figures 1 and 2:

- Contention for the shared last-level cache (CA). This also includes contention for the system request queue and the crossbar.
- Contention for the memory controller (MC). This also includes contention for the DRAM prefetching unit.
- Contention for the inter-domain interconnect (IC).
- Remote access latency, occurring when a thread's memory is placed in a remote node (RL).

To quantify the effects of performance degradation caused by these factors we use the methodology depicted in Figure 3. We run a target application, denoted as T with a set of three competing applications, denoted as C. The memory of the target application is denoted MT, and the memory of the competing applications is denoted MC. We vary (1) how the target application is placed with respect to its memory, (2) how it is placed with respect to the competing applications, and (3) how the memory of the target is placed with respect to the memory of the competing applications. Exploring performance in these various scenarios allows us to quantify the effects of NUMA-agnostic thread placement.

Figure 3 summarizes the relative placement of memory and applications that we used in our experiments. Next to each scenario we show factors affecting the performance of the target application: CA, IC, MC or RL. For example, in Scenario 0, an application runs contention-free with its memory on a local node, so no performance-degrading factors are present. We term this the *base* case and compare to it the performance in other cases. The scenarios where there is cache contention are shown on the right and the scenarios where there is no cache contention are shown on the left.

We used two types of target and competing applications, classified according to their memory intensity: *devil* and *turtle*. The terminology is borrowed from an earlier study on application classification [21]. Devils are memory intensive: they generate a large number of memory requests. We classify an application as a devil if it generates more than two misses per 1000 instructions (MPI). Otherwise, an application is deemed a turtle. We further divide devils into two subcategories: *regular devils* and *soft-devils*. Regular devils have a miss rate that exceeds 15 misses per 1000 instructions. Soft-devils

have an MPI between two and 15. Solo miss rates, obtained when an application runs on a machine alone, are used for classification.

We experimented with nine different target applications: three devils (*mcf*, *omnetpp* and *milc*), three soft-devils, (*gcc*, *bwaves* and *bzip*) and three turtles (*povray*, *calculix* and *h264*).

Figure 4 shows how an application's performance degrades in Scenarios 1-7 from Figure 3 relative to Scenario 0. Performance degradation, shown on the y-axis, is measured as the increase in completion time relative to Scenario 0. The x-axis shows the type of competing applications that were running concurrently to generate contention: devil, soft-devil, or turtle.

These results demonstrate a very important point exhibited in Scenario 3: when a thread runs alone on a memory node (i.e., there is no contention for cache), but its memory is remote and is in the same domain as the memory of another memory-intensive thread, performance degradation can be very severe, reaching 110% (see MILC, Scenario 3). One of the reasons is that the threads are still competing for the *memory controller* of the node that holds their memory. But this is exactly the scenario that can be created by a NUMA-agnostic migration, which migrates a thread to a different node without migrating its memory. This is the first piece of evidence showing why NUMA-agnostic migrations will cause problems.

We now present further evidence. Using the data in these experiments, we are able to estimate how much each of the four factors (CA, MC, IC, and RL) contributes to the overall performance degradation in Scenario 7 – the one where performance degradation is the worst. For that, we compare experiments that differ from each other precisely by one degradation factor involved. This allows us to single out the influence of this differentiating factor on the application performance. Figure 5 shows the breakdown for the devil and soft-devil applications. Turtles are not shown, because their performance degradation is negligible. The overall degradation for each application relative to the base case is shown at the top of the corresponding bar. The y-axis shows the fraction of the total performance degradation that each factor causes. Since contention causing factors on a real system overlap in complex and integrated ways, it is not possible to obtain a precise separation. These results are an approximation that is intended to direct attention to the true bottlenecks in the system.

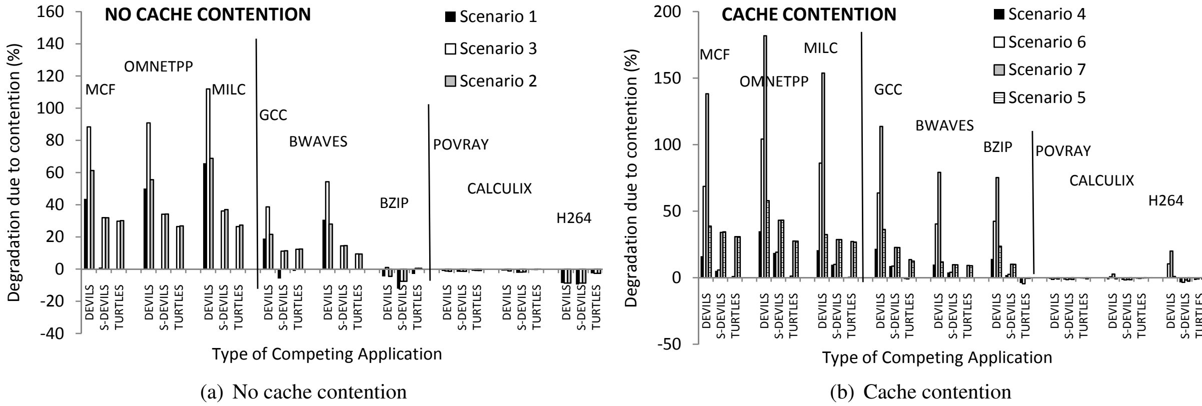
The results show that of all performance-degrading factors contention for cache constitutes only a very small part, contributing at most 20% to the overall degradation. And yet, NUMA-agnostic migrations eliminate only contention for the shared cache (CA), leaving the more important factors (MC, IC, RL) unaddressed! Since the memory is not migrated with the thread, several memory-intensive threads could still have their memory placed in the same memory node and so they would compete for the memory controller when accessing their memory. Furthermore, a migrated thread could be subject to the remote access latency, and because a thread would use the inter-node interconnect to access its memory, it would be subject to the interconnect contention. In summary, NUMA-agnostic migrations fail to eliminate or even exacerbate the most crucial performance-degrading factors: MC, IC, RL.

2.2 Why existing contention management algorithms hurt performance

Now that we are familiar with causes of performance degradation on NUMA systems, we are ready to explain why existing contention management algorithms fail to work on NUMA systems. Consider the following example. Suppose that two competing threads A and B run on cores C1 and C2 on a system shown in Figure 1. A contention-aware scheduler would detect that A and B compete and migrate one of the threads, for example thread B, to a core in a different memory domain, for example core C5. Now A and B are not competing for the last-level (L3) cache, and on UMA systems this would be sufficient to eliminate contention. But on a NUMA system shown in Figure 1, A and B are still competing for the memory controller at Memory Node #1 (MC in Figure 5), assuming that their memory is physically located in Node #1. So by simply migrating thread B to another memory domain, the scheduler does not eliminate one of the most significant sources of contention – contention for the memory controller.

Furthermore, the migration of thread B to a different memory domain creates two additional problems, which degrade thread B's performance. Assuming that thread B's memory is physically located in Memory Node #1 (all operating systems of which we are aware would allocate B's memory on Node #1 if B is running on a core attached to Node #1 and then leave the memory on Node #1 even after thread migration), B is now suffering from two additional sources of overhead: interconnect contention and remote latency (labeled IC and RL respectively in Figure 5). Although remote latency is not a crucially important factor, interconnect contention could hurt performance quite significantly.

To summarize, NUMA-agnostic migrations in the existing contention management algorithms cause the following problems, listed in the order of severity according to their effect on performance: (1) They fail to eliminate memory-controller contention; (2) They may create additional interconnect contention; (3) They introduce remote latency overhead.



(a) No cache contention

(b) Cache contention

Figure 4: Performance degradation due to contention, cases 1-7 from Figure 3 relative to running contention free (case 0).

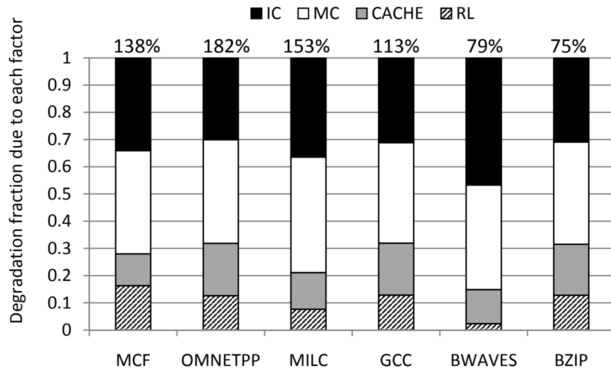


Figure 5: Contribution of each factor to the worst-case performance degradation.

3 A Contention-Aware Scheduling Algorithm for NUMA Systems

We design a new contention-aware scheduling algorithm for NUMA systems. We borrow the contention-modeling heuristic from the *Distributed Intensity* (DI) algorithm, because it was shown to perform within 3% percent of optimal on non-NUMA systems[1] [23]. Other contention aware algorithms use similar principles as DI [10, 16].

We begin by explaining how the original DI algorithm works (Section 3.1). For clarity we will refer to it from now on as *DI-Plain*. We proceed to show that simply extending DI-Plain to migrate memory – this version of the algorithm is called *DI-Migrate* – is not sufficient to achieve good performance on NUMA systems. We conclude with the description of our new *DI-NUMA Online*, or DINO, that in addition to migrating thread memory along with the thread eliminates superfluous migrations and unlike other algorithms improves performance on NUMA systems.

[1] Although some experiments with DI reported in [23] were performed on a NUMA machine, the experimental environment was configured so as to eliminate any effects of NUMA.

3.1 DI-Plain

DI-Plain works by predicting which threads will interfere if co-scheduled on the same memory domain and placing those threads on separate domains. Prediction is performed online, based on performance characteristics of threads measured via hardware counters. To predict interference, DI uses the *miss-rate heuristic* – a measure of last-level cache misses per thousand instructions, which includes the misses resulting from hardware prefetch requests. As we and other researchers showed in earlier work the miss-rate heuristic is a good approximation of contention: if two threads have a high LLC miss rate they are likely to compete for shared CPU resources and degrade each other's performance [23, 2, 10, 16].

Even though the miss rate does not capture the full complexity of thread interactions on modern multicore systems, it is an excellent predictor of contention for memory controllers and interconnects – key resource bottlenecks on these systems – because it reflects how intensely threads use these resources. Detailed study showing why the miss rate heuristic works well and how it compares to other modeling heuristics is reported in [23, 2].

DI-Plain continuously monitors the miss rates of running threads. Once in a while (every second in the original implementation), it sorts the threads according to their miss rates, and assigns them to memory domains so as to co-schedule low-miss-rate threads with high-miss-rate threads. It does so by first iterating over the sorted threads starting from the most memory-intensive (the one

with the highest miss rate) and placing each thread in a separate domain, iterating over domains consecutively. This way it separates memory-intensive threads. Then it iterates over the array from the other end, starting from the least memory-intensive thread, placing each on an unused core in consecutive domains. Then it iterates from the other end of the array again, and continues alternating iterations until all threads have been placed. This strategy results in balancing the memory intensity across domains. DI-Plain performs no memory migration when it migrates the threads.

Existing operating systems (Linux, Solaris) would not move the thread's memory to another node when a thread is moved to a new domain. Linux performs new memory allocations in the new domain, but will leave the memory allocated before migration in the old one. Solaris will act similarly[2]. So on either of these systems, if the thread after migration keeps accessing the memory that was allocated on another domain, it will cause negative performance effects described in Section 2.

3.2 DI-Migrate

Our first (and obvious) attempt to make DI-Plain NUMA-aware was to make it migrate the thread's memory along with the thread. We refer to this "intermediate" algorithm in our design exploration as *DI-Migrate*. The description of the memory migration algorithm is deferred until Section 4, but the general idea is that it detects which pages are actively accessed and migrates them to the new node along with a chunk of surrounding pages. For now we present a few experiments comparing DI-Plain with DI-Migrate. Our experiments will reveal that memory migration is insufficient to make DI-Plain work well on NUMA systems, and this will motivate the design of DINO.

Our experiments were performed on the same system as described in Section 2.1.

The benchmarks shown in this section are scientific applications from SPEC CPU2006 and SPEC MPI2007 suites with reference sets in both cases. (In a later section we also show results for the multithreaded Apache/MySQL workload.) We evaluated scientific applications for two reasons. First, they are CPU-intensive and often suffer from contention. Second, they were of interest for our partner Western Canadian Research Grid (WestGrid) – a network of compute clusters used by scientists at Canadian universities and in particular by physicists involved in ATLAS, an international particle physics experiment at the Large Hadron Collider at CERN. The WestGrid site at our university is interested in deploying contention management algorithms on their clusters. Prospect of adoption of contention management algorithms in a real setting also motivated their user-level implementation – not requiring a custom kernel makes the adoption less risky. Our algorithms are implemented on Linux as user-level daemons that measure threads' miss rates using `perfmon`, migrate threads using scheduling affinity system calls, and move memory using the `numa_migrate_pages` system call.

For SPEC CPU we show one workload for brevity; complete results are presented in Section 5. All benchmarks in the workload are launched simultaneously and if one benchmark terminates it is restarted until each benchmark completes three times. We use the result of the second execution for each benchmark, and perform the experiment ten times, reporting the average of these runs.

For SPEC MPI we show results for eleven different MPI jobs. In each experiment we run a single job, each comprised of 16 processes. We perform ten runs of each job and present the average completion times.

We compare performance under DI-Plain and DI-Migrate relative to the default Linux Completely Fair Scheduler, to which we refer as Default. Standard deviation across the runs is under 6% for the DI algorithms. Deviation under Default is necessarily high, because being unaware of resource contention it may force a low-contention thread placement in one run and a high-contention mapping in another. Detailed comparison of deviations under different schedulers is also presented in Section 5.

Figures 6 and 7 show the average completion time improvement for the SPEC CPU and SPEC MPI workloads respectively (higher numbers are better) under DI algorithms relative to Default. We draw two important conclusions. First of all, DI-Plain often *hurts* performance on NUMA systems, sometimes by as much as 36%. Second, while DI-Migrate eliminates performance loss and even improves it for SPEC CPU workloads, it fails to excel with SPEC MPI workloads, hurting performance by as much as 25% for GAPgeofem.

Our investigation revealed DI-Migrate migrated processes a lot more frequently in the SPEC MPI workload than in the SPEC CPU workload. While fewer than 50 migrations per process per hour were performed for SPEC CPU workloads, but as many as 400 (per process) were performed for SPEC MPI! DI-Migrate will migrate a thread to a different core any time its miss rate (and its position in the array sorted by miss rates) changes. For the dynamic SPEC MPI workload this happened rather frequently and led to frequent migrations.

[2]Solaris will perform new allocations in the new domain if a thread's home *lgroup* – a representation of a thread's home memory domain – is reassigned upon migration, but will not move the memory allocated prior to home lgroup reassignment. If the lgroup is unchanged, even new memory allocations will be performed in the old domain.

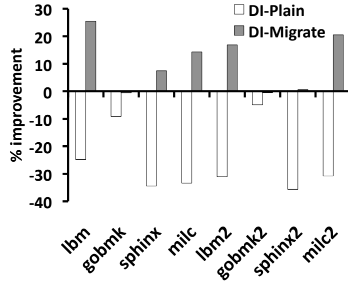


Figure 6: Improvement of completion time under DI-Plain and DI-Migrate relative to the Default for a SPEC CPU 2006 workload.

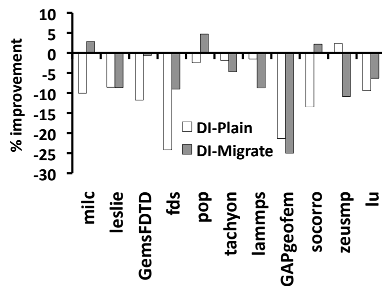


Figure 7: Improvement of completion time under DI-Plain and DI-Migrate relative to Default for eleven SPEC MPI 2007 jobs.

Unlike on UMA systems, thread migrations are not cheap on NUMA systems, because you also have to move the memory of the thread. No matter how efficient memory migrations are, they will never be completely free, so it is always worth reducing the number of migrations to the minimum, performing them only when they are likely to result in improved performance. Our analysis of DI-Migrate behaviour for the SPEC MPI workload revealed that oftentimes migrations resulted in a thread placement that was not better in terms of contention than the placement prior to migration. This invited opportunities for improvement, which we used in design of DINO.

3.3 DINO

3.3.1 Motivation

DINO's key novelty is in eliminating superfluous thread migrations – those that are not likely to reduce contention. Recall that DI-Plain (Section 3.1) triggers migrations when threads change their miss rates and their relative positions in the sorted array. Miss rates may change rather often, but we found that it is not necessary to respond to every change in order to reduce contention.

This insight comes from the observation that while the miss rate is an excellent heuristic for predicting relative contention at *coarse* granularity (and that is why it was shown to perform within 3% of the optimal oracular scheduler in DI) it does not perfectly predict how contention is affected by small changes in the miss rate.

Figure 8 illustrates this point. It shows on the x-axis SPEC CPU 2006 applications sorted in the decreasing order by their performance degradation when co-scheduled on the same domain with three instances of itself, relative to running solo. The bars show the miss rates and the line shows the degradations[3]. In general, with the exception of one outlier *mcf*, if one application has a much higher miss rate than another, it will have a much higher degradation. But if the difference in the miss rates is small, it is difficult to predict the relative difference in degradations.

What this means is that it is not necessary for the scheduler to migrate threads upon small changes in the miss rate, only upon the large ones.

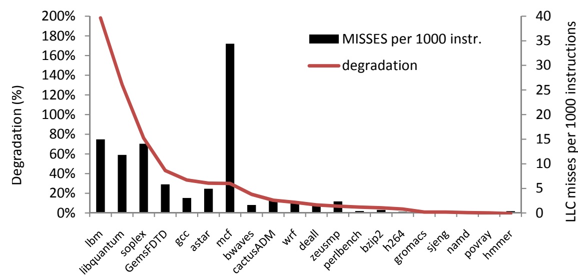


Figure 8: Performance degradation due to contention and miss rates for SPEC CPU2006 applications.

3.3.2 Thread classification in DINO and multi-threaded support

To build upon this insight, we design DINO to organize threads into broad *classes* according to their miss rates, and to perform migrations only when threads change their class, while trying to preserve thread-core affinities whenever possible. Classes are defined as follows (again, we borrow the animalistic classification from previous work):

Class 1: ***turtles*** – fewer than two LLC misses per 1000 instructions.
Class 2: ***devils*** – 2-100 LLC misses per 1000 instructions.
Class 3: ***super-devils*** – more than 100 LLC misses per 1000 instructions.

Threshold values for classes were chosen for our target architecture. Values for other architectures should be

[3] We omit several benchmarks whose counters failed to record during the experiment.

chosen by examining the relationship between the miss rates and degradations on that architecture.

Before we describe DINO in detail, we explain the special new features in DINO to deal with multithreaded applications.

First of all, DINO tries to co-schedule threads of the same application on the same memory domain, provided that this does not conflict with DINO's contention-aware assignment (described below). This assumes that performance improvement from co-operative data sharing when threads are co-scheduled on the same domain are much smaller than the negative effects of contention. This is true for many applications [22]. However, when this assumption does not hold, DINO can be extended to predict when co-scheduling threads on the same domain is more beneficial than separating them, using techniques described in [9] or [19].

When it is not possible to co-schedule all threads in an application on the same domain, and if threads actively share data, they will put pressure on memory controller and interconnects. While there is not much the scheduler can do in this situation (re-designing the application is the best alternative), it must at least avoid migrating the memory back and forth, so as not to make the performance worse. Therefore, DINO detects when the memory is being "ping-ponged" between nodes and discontinues memory migration in that case.

3.3.3 DINO algorithm description

We now explain how DINO works using an example.

In every rebalancing interval, set to one second in our implementation, DINO reads the miss rate of each thread from hardware counters. It then determines each thread's class based on its miss rate. To reduce the influence of sudden spikes, the thread only changes the class if it spent at least 7 out of the last 10 intervals with the missrate from the new class. Otherwise, the thread's class remains the same. We save this data as an array of tuples <new_class, new_processID, new_threadID>, sorted by memory-intensity of the class (e.g., super-devils, followed by devils and followed by turtles). Suppose we have a workload of eight threads containing two super-devils (D), three devils (d) and three turtles (t). Threads numbered <0, 3, 4, 5> are part of process 0. The remaining threads, numbered 1, 2, 6 and 7 each belong to a separate process, numbered 1, 2, 3 and 4 respectively [4]. Then the sorted tuple array will look like this:

```
new_class:     D D  d d d  t t t
new_processID: 0 4  0 2 3  0 0 1
new_threadID:  0 7  4 2 6  3 5 1
```

[4]DINO assigns a unique thread ID to each thread in the workload.

DINO then proceeds with the computation of *the placement layout* for the next interval. The placement layout defines how threads are placed on cores. It is computed by taking the most aggressive class instance (a super devil in our example) and placing it on a core in the first memory domain `dom0`, then the second aggressive (also a super devil) – on a core in the second domain and so on until we reach the last domain. Then we iterate from the opposite end of the array (starting with the least memory-intensive instance) and spread them across domains starting with `dom3`. We continue alternating between two ends of the array until all class instances have been placed on cores. In our example, for the NUMA machine with four memory domains and two cores per domain, the layout will be computed as follows:

```
domain:       dom0  dom1  dom2  dom3
new_core:     0 1   2 3   4 5   6 7
layout:       D t   D t   d t   d d
```

Although this example assumes that the number of threads equals the number of cores, the algorithm generalizes for scenarios when the number of threads is smaller or greater than the number of cores. In the latter case, each core will have T "slots" that can be filled with threads, where $T = num_threads/num_cores$, and instead of taking one class-instance from the array at a time, DINO will take T.

Now that we determined the layout for class-instances, we are yet to decide which thread will fill each core-class slot – any thread of the given class can potentially fill the slot corresponding to the class. In making this decision, we would like to match threads to class instances so as to *minimize the number of migrations*. And to achieve that, we refer to the matching solution for the old rebalancing interval, saved in the form of a tuple array: <old_domain, old_core, old_class, old_processID, old_threadID> for each thread.

Migrations are deemed superfluous if they change thread-core assignment, while not changing the placement of class-instances on cores. For example, if a thread that happens to be a devil (d) runs on a core that has been assigned the (d)-slot in the new assignment, it is not necessary to migrate this thread to another core with a (d)-slot. DI-Plain did not take this into consideration and thus performed a lot of superfluous migrations. To avoid them in DINO we first decide the thread assignment for any tuple that preserves core-class placement according to the new layout. So, if for a given thread `old_core = new_core` and `old_class = new_class`, then the corresponding tuple in the new solution for that thread will be <new_core, new_class, old_processID, old_threadID>.

For example, if the old solution were:

```
domain:          dom0  dom1  dom2  dom3
```

```
old_core:          0 1   2 3   4 5   6 7
old_class:         D t   d t   d t   d t
old_processID:     0 1   2 0   0 0   3 4
old_threadID:      0 1   2 3   4 5   6 7
```

then the initial shape of the new solution would be:

```
domain:            dom0  dom1  dom2  dom3
new_core:          0 1   2 3   4 5   6 7
new_class:         D t   D t   d t   d d
new_processID:     0 1     0   0 0   3
new_threadID:      0 1     3   4 5   6
```

Then, the threads whose placement was not determined in the previous step – i.e., those whose old class is not the same as their current core's new class, as determined by the new placement, will fill the unused cores according to their new class:

```
domain:            dom0  dom1  dom2  dom3
new_core:          0 1   2 3   4 5   6 7
new_class:         D t   D t   d t   d d
new_processID:     0 1   4 0   0 0   3 2
new_threadID:      0 1   7 3   4 5   6 2
```

Now that the thread placement is determined, DINO makes the final pass over the thread tuples to take care of multithreaded applications. For each thread A it checks if there is another thread B of the same multithreaded application (`new_processID(A) = new_processID(B)`) among the thread tuples not yet iterated so that B is not placed in the same memory domain with A. If there is one, we check the threads that are placed in the same memory domain with A. If there is a thread C in the same domain with A, such that `new_processID(A) != new_processID(C)` and `new_class(B) = new_class(C)` then we switch tuples B and C in the new solution. In our example this would result in the following assignment:

```
domain:            dom0  dom1  dom2  dom3
new_core:          0 1   2 3   4 5   6 7
new_class:         D t   D t   d t   d d
new_processID:     0 0   4 1   0 0   3 2
new_threadID:      0 3   7 1   4 5   6 2
```

DINO has complexity of $O(N)$ in the number of threads. Since the algorithm runs at most once a second, this has little overhead even for a large number of threads. We found that more frequent thread rebalancing did not yield better performance. Relatively infrequent changes of thread affinities mean that the algorithm is best suited for long-lived applications, such as the scientific applications we target in our study, data analytics (e.g., MapReduce), or servers. When there's more threads than cores coarse-grained rebalancing is performed by DINO, but fine-grained time sharing of cores between threads is performed by the kernel scheduler. If threads are I/O- or synchronization-intensive and have unequal sleep-awake periods, any resulting load imbalance must be corrected, e.g., as in [16].

3.3.4 DINO's Effect on Migration Frequency

We conclude this section by demonstrating how DINO is able to reduce migration frequency relative to DI-Migrate. Table 1 shows the average number of memory migrations per hour of execution under DI-Migrate and DINO for different applications from the workloads evaluated in Section 3.2. The results for MPI jobs are given for one of its processes and not for the whole job. Due to space limitations, we show the numbers for selected applications that are representative of the overall trend. The numbers show that DINO significantly reduces the number of migrations. As will be shown in Section 5, this results in up to 30% performance improvements for jobs in the MPI workload.

4 Memory migration

The straightforward solution to implement memory migration is to migrate the entire resident set of the thread when the thread is moved to another domain. This does not work for the following reasons. First of all, for multithreaded applications, even those where data sharing is rare, it is difficult to determine how the resident set is partitioned among the threads. Second, even if the application is single-threaded, if its resident set is large it will not fit into a single memory domain, so it is not possible to migrate it in its entirety. Finally, we experimentally found that even in cases where it is possible to migrate the entire resident set of a process, this can hurt performance of applications with large memory footprints. So in this section we describe how we designed and implemented a memory migration strategy that determines which of the thread's pages are most profitable to migrate when the thread is moved to a new core.

4.1 Designing the migration strategy

In order to rapidly evaluate various memory migration strategies, we designed a simulator based on a widely used binary instrumentation tool for x86 binaries called Pin [15]. Using Pin, we collected memory access traces of all SPEC CPU2006 benchmarks and then used a cache simulator on top of Pin to determine which of those accesses would be LLC misses, and so require an access to memory.

To evaluate memory migration strategies we used a metric called *Saved Remote Accesses* (SRA). SRA is the percent of the remote memory accesses that were eliminated using a particular memory migration strategy (after the thread was migrated) relative to not migrating the memory at all. For example, if we detect every remote access and migrate the corresponding page to the

Table 1: Average number of memory migrations per hour of execution under DI-Migrate and DINO for applications evaluated in Section 3.2.

	SPEC CPU2006					SPEC MPI2007				
	soplex	*milc*	*lbm*	*gamess*	*namd*	*leslie*	*lamps*	*GAPgeofem*	*socorro*	*lu*
DI-Migrate	36	22	11	47	41	381	135	237	340	256
DINO	8	6	5	7	6	2	1	3	2	1

thread's new memory node, we are eliminating all remote accesses, so the SRA would be 100%.

Each strategy that we evaluated detects when a thread is about to perform an access to a remote domain, and migrates one or more memory pages from the thread's virtual address space associated with the requested address. We tried the following strategies: ***sequential-forward*** where K pages including and following the one corresponding to the requested address are migrated; ***sequential-forward-backward*** where $K/2$ pages sequentially preceding and $K/2$ pages sequentially following the requested address are migrated; ***random*** where randomly chosen K pages are migrated; ***pattern-based*** where we detect a thread's memory-access pattern by monitoring its previous accesses, similarly to how hardware pre-fetchers do this, and migrate K pages that match the pattern. We found that sequential-forward-backward was the most effective migration policy in terms of SRA.

Another challenge in designing a memory migration strategy is minimizing the overhead of detecting which of the remote memory addresses are actually being accessed. Ideally, we want to be able to detect every remote access and migrate the associated pages. However, on modern hardware this would require unmapping address translations on a remote domain and handling a page fault every time a remote access occurs. This results in frequent interrupts and is therefore expensive.

After analyzing our options we decided to use hardware counter sampling available on modern x86 systems: PEBS (Precise Event-Based Sampling) on Intel processors and IBS (Instruction-Based Sampling) on AMD processors. These mechanisms tag a sample of instruction with various pieces of information; load and store instructions are annotated with the memory address.

While hardware-based event sampling has low overhead, it also provides relatively low sampling accuracy – on our system it samples less than one percent of instructions. So we also analysed how SRA is affected depending on the sampling accuracy as well as the number of pages that are being migrated. The lower the accuracy, the higher the value of K (pages to be migrated) needs to be to achieve a high SRA. For the hardware sampling accuracy that was acceptable in terms of CPU overhead (less than 1% per core), we found that migrating 4096 pages enables us to achieve the SRA as high as 74.9%. We also confirmed experimentally that this was a good value for K (results shown later).

4.2 Implementation of the memory migration algorithm

Our memory migration algorithm is implemented for AMD systems, and so we use IBS, which we access via Linux performance-monitoring tool `perfmon` [5].

Migration in DINO is performed in a user-level daemon running separately from the scheduling daemon. The daemon wakes up every ten milliseconds, sets up IBS to perform sampling, reads the next sample and migrates the page containing the memory address in the sample (if the sampled instruction was a load or a store) along with K pages in the application address space that sequentially precede and follow the accessed page. Page migration is effected using the *numa_move_pages* system call.

5 Evaluation

5.1 Workloads

In this section we evaluate DINO implemented using the migration strategy described in the previous section. We evaluate three workload types: SPEC CPU2006 applications, SPEC MPI2007 applications, and LAMP – Linux/Apache/MySQL/PHP.

We used two experimental systems for evaluation. One was described in Section 2.1. Another one is a Dell PowerEdge server equipped with two AMD Barcelona processors running at 2GHz, and 8GB of RAM, 4GB per domain. The operating system is Linux 2.6.29.6. The experimental design for SPEC CPU and MPI workloads was described in Section 3.2. The LAMP workload is described below.

The LAMP acronym is used to describe the application environment consisting of Linux, Apache, MySQL and PHP. The main data processing in LAMP is done by the Apache HTTP server and the MySQL database engine. The server management daemons *apache2* and *mysqld* are responsible for arranging access to the web-

site scripts and database files and performing the actual work of data storage and retrieval. We use Apache 2.2.14 with PHP version 5.2.12 and MySQL 5.0.84. Both *apache2* and *mysqld* are multithreaded applications that spawn one new distinct thread for each new client connection. This client thread within a daemon is then responsible for executing the client's request.

In our experiment, clients continuously retrieve from the Apache server various statistics about website activity. Our database is populated with the data gathered by the web statistics system for five real commercial websites. This data includes the information about website's audience activity (what pages on what website were accessed, in what order, etc.) as well as the information about visitors themselves (client OS, user agent information, browser settings, session id retrieved from the cookies, etc.). The total number of records in the database is more than 3 million. We have four Apache daemons, each responsible for handling a different type of request. There are also four MySQL daemons that perform maintenance of the website database.

We further demonstrate the effect that the choice of K (the number of pages that are moved on every migration) has on performance of DINO. Then we compare DINO to DI-Plain, DI-Migrate and Default.

5.2 Effect of K

Two of our workloads, SPEC CPU and LAMP demonstrate the key insights, and so we focus on those workloads. We show how performance changes as we vary the value of K. We compare to the scenario where DINO migrates the thread's entire resident set upon migrating the thread itself. The per-process resident sets of the two chosen workloads could actually fit in a single memory node on our system (it had 4GB per node), so whole-resident-set migration was possible. For SPEC CPU applications, resident sets vary from under a megabyte to 1.6GB for *mcf*. In general, they are in hundreds of megabytes for memory-intensive applications and much smaller for others. In LAMP, MySQL's resident set was about 400MB and Apache's was 120MB.

We show average completion time improvement (for Apache/MySQL this is average completion time per request), worst-case execution time improvement, and deviation improvement. Completion time improvement is the average over ten runs. To compute the worst-case execution time we run each workload ten times and record the longest completion time. Improvement in deviation is the percent reduction in standard deviation of the average completion time.

Figure 9 shows the results for the SPEC CPU workloads. Performance is hurt when we migrate a small number of pages, but becomes comparable to whole-resident-set migration when K reaches 4096. Whole-resident set migration actually works quite well for this workload, because migrations are performed infrequently and the resident set is small.

However upon experimenting with the LAMP workload we found that whole-resident set migration was detrimental to performance, most likely because the resident sets were much larger and also because this is a multithreaded workload where threads share data. Figure 10 shows performance and deviation improvement when $K = 4096$ relative to whole-resident-set migration. Performance is substantially improved when $K = 4096$. We experimented with smaller values of K, but found no substantial differences on performance.

We conclude that migrating very large chunks of memory is acceptable for processes with small resident sets, but not advisable for multithreaded applications and/or applications with large resident sets. DINO migrates threads infrequently, so a relatively large value of K results in good performance.

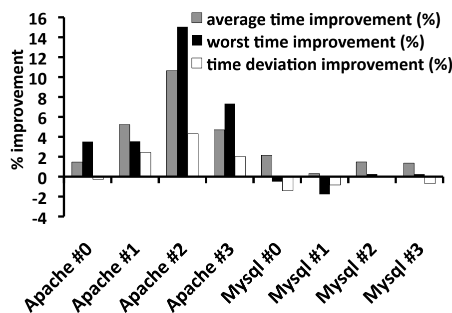
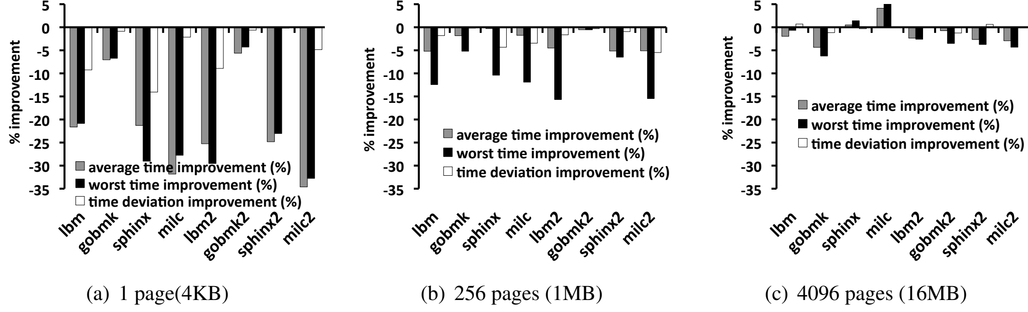


Figure 10: Performance improvement with DINO for $K = 4096$ relative to whole-resident-set migration for LAMP.

5.3 DINO vs. other algorithms

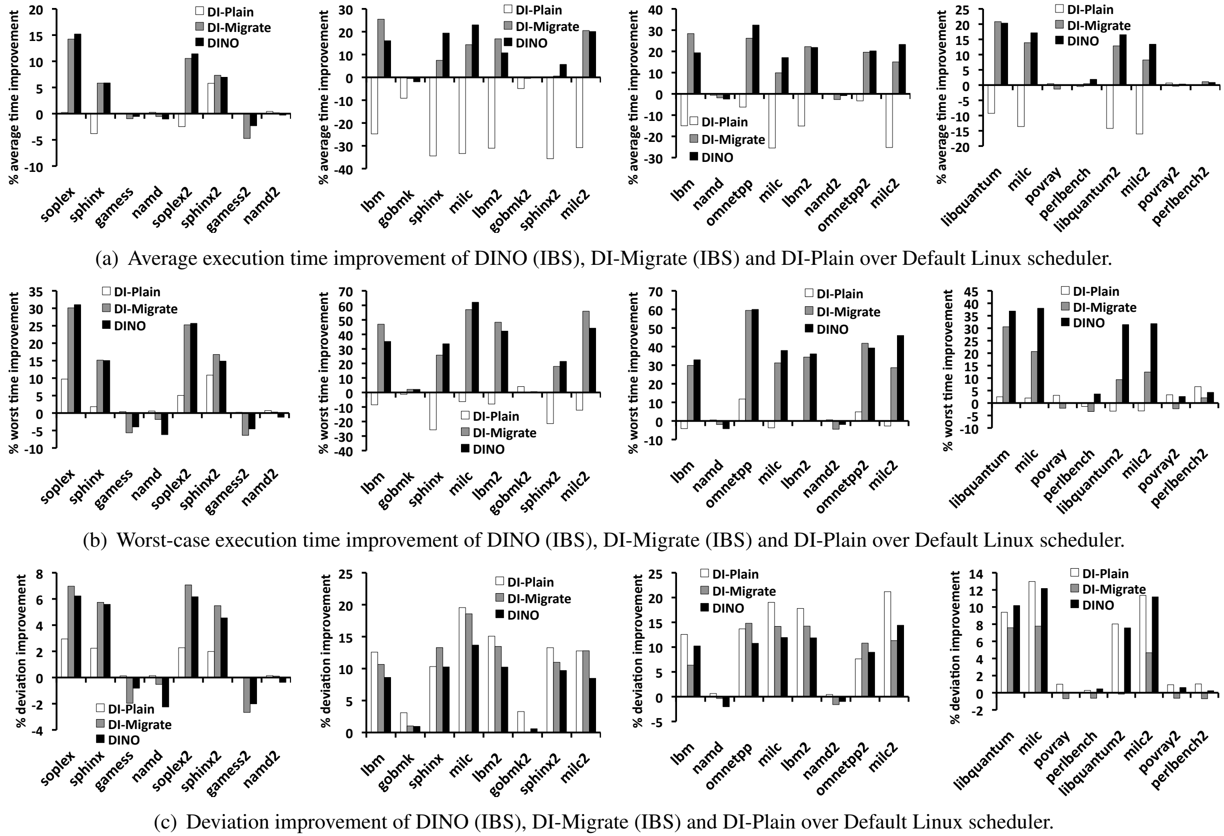
We compare performance under DINO, DI-Plain and DI-Migrate relative to Default, and similarly to the previous section, report completion time improvement, worst-case execution time improvement and deviation improvement.

Figures 11-13 show the results for the three workload types, SPEC CPU, SPEC MPI and LAMP respectively. For SPEC CPU, DI-Plain hurts completion time for many applications, but both DI-Migrate and DINO improve, with DINO performing slightly better than DI-Migrate for most applications. Worst-case improvement numbers show a similar trend, although DI-Plain does not perform as poorly here. Improvements in the worst-case execution time indicate that a scheduler is able to avoid pathological thread assignments that create especially high contention, and produce more stable performance. Deviation of running times is improved by all



(a) 1 page(4KB) (b) 256 pages (1MB) (c) 4096 pages (16MB)

Figure 9: Performance improvement with DINO as K is varied relative to whole-resident-set migration for SPEC CPU.



(a) Average execution time improvement of DINO (IBS), DI-Migrate (IBS) and DI-Plain over Default Linux scheduler.

(b) Worst-case execution time improvement of DINO (IBS), DI-Migrate (IBS) and DI-Plain over Default Linux scheduler.

(c) Deviation improvement of DINO (IBS), DI-Migrate (IBS) and DI-Plain over Default Linux scheduler.

Figure 11: DINO, DI-Migrate and DI-Plain relative to Default for SPEC CPU 2006 workloads.

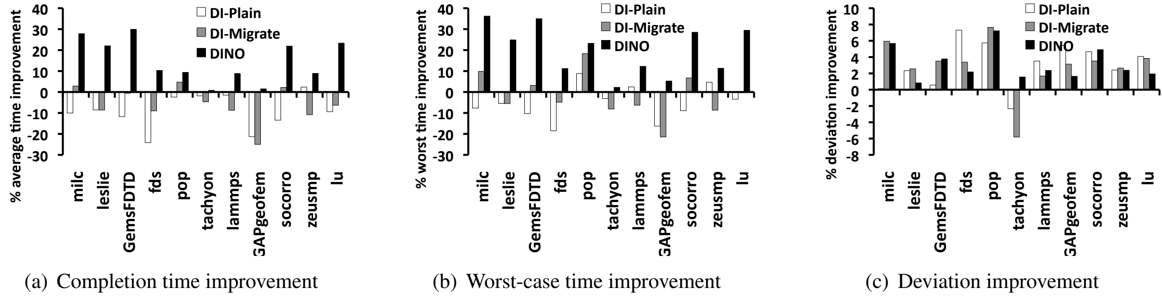
three schedulers relative to Default.

As to SPEC MPI workloads (Figure 12) only DINO is able to improve completion times across the board, by as much as 30% for some jobs. DI-Plain and DI-Migrate, on the other hand, can hurt performance by as much as 20%. Worst-case execution time also consistently improves under DINO, while sometimes degrading under DI-Plain and DI-Migrate.

LAMP is a tough workload for DINO or any scheduler that optimizes memory placement, because the workload is multithreaded and no matter how you place threads they still share data, putting pressure on interconnects. Nevertheless, DINO still manages to improve completion time and worst-case execution time in some cases, to a larger extent than the other two algorithms.

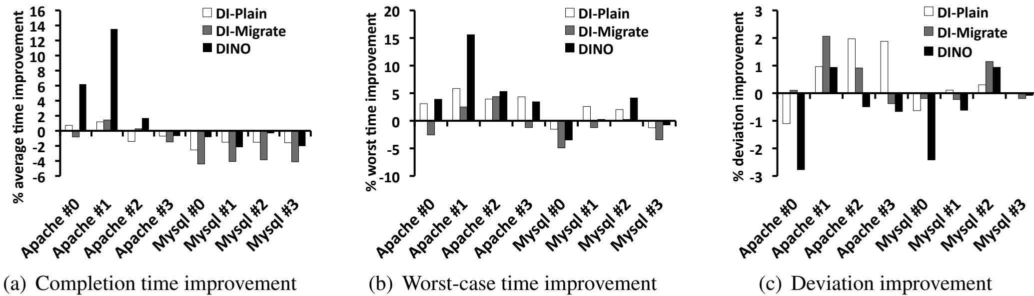
5.4 Discussion

Our evaluation demonstrates that DINO is significantly better at managing contention on NUMA systems than the DI algorithm designed without NUMA awareness or DI that was simply extended with memory migration.



(a) Completion time improvement (b) Worst-case time improvement (c) Deviation improvement

Figure 12: DINO, DI-Migrate and DI-Plain relative to Default for SPEC MPI 2007.



(a) Completion time improvement (b) Worst-case time improvement (c) Deviation improvement

Figure 13: DINO, DI-Migrate and DI-Plain relative to Default for LAMP.

Multiprocess workloads representative of scientific Grid clusters show excellent performance under DINO. Improvements for the challenging multithreaded workloads are less significant as expected, and wherever degradation occurs for some threads it is outweighed by performance improvements for other threads.

6 Related Work

Research on NUMA-related optimizations to systems is rich and dates back many years. Many research efforts addressed efficient co-location of the computation and related memory on the same node [14, 3, 12, 19, 1, 4]. More ambitious proposals aimed to holistically redesign the operating system to dovetail with NUMA architectures [7, 17, 6, 20, 11]. None of the previous efforts, however, addressed shared resource contention in the context of NUMA systems and the associated challenges.

Li et al. in [14] introduced AMPS, an operating system scheduler for asymmetric multicore systems that supports NUMA architectures. AMPS implemented a NUMA-aware migration policy that can allow or deny thread migration requested by the scheduler. The authors used the *resident set size* of a thread in deciding whether or not the OS schedule is allowed to migrate thread to a different domain. If the migration overhead were expected to be high the migration would be disallowed. Our scheduler, instead of prohibiting migrations, detects which pages are being actively accessed and moves them as well as surrounding pages to the new domain.

LaRowe et al. [12] presented a dynamic multiple-copy policy placement and migration policy for NUMA systems. The policy periodically reevaluates its memory placement decisions and allows multiple physical copies of a single virtual page. It supports both migration and replication with the choice between the two operations based on reference history. A directory-based invalidation scheme is used to ensure the coherence of replicated pages. The policy applies a freeze/defrost strategy: to determine when to defrost a frozen page and trigger reevaluation of its placement is based on both time and reference history of the page. The authors evaluate various fine-grained page migration and/or replication strategies, however, since their test machine only has one processor per NUMA node, they do not address contention. The strategies developed in this work could have been very useful for our contention aware scheduler if the inexpensive mechanisms that the authors used for detecting page accesses were available to us. Detailed page reference history is difficult to obtain without hardware support; obtaining it in software may cause overhead for some workloads.

Goglin et al. [8] developed an effective implementation of the *move_pages* system call in Linux, which allows the dynamic migration of large memory areas to be significantly faster than in previous versions of the OS. This work is integrated in Linux kernel 2.6.29 [8], which we use for our experiments. The *Next-touch* pol-

icy, also introduced in the paper to facilitate thread-data affinity, works as follows: the application marks pages that it will likely access in the future as *Migrate-on-next-touch* using a new parameter to the `madvise()` system call. The Linux kernel then ensures that the next access to these pages causes a special page fault resulting in the pages being migrated to their threads. The work provides developers with an opportunity to improve memory proximity for their programs. Our work, on the other hand, improves memory proximity by using hardware counters data available on every modern machine. No involvement from the developer is needed.

Linux kernel since 2.6.12 supports the *cpusets* mechanism and its ability to migrate the memory of the applications confined to the cpuset along with their threads to the new nodes if the parameters of a cpuset change. Schermerhorn et al. further extended the cpuset functionality by adding an automatic page migration mechanism to it [18]: if enabled, it migrates the memory of a thread within the cpuset nodes whenever the thread migrates to a core adjacent to a different node. Two options for the memory migration are possible. The first is a lazy migration, when the kernel attempts to unmap any anonymous pages in the process's page table. When the process subsequently touches any of these unmapped pages, the swap fault handler will use the "migrate-on-fault" mechanism to migrate the misplaced pages to the correct node. Lazy migration may be disabled, in which case, automigration will use direct, synchronous migration to pull all anonymous pages mapped by the process to new node. The efficiency of lazy automigration is comparable to our memory migration solution based on IBS (we performed experiments to verify). However, automigration requires kernel modification (it is implemented as a collection of kernel patches), while our solution is implemented on user level. Cpuset mechanism needs explicit configuration from the system administrator and it does not perform contention management.

In [19] the authors group threads of the same application that are likely to share data onto neighbouring cores to minimize the costs of data sharing between them. They rely on several features of Performance Monitoring Unit unique to IBM Open-Power 720 PCs: the ability to monitor CPU stall breakdown charged to different microprocessor components and using the data sampling to track the sharing pattern between threads. The DINO algorithm introduced in our work complements [19] as it is designed to mitigate contention between applications. DINO provides sharing support by attempting to group threads of the same application and their memory on the same NUMA node, but as long as co-scheduling multiple threads of the same application does not contradict with a contention-aware schedule. In order to develop a more precise metric that assesses the effects of performance degradation versus the benefits from co-scheduling, we would need stronger hardware support, such as that available on IBM Open-Power 720 PCs or on the newest Nehalem systems (as demonstrated by the member of our team [9]).

The VMware ESX hypervisor supports NUMA load balancing and automatic page migration for its virtual machines (VMs) in commercial systems [1]. ESX Server 2 assigns each virtual machine a home node on whose processors a VM is allowed to run and its newly-allocated memory comes from the home node as well. Periodically, a special rebalancer module selects a VM and changes its home node to the least-loaded node. In our work we do not consider load balancing. Instead, we make thread migration decisions based on shared resource contention. To eliminate possible remote access penalties associated with accessing the memory on the old node, ESX Server 2 performs page migration from the virtual machine's original node to its new home node. ESX selects migration candidates based on finding hot remotely-accessed memory from page faults. The DINO scheduler, on the other hand, identifies hot pages using Instruction-Based Sampling. No modification to the OS is required.

The SGI Origin 2000 system [4] implemented the following hardware-supported [13] mechanism for co-location of computation and memory. When the difference between remote and local accesses for a given memory page is greater than a tunable threshold, an interrupt is generated to inform the operating system that the physical memory page is suffering an excessive number of remote references and hence has to be migrated. Our solution to page migration is different in that it detects "hot" remotely accessed pages via Instruction-Based Sampling, and performs migration in the context of a contention-aware scheduler.

In a series of papers [7] [17] [6] [20] the authors describe a novel operating system Tornado specifically designed for NUMA machines. The goal of this new OS is to provide data locality and application independence for OS objects thus minimizing penalties due to remote memory access in a NUMA system. The K42 [11] project, which is based on Tornado, is an open-source research operating system kernel that incorporates such innovative design principles like structuring the system using modular, object-oriented code (originally demonstrated in Tornado), designing the system to scale to very large shared-memory multiprocessors, avoiding centralized code paths and global locks and data structures and many more. K42 keeps physical memory close to where it is accessed. It uses large pages to reduce hardware and software costs of virtual memory. K42 project has resulted in many important contributions to Linux, on which our work relies. As a result, we were able to avoid

deleterious effects of remote memory accesses without requiring changes to the applications or the operating system. We believe that our NUMA contention-aware scheduling approach that was demonstrated to work effectively in Linux can also be easily implemented in K42 with its inherent user-level implementation of kernel functionality and native performance monitoring infrastructure.

7 Conclusions

We discovered that contention-aware algorithms designed for UMA systems may hurt performance on systems that are NUMA. We found that contention for memory controllers and interconnects occurring when thread runs remotely from its memory are the key causes. To address this problem we presented DINO: a new contention management algorithm for NUMA systems. While designing DINO we found that simply migrating a thread's memory when the thread is moved to a new node is not a sufficient solution; it is also important to eliminate superfluous migrations: those that add to migration cost without providing the benefit. The goals for our future work are (1) devising metric for predicting a trade-off between performance degradation and benefits from thread sharing and (2) investigate the impact of using small versus large memory pages during migration.

References

[1] VMware ESX Server 2 NUMA Support. White paper. *[Online] Available: http://www.vmware.com/pdf/esx2_NUMA.pdf*.

[2] BLAGODUROV, S., ZHURAVLEV, S., AND FEDOROVA, A. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst. 28* (December 2010), 8:1–8:45.

[3] BRECHT, T. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *USENIX SEDMS* (1993).

[4] CORBALAN, J., MARTORELL, X., AND LABARTA, J. Evaluation of the Memory Page Migration Influence in the System Performance: the Case of the SGI O2000. In *Proceedings of Supercomputing* (2003), pp. 121–129.

[5] ERANIAN, S. What can performance counters do for memory subsystem analysis? In *Proceedings of MSPC* (2008).

[6] GAMSA, B., KRIEGER, O., AND STUMM, M. Optimizing IPC Performance for Shared-Memory Multiprocessors. In *Proceedings of ICPP* (1994).

[7] GAMSA, B., KRIEGER, O., AND STUMM, M. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of OSDI* (1999).

[8] GOGLIN, B., AND FURMENTO, N. Enabling High-Performance Memory Migration for Multithreaded Applications on Linux. In *Proceedings of IPDPS* (2009).

[9] KAMALI, A. Sharing Aware Scheduling on Multicore Systems. Master's thesis, Simon Fraser University, 2010.

[10] KNAUERHASE, R., BRETT, P., HOHLT, B., LI, T., AND HAHN, S. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro 28*, 3 (2008), pp. 54–66.

[11] KRIEGER, O., AUSLANDER, M., ROSENBURG, B., WISNIEWSKI, R. W., XENIDIS, J., DA SILVA, D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M., MERGEN, M., WATERLAND, A., AND UHLIG, V. K42: Building a Complete Operating System. In *Proceedings of EuroSys* (2006).

[12] LAROWE, R. P., JR., ELLIS, C. S., AND HOLLIDAY, M. A. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE Transactions on Parallel and Distributed Systems 3* (1991), 686–701.

[13] LAUDON, J., AND LENOSKI, D. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of ISCA* (1997).

[14] LI, T., BAUMBERGER, D., KOUFATY, D. A., AND HAHN, S. Efficient Operating System Scheduling for Performance-Asymmetric Multi-core Architectures. In *Proceedings of Supercomputing* (2007).

[15] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of PLDI* (2005).

[16] MERKEL, A., STOESS, J., AND BELLOSA, F. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of EuroSys* (2010).

[17] PARSONS, E., GAMSA, B., KRIEGER, O., AND STUMM, M. (De-)Clustering Objects for Multiprocessor System Software. In *Proceedings of IWOOS* (1995).

[18] SCHERMERHORN, L. T. Automatic Page Migration for Linux.

[19] TAM, D., AZIMI, R., AND STUMM, M. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of EuroSys* (2007).

[20] UNRAU, R. C., KRIEGER, O., GAMSA, B., AND STUMM, M. Hierarchical Clustering: a Structure for Scalable Multiprocessor Operating System Design. *J. Supercomput. 9*, 1-2 (1995), 105–134.

[21] XIE, Y., AND LOH, G. Dynamic Classification of Program Memory Behaviors in CMPs. In *Proceedings of CMP-MSI* (2008).

[22] ZHANG, E. Z., JIANG, Y., AND SHEN, X. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *Proceedings of PPOPP* (2010).

[23] ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. Addressing Contention on Multicore Processors via Scheduling. In *Proceedings of ASPLOS* (2010).