VIRENTRACK:
A HEURISTIC FOR REDUCING CACHE CONTENTION

by

Viren Kumar
B.Sc., Simon Fraser University, 2007

a Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the School
of
Computing Science

APPROVAL

Name:	Viren Kumar
Degree:	Master of Science
Title of Thesis:	Virentrack: A heuristic for reducing cache contention

Examining Committee: Dr. Arthur Kirkpatrick
Chair

Dr. Alexandra Fedorova,
Assistant Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Jim Delgrande,
Professor, Computing Science
Simon Fraser University
Supervisor

Dr. F. Warren Burton,
Professor Emeritus, Computing Science
Simon Fraser University
SFU Examiner

Date Approved: DEC 14, 2009

Abstract

Multicore processors are the dominant paradigm in mainstream computing for the present and foreseeable future. Current operating system schedulers on multicore systems co-schedule applications on cores at random. This often exacerbates issues such as cache contention, leading to a performance decrease. Optimally scheduling applications to take advantage of multicore characteristics remains a difficult and open problem.

In this thesis, I advocate a method of optimized scheduling on multicore systems that takes advantage of the caching attributes of applications. My scheduler is a user-level process that co-schedules applications based on cache metrics obtained from hardware performance counters. This phase-aware scheduler is able to effectively co-schedule two pairs of applications, extracting up to a 100% of all possible improvement in some workloads. Additionally, individual application performance gains of up to 13% are observed in some applications in co-schedules of two pairs.

To Chuck, Tom, Jeff, Dave and Kerry

"Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn"

— H. P. Lovecraft, 1928

Acknowledgments

I would like to thank Sasha Fedorova, without whose help this thesis would never have materialized.

I would also like to thank Mareija and all my colleagues in the systems lab at SFU.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multicore processors have established themselves as the *de facto* standard in computing today. The problem of optimally scheduling applications on multicore platforms to maximize performance remains wide open even today, with many possible solutions being proposed over the last decade. Before we describe scheduling and why optimal scheduling is so hard, let us examine the foundation of this problem in more detail.

1.1 Processor Composition

Modern processors are comprised of several components, including floating point units (FPUs), branch predictors, reorder buffers, instruction pipelines and caches. Main memory, which can be very large, is very slow, often taking hundreds of clock cycles to access, due to being located off-chip. Exploiting the principle of locality are high-speed memory buffers called caches, that are much smaller than main memory but many orders of magnitude faster to access. Caches are organized hierarchically as well, with the L1 cache being very small but very fast, and the L2 cache being bigger but a bit slower in terms of access time. Some processors have an L3 cache which is larger than the L2, but again is slower to access. The last level cache (LLC) of a processor is the lowest cache in the hierarchy before main memory.

Uniprocessors consist of a single core that runs on these components and uses them exclusively. Multicore processors, by definition have more than one core residing on a physical processor. Cores have private resources such as L1 caches, but share some on-chip resources such as L2 caches, L3 caches and FPUs. Since multiple cores compete for the same resources, this leads to contention for these on-chip resources. This thesis focuses on cache contention, specifically last-level caches, the caches at the lowest level before accessing main memory.

1.2 Scheduling

When a user starts an application, the operating system executes that application by creating a process for it. A process is defined as a unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources[11]. A process can have many lightweight processes or threads, which can execute in parallel. In this thesis, I focus on single-threaded applications, which have a single thread of execution on a given core at any given time. When a process executes, the operating system allocates a core to that process for its execution. Since multicore processors have more than one core on a chip, more than one application can be co-scheduled at a time on a multicore processor. These applications end up competing for and sharing the on-chip resources described above.

Applications that share these resources do so in varying amounts and frequencies, leading to differing access patterns and usage. When an application requests data that is present in a cache, it is termed a *cache hit*. If the data is not found in the cache, it is termed a *cache miss* and the data is fetched from main memory. To make room for the data fetched from main memory, some data present in the cache is evicted. If an application evicts data belonging to another co-scheduled application, it negatively affects the other application, because the latter application now has to re-fetch its data back into the cache. We shall see how, when two applications run simultaneously on a multicore processor and share a cache, their access patterns and usage determine how they impact each other.

1.3 Optimal Scheduling

What makes scheduling applications on a multicore platform so difficult is the impact co-scheduled applications have on each other, with regards to performance. Applications can impact each other in a negative fashion, leading to a degradation in performance. Performance is usually measured with the Instructions Per Cycle (IPC) metric, which measures how many instructions a processor can execute per clock cycle. When two or more applications that could negatively impact each other are co-scheduled on cores on the same

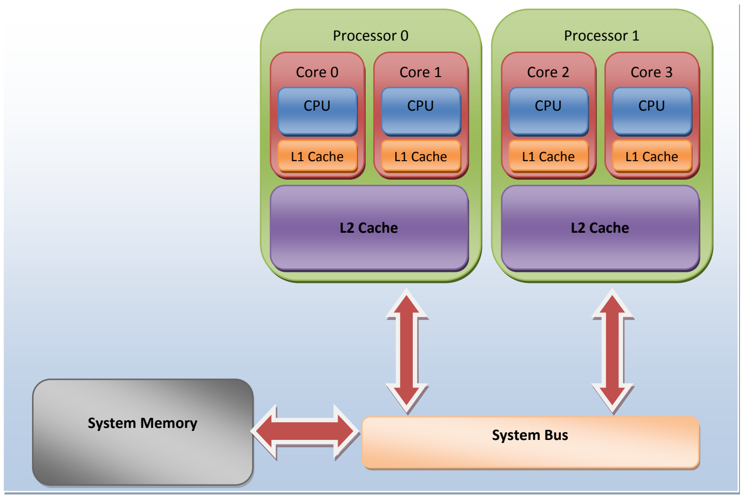


Figure 1.1: Intel Quad-core Shared Caches

processor, the performance degradation manifests itself as a reduction in the IPC of the applications. This is evident from the following example.

We consider four applications from the SPEC CPU 2006 benchmark suite. Our target system has two dual-core processors; each pair of cores share a last level cache, as seen in Figure 1.1. Two applications run concurrently on two cores and end up sharing that LLC. An application's solo IPC is defined as its IPC when run on a core with no cache-sharing, i.e., no other applications are co-scheduled to run on the other cores that share a cache with that application. By measuring the IPC of an application as it runs against another application, we can compute the decrease in IPC compared to the application's solo IPC, caused by cache contention. Doing this for all pairs gives us all possible IPC co-scheduled degradation data for the set of four benchmarks. In Table 1.1, we see the reduction in IPC when two applications are co-scheduled, displayed as a percentage decrease from the application's solo IPC. For example, the *milc* row shows the IPC degradation of the *milc* benchmark when co-scheduled with *omnetpp* as 11.79%.

From the set of four applications, three possible pairs of schedules are possible. If we consider the total degradation of a schedule to be the sum of degradation of its co-schedules, then an optimal schedule is the schedule with the minimum total IPC degradation. Table 1.2 shows all possible schedules and their total degradation. We calculate pair degradation of a pair of applications (A,B) as the sum of the IPC degradation when A is co-scheduled

Benchmark	mcf	milc	omnetpp	lbm
mcf	5.84	9.99	17.41	19.16
milc	4.4	10.13	11.79	18.96
omnetpp	8.73	13.53	17.69	21.95
lbm	9.13	15.05	10.16	24.11

Table 1.1: Co-schedule Degradation

with B, and when B is co-scheduled with A. For example, in Schedule #1, we see the Pair 1 degradation of *mcf* and *milc* as 9.99 + 4.4, which is the sum of the degradation of *mcf* when co-scheduled with *milc* (9.99), and the degradation of *milc* when co-scheduled with *mcf* (4.4), from Table 1.1 above.

#	Pair 1	Pair 2	Pair 1 degradation	Pair 2 degradation	Total IPC degradation
1	mcf + milc	omnetpp + lbm	9.99 + 4.4	21.95 + 10.16	55.50
2	mcf + lbm	milc + omnetpp	19.16 + 9.13	11.79 + 13.53	53.61
3	mcf + omnetpp	milc + lbm	17.41 + 8.73	18.96 + 15.05	60.15

Table 1.2: Total Schedule Degradation

We see that the schedule with the least IPC degradation is schedule #2, with a total of 53.61. This schedule is the optimal schedule, which delivers the least IPC degradation out of all the schedules. This defines the problem of optimal scheduling: finding an optimal schedule for a multicore processor implies finding a schedule where the performance degradation between the co-scheduled applications is minimized.

In our dual-core example above, all performance degradations for the applications were calculated beforehand. Unfortunately, even if all possible combinations of co-scheduled applications and their deleterious effects were known beforehand, optimal scheduling for more than 2 cores on a processor is an NP-complete problem.

Both the unrealistic assumption of total oracular application knowledge and the NP-completeness of the problem make it practically impossible to implement optimal scheduling in a real-world operating system. However, there are easier and more practical ways to minimize application performance degradation in a real system. Most modern processors have hardware performance counters, which can be used to track salient features of a running application. By accessing and tracking some prominent hardware parameters, such as an application's cache access rate and cache miss rate, we can effect more efficient and optimized scheduling policies.

Chapter 2

Related Work

Much work has been done in the field of multicore scheduling. Since my thesis focuses on cache-conscious scheduling on multicore processors, this section focuses on those papers that are directly related to this subset of multicore scheduling.

2.1 Cache-conscious Scheduling

2.1.1 Software Cache Partitioning

Xie et al.[13] provided the biggest inspiration for this work, with their animalistic classification of applications. In their work, the authors classify applications into four classes based on their cache access rates and miss rates. The classes are named after animals and provide a straightforward way to distinguish between applications based on their cache access patterns. A dynamic cache partitioning scheme then isolates applications that share a cache from each other, leading to improved performance for some pairings of application classes. However, their scheme is meant to be implemented in hardware, while the present approach is implemented in software and is hardware-independent. This approach doesn't isolate applications by partitioning the cache, or restrain certain types by limiting the number of ways an application can have in the cache[1]. Rather, it simply checks which applications are least likely to impact each other when scheduled together and co-schedules them.

In the animalistic paper described above, the authors relied on the Utility Cache Partitioning (UCP) scheme devised by Qureshi and Patt [8]. UCP is a dynamic cache partitioning

[1]See footnote in Section 3.2

algorithm that divides the cache among applications that share it. It too, is implemented in hardware and works on the premise that the cache should be given to those who benefit more from it, not those who demand more of it. Unlike their approach, the present scheme doesn't rely on cache partitioning to alleviate cache contention, but simply co-schedules applications that share caches together, based on simple parameters.

Tam et al. [12] attempt to implement cache conscious scheduling by implementing cache partitioning in software. Their approach partitions the cache between applications using page allocation, à la the OS page-colouring method. Page-colouring is a cache-friendly technique of allocating free pages in main memory to an application, such that the newly-allocated pages map to contiguous sections in the application's cache. By removing control of the page allocation process from the OS, the authors guarantee that newly allocated pages in memory map to the lines allocated to the application in the L2 cache. This enforces application isolation via partitioning in the L2 cache. Furthermore, they use an application's instruction retirement stall rate curve (SRC), gathered from hardware performance counters, as a predictor of performance as a function of L2 cache size. While this approach gives them an efficient online method for estimating performance, their technique requires changes to the virtual memory system, a very complex part of the OS. Additionally, reducing the size of an L2 cache partition results in expensive memory copying, should the need arise. The current approach is similar to theirs in that it also relies on observable events from the hardware performance counters, but differs in that it needs no changes to the virtual memory system of the OS. Finally, this approach eschews static partitioning, or indeed, partitioning of any kind and co-schedules applications to share a cache without any partitioning scheme.

In the same vein as Tam above, Zhang et al. also suggest software-based cache partitioning using hot page colouring [14]. By colouring only frequently-accessed or "hot" pages, the authors hope to avoid some of the drawbacks of page allocation by colouring. However, they mention that their approach suffers from overhead which can be mitigated by partial hardware support. The present scheme is different by virtue of sidestepping cache partitioning altogether.

Soares et al. describe another software partitioning scheme in their paper on reducing cache pollution with a pollute buffer [10]. Unlike other software partitioning schemes that try to reduce cache contention between two or more co-scheduled applications, the pollute buffer attempts to improve cache performance from within a single application. The pollute buffer is simply a software partition of the cache that is used by pages in the application with a high last level cache miss rate. By correlating the L2 cache miss rate of an application's pages with its ability to pollute the cache, the authors implement a dynamic, online monitoring system that uses hardware performance counters to reduce L2 cache misses and improve performance. The present approach is superficially similar in that it also uses hardware performance counters to improve performance, but as noted for all the papers in this section, cache partitioning is avoided altogether. Additionally, the presented method works for two or more applications in a chip multiprocessor with shared last-level caches, as opposed to just one application.

2.1.2 Other

Snavely and Tullsen make a case for symbiotic job scheduling in their paper on scheduling on a simultaneous multi-threading (SMT) processor [9]. Their approach works at a finer granularity than mine, focusing on SMT processors, while mine focuses on single-threaded multicore processors (CMP). The authors' approach is two-pronged, with an online sampling phase where all permutations of possible schedules are evaluated, followed by a symbios phase where the best candidate schedule from the sampling phase is executed. The current approach differs from theirs in that it does not require all permutations of schedules, but rather uses hardware counter events to decide on optimal scheduling strategies.

In their paper on the analysis and approximation of optimal co-scheduling [5], Jiang et al. provide a thorough theoretic grounding for the problem of optimal co-scheduling on multicore processors. The approach in this thesis mirrors theirs, in aspects such as measuring co-run degradation between co-scheduled apps and calculating optimal schedules. The authors first prove that optimal co-scheduling for k processors, where $k > 2$ is NP-complete. They provide an implementation of Edmonds' blossom algorithm [3] to find an optimal co-schedule for dual-core processors, and then suggest approximation algorithms for processors with more than two cores. However, their approach relies on *a priori* knowledge of all possible co-run degradations to suggest solutions for optimal scheduling. While the current approach does use *a priori* co-run degradation knowledge to build a model for co-scheduling, the live scheduler does not require prior knowledge of co-schedule degradation to produce a schedule, relying on hardware performance counter-generated events instead.

The work of Blagodurov et al. [1], also in the systems lab at Simon Fraser University, is closely related to my work in this thesis. The report has the same overall goal of reducing multicore contention, but the authors pursue several different approaches to reach the same goal. They attempt to optimize multicore scheduling by classifying applications according to several schemes, including ones based on stack distance profiles and animalistic classes. One major difference between their work and mine is that they only consider cache miss rates, while this work considers both cache miss rates and access rates as inputs for its live scheduler. Their report provides an independent confirmation of the methods used in this thesis, and correspondingly, the results of this thesis provide independent verification of some of theirs.

2.1.3 Models

Petoumenos et al. suggest using a cache sharing and management framework called StatShare [7], to model an application's behaviour in a shared cache. By building a reuse distance histogram with cache replacements instead of the usual intervening memory accesses, the authors are able to predict an application's hits and misses with high accuracy. The model's output can then be used to drive co-scheduling decisions. However, their approach would involve modifying hardware and this precludes any kind of software portability. In contrast, this thesis proposes a software-based method for making co-scheduling decisions based on cache access patterns.

Hsu et al. provide a framework for comparing optimal performance targets in CMP architectures with their paper on *communist*, *utilitarian* and *capitalist* cache sharing [4]. They contend that defining performance itself on multicore processors that share caches amongst threads is not as simple as it was on uniprocessors, with alternative definitions such as fairness or IPC throughput. A whole host of metrics present themselves to be used in analysis, such as IPC, miss rates, misses per access and so on. Furthermore, cache policies such as Least Recently Used (LRU) can have an effect on the achievement of the overall goal. The authors define a *policy metric* as an easily observable metric that can be used for online scheduling in lieu of a more complex *evaluation metric* that may be harder to measure in an online scheduler. To borrow their terminology, in the present work the policy metrics are hardware counter events and the evaluation metric is the IPC.

In their paper on predicting inter-thread cache contention, Chandra et al. propose three models that evaluate the impact of cache sharing among co-scheduled threads [2]. However, their models rely on stack distance profiles, which have to be calculated offline, and thus are not very practical for live scheduling decisions at runtime. The *Frequency of Access* model depends solely on the access frequency of an application and can be inaccurate if the ratios of miss and reuse frequency are different. The second model, the *Stack Distance Competition* model can also be inaccurate if the threads being measured differ wildly in their miss frequency. The third and most accurate model, the *Inductive Probability Model* is the most accurate, but is too computationally demanding to be used in realtime scheduling. In contrast, my approach relies on simpler events that can be easily measured online.

Knauerhase et al, follow a similar approach to mine, in their work on using observable events in the OS to improve performance in multicore systems [6]. While their method of co-scheduling applications based on cache events observed online is similar to mine, they do not provide a rigorous analysis of the justification behind their approach. The experimental section of their paper omits to mention some details, such as why the hardware events are measured at every context switch and not every million cycles.

2.1.4 Summary

Thus, we have seen that there are three major approaches one can consider, as a potential means of alleviating cache contention. Hardware cache partitioning is one of the most efficient, but unfortunately it requires special hardware, which is not easily available on most production systems today. Software partitioning of the caches can be effective as well, but doing so requires non-trivial changes to the virtual memory subsystem of an operating system, something that cannot be undertaken lightly. Finally, scheduling to mitigate cache contention makes up the third category and it is this category that this thesis falls into. Scheduling applications on the basis of their cache attributes can be done without either complex changes to the OS or exotic underlying hardware. In the rest of the thesis, we shall see how co-scheduling applications based on their cache access patterns can minimize cache contention and improve performance.

Chapter 3

Analysis

This chapter explains the rationale behind the metrics used in this thesis, including an overview of the foundation behind the choices. The chapter begins with a brief overview of the two types of hardware memory architectures used in the experiments in this thesis. The succeeding sections explain the animalistic classes used in this body of work and their source. Some anomalies with the animalistic model are described, along with attempts at explaining some of them. The terms: sensitivity and intensity, are described and used in the context of co-scheduling. This is followed by a justification for certain cache events as good predictors for both intensity and sensitivity. Statistical analysis then shows that these cache events can be used as proxies for sensitivity and intensity in online scheduling. Finally, this chapter concludes by presenting data showing the differences between scheduling sensitive and insensitive applications with intensive applications, as well as the differences between intensive and non-intensive applications.

3.1 Experimental Setup

This thesis uses two different systems in its experimental section. A short note about memory interleaving on Non-Uniform Memory Access (NUMA) systems is in order. Systems that implement a NUMA memory design offer faster access to regions of memory for select cores. A core thus has *local* and *remote* memory, where access to local memory is faster than access to remote memory. When accessing *local* memory, these cores benefit from smaller cache penalties, since they are able to access certain areas of memory faster than the other cores. To reduce this unfairness in memory access, node interleaving distributes or interleaves memory allocation across all the regions of memory equally, for a running application. However, this also reduces the potential gains from NUMA, since a processor has to access *remote* memory as well as *local* memory for its operations. Additionally, performance gains from NUMA are hard to predict and measure, being highly dependent on many factors such as cache sizes, bus latencies, memory sizes and the nature of the actual workload.

In contrast with NUMA, we have the older, more established Uniform Memory Access (UMA) which implies that no processor has *local* or *remote* memory, with each processor having the same access speeds for all regions of shared memory. With standardized access to memory regions from all processors, all processors suffer an equal penalty for a cache miss. This levels the playing field and makes performance measurement more predictable and repeatable, by eliminating the varying dependence on local and remote memory speeds.

The first system is a NUMA system with memory interleaving and is based on an AMD quad-core processor where the last-level cache is an L3 cache, not an L2 cache as in the previous system. The details of the first system used in the experiments in the rest of this thesis are presented below:

- Two Quad-Core AMD Opteron(tm) Processor 2350s for a total of 8 cores, all at 2 GHz
- Each core has a 64 KB L1 instruction cache, a 64 KB L1 data cache and a 512 KB L2 cache
- All 4 cores on a chip share a common 2 MB L3 cache
- 8 GB of RAM
- The operating system is Open Solaris 5.11 build 95

The NUMA system is a quad-core system with more RAM than the original system seen in Section 3.2.1, permitting more interesting experiments to be run. Also, being more modern, the new system is more representative of future systems.

The second UMA system has no memory interleaving, by definition and is an Intel Xeon-based machine, with a shared L2 cache as the last level cache. The details of this second UMA system are similar to the system used in Section 3.2.1, except that it has twice the number of cores and memory.

- Two Quad-Core Intel Xeon Processor 5365 processors for a total of 8 cores, all at 3.00 GHz
- Each core has a 32 KB L1 instruction cache and a 32 KB L1 data cache
- Pairs of cores share two 4 MB L2 caches
- 8 GB of RAM
- The operating system is Open Solaris 5.11 build 86

3.2 Animalistic Classification

This work was inspired by the research done by Xie et al. [13], who classified applications into animalistic categories, based on some select criteria. In their paper, Xie et al. categorized applications as either turtles, rabbits, sheep or devils. By monitoring the cache access rate and miss rate of applications, the authors were able to slot running applications into one of the four categories based on criteria explained below. The paper defines the following metrics for use in the classification of applications:

- $Accesses$ – the total number of accesses to the L2 cache, including instruction, data and prefetch requests
- $Misses_{solo}$ – the total number of L2 misses if the program had sole use of the entire n ways[1] of the cache
- $MissRate_{solo}$ – the relative L2 miss rate if the program had sole use of the entire n ways of the cache ($Misses_{solo}/Accesses$)
- $WaysNeeded_{k\%}$ – The smallest number of ways needed to achieve a miss rate that is less than or equal to $k\%$ of $MissRate_{solo}$.

The above metrics are sampled once every million cycles, with the following conditions being used to classify an application into one of the four animal classes:

[1]A cache is divided into lines, with entire lines being fetched at a time from main memory. A group of lines form a set. A cache with n lines in a set is called *n-way* set associative. The number of ways in a cache denotes the cache placement of a line and lies on a continuum, from a direct-mapped cache (one-way set associative) to a fully associative cache (n-way set associative, where n is the number of lines in the cache).

- Turtle – if ($Accesses < 1000$), i.e. an application is a turtle if its cache access rate is low. This type of application doesn't access the cache much at all.
- Rabbit – if ($WaysNeeded_{95\%} > \frac{n}{2}$), i.e. an application is a rabbit if it needs many cache ways to keep its miss rate low. An application that's a rabbit suffers when its cache is shared.
- Devil – if (($MissRate_{solo} > 10\%$) OR ($Misses_{solo} > 4,000$)), i.e. an application is a devil if its miss rate is high. A devil application hurts the other applications it is co-scheduled with.
- Sheep – if an application doesn't match any of the conditions above, i.e. an application is a sheep if it needs a few cache ways to keep its miss rate low. This type of application is content with very few cache ways.

Our primary goal to investigate whether these classifications could be leveraged to aid in co-scheduling applications. When performance is measured with the Instructions per Cycle (IPC) metric, scheduling a devil with any application should leave the devil's performance untouched, while degrading the IPC of the co-scheduled application. Conversely, scheduling a turtle with any application should leave both the co-scheduled application's and the turtle's IPC unharmed.

3.2.1 Original Experimental Setup

The original goal of Xie et al.[13] was to investigate a method that could be used to classify applications, in the event that such a classification proved helpful for co-scheduling applications. By adapting the animalistic scheme to a real operating system running on real hardware, we could determine whether classifying applications using performance counters was a viable approach to co-scheduling applications, on a real system. All benchmarks from the SPEC CPU 2006 suite (both CINT and CFP) were compiled for a 64-bit environment. Reference inputs were used and the benchmarks were run on a system with the following characteristics:

- One Quad-Core Intel Xeon Processor 5320 for a total of 4 cores, all at 1.86 GHz
- Each core has a 32 KB L1 instruction cache and a 32 KB L1 data cache

- Pairs of cores share two 4 MB L2 caches
- 4 GB of RAM

The above system was chosen because it matched the animalistic model's system very closely, in terms of hardware specifications. The animalistic paper's setup was mimicked as closely as possible to reduce disparity between their results and the present approach's.

Measuring the last-level (L2) cache misses and access rates allowed us to sort the data and enable a preliminary classification of devils and turtles. Recall that devils are applications with a high cache miss rate, while turtles are applications with a low cache access rate. This work focused on devils and turtles because they had easily observable characteristics. Determining if an application is a sheep or rabbit is not possible in a real system, because there is no easy way to modify the number of ways in a cache while the system is operational.

To determine whether the classification would aid co-scheduling, we decided to first classify some applications in the SPEC suite as devils and turtles. Having classified some applications into these two categories, the next step was to run devils with non-devil applications and measure the IPC degradation of both the devils and non-devils.

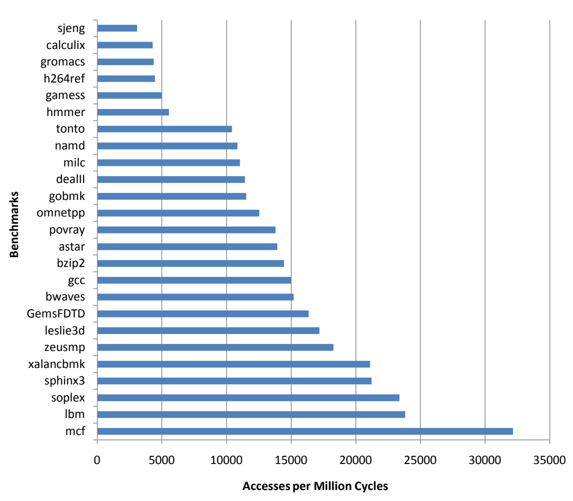
Figure 5.8 lists the benchmarks in the SPEC CPU 2006 benchmark suite, sorted by the number of accesses per million cycles, and misses per million cycles, in parts (a) and (b) respectively.

3.2.2 Identifying Turtles

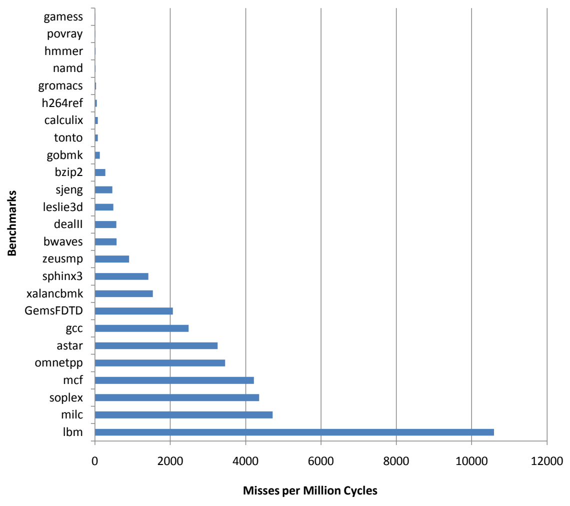
From Figure 3.1, we see that we can identify some turtles tentatively. The original paper states that a turtle is an application with < 1000 accesses per million cycles. Since that would mean I had no turtles, it's reasonable to consider some of the lowest applications as tentative turtles. By that logic *sjeng*, *calculix* and *gromacs* would be candidates for turtles.

3.2.3 Identifying Devils

Similarly, the original work lists devils as applications with > 4000 misses per million cycles. Using this as the cutoff, the applications with the highest misses were chosen as tentative devils. This leaves us with *omnetpp*, *mcf*, *soplex*, *milc* and *lbm* as potential devils.



(a) Accesses



(b) Misses

Figure 3.1: Accesses and Misses per Million Cycles

3.2.4 Inconsistencies Revealed

When devils and non-devils were co-scheduled, the results showed that the devils inflicted damage on most of the non-devil applications, but some non-devils benefited from being co-scheduled with devils. Figure 3.2 shows the reduction in IPC of a non-devil application when co-scheduled with a devil, as a percentage decrease over the non-devil application's solo IPC. The devils in this scenario are the five potential devils identified in Section 3.2.3, while the non-devils are the remaining benchmarks in the SPEC 2006 benchmark suite. A negative reduction in IPC is an increase in IPC, which is an unexpected result. This increase in performance when scheduled with a devil is not touched upon in the original paper. As an extreme example, consider the performance of *h264ref* when co-scheduled with the devils, in Figure 3.2. Its negative decrease in IPC, or increase in performance is almost 100%, a phenomenon that we are at a loss to explain.

Additionally, the devils themselves suffered performance degradation, something that was unexpected according to the animalistic classification scheme. The devils were run against a cross-section of other benchmarks from the suite, and the IPC degradation of each devil was recorded. Figure 3.3 shows us the average degradation in IPC of a devil in its co-schedules with the other benchmarks, as a percentage of the devil's solo IPC. All five devils show a loss in performance, something that is not explained in the original paper.

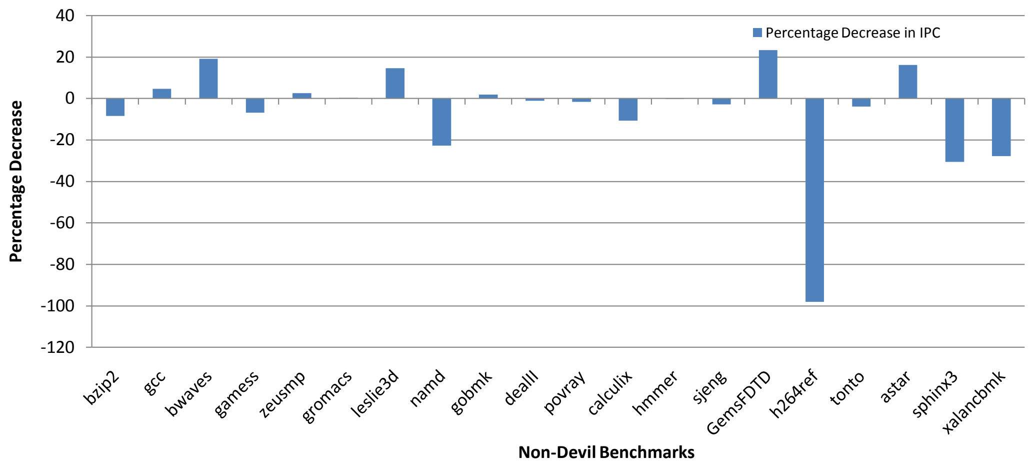


Figure 3.2: IPC Reduction of Non-devils

A technical report done by Blagodurov et al. [1] shows that devils suffer IPC degradation because of factors apart from just cache contention. Contention for other hardware components such as the memory bus and the memory controller serves to negatively impact the IPC of a devil when co-scheduled with other applications. Pre-fetching lines from memory into the cache is another approach heavily used by devils, and one prone to cause heavy IPC degradation when contended for by other co-scheduled applications. These factors explain some of the anomalous behaviour of devils, as seen by my experiments but not explained in the original animalistic paper.

3.3 New Method

The experiments in this thesis gave rise to a scheme for co-scheduling applications, based on the twin parameters of *sensitivity* and *intensity*. Sensitivity is a measure of how much an application is affected when co-scheduled with another application. If an application suffers a tremendous degradation in its IPC when co-scheduled, as compared to its solo IPC, then we say that an application is sensitive. On the other hand, if an application suffers little to no degradation when co-scheduled with another application, the application is deemed to be insensitive. Intensity is defined as a measure of how much an application affects the other application it is co-scheduled with. If an application causes significant degradation in the IPC of the other application it is co-scheduled with, then we say that the application

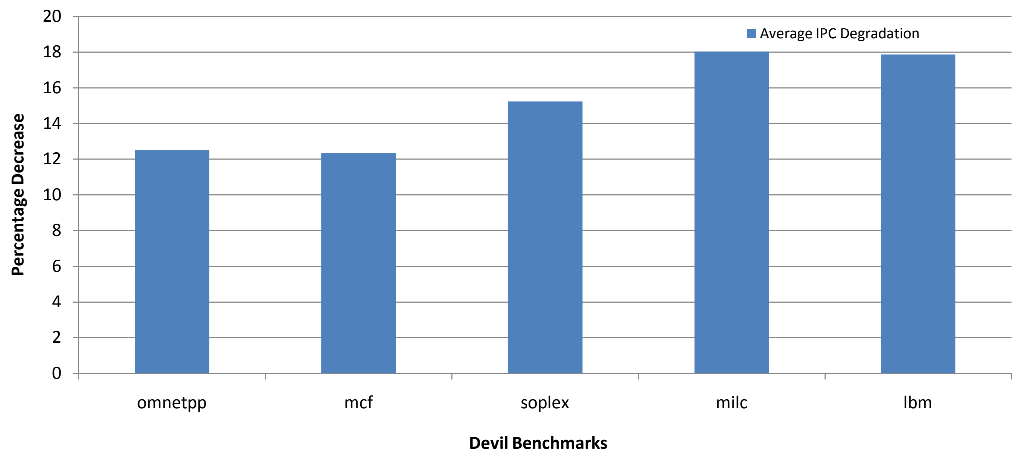


Figure 3.3: IPC Reduction of Devils

is highly intensive. Analogous to the sensitivity definition above, an application is lightly intensive if when co-scheduled with another application, it induces little to no degradation in the IPC of the other application.

A further point of importance is that neither sensitivity nor intensity need to be precisely defined. It is not necessary to decide how much an application's IPC should suffer, for it to be classed as sensitive. Similarly, one doesn't need a precise amount of inflicted IPC damage to classify an application as intensive. All that is needed is a way to compare the *relative* sensitivity and intensity of different applications.

If sensitivity and intensity are to be used for co-scheduling applications in real-time, a low-overhead method of calculating them is needed. Most modern processors have hardware performance counters which allow one to gather several statistics about a running application, including events such as the number of clock cycles consumed, the number of main memory accesses and so on. Two straightforward events that could be investigated further were:

- Cache access rate
- Cache miss rate

To better understand and define sensitivity in terms of easily measurable events, we measured how much an application's performance suffered when co-scheduled with another application. Performance is measured in terms of the IPC metric, and sensitivity is measured as a percentage drop in the IPC of the application in the co-schedule, compared to the IPC when the application runs by itself. The average sensitivity of an application is the average sensitivity of the application over many runs with the benchmarks in the SPEC suite.

Similarly, the intensity of an application is measured by calculating the IPC degradation inflicted on a co-scheduled application. The intensity is measured as a percentage drop in the IPC of the other co-scheduled application, compared to its solo IPC. The average intensity of an application is the average of its intensities over many runs with other SPEC benchmarks.

Once the average sensitivities and intensities had been gathered, the next step was to find a reliable way to predict the sensitivity and intensity of an application during its runtime. With that in mind, the twelve cache metrics described in the next section were tested as possible predictors for an application's sensitivity and intensity. An application's cache access rate and miss rate were not gathered live during each run, with the twelve solo cache metrics described below being used instead.

3.3.1 Choice of Metric

Though the work of Xie et al. [13] focused on the last-level cache, the last-level cache suffers from cache contention when two or more co-scheduled applications share the cache. This results in cache access and miss numbers that may differ substantially from solo runs. However, an application's cache access and miss rate for its private caches would be expected to stay relatively constant, whether the application is run solo or co-scheduled with other applications, due to the opacity of the private caches to other co-running applications. Measuring an application's private cache access and miss rates would thus be stabler across co-scheduling runs. However, a private cache metric might not be a good predictor of sensitivity or intensity. To determine which cache metric was the best predictor of sensitivity and intensity, the solo access and miss rates of applications in all three levels of caches in the experimental system above were measured. By running linear regressions on the solo access rates and miss rates, and the intensity and sensitivity, we looked for a strong correlation between these cache metrics and sensitivity and intensity. A strong correlation between a cache event and sensitivity would mean that the cache event could be used as a predictor for sensitivity in an online scheduler. The same holds true for intensity as well. Each metric could be measured in per-cycle or per-instruction terms, leading to twelve possible choices for the cache metrics.

- Per cycle
  - L1 cache
    1. Accesses
    2. Misses
  - L2 cache
    1. Accesses
    2. Misses
  - L3 cache
    1. Accesses
    2. Misses
- Per instruction
  - L1 cache
    1. Accesses
    2. Misses
  - L2 cache
    1. Accesses
    2. Misses
  - L3 cache
    1. Accesses
    2. Misses

3.3.2 Sensitivity

This section describes how sensitivity was correlated to the cache metrics described above.

All the benchmarks in the SPEC CPU 2006 suite were co-scheduled with all the other benchmarks in the suite to measure performance degradation, for a total of 625 co-schedules (four would not compile on my system). Figure 3.4 shows the benchmarks from SPEC CPU

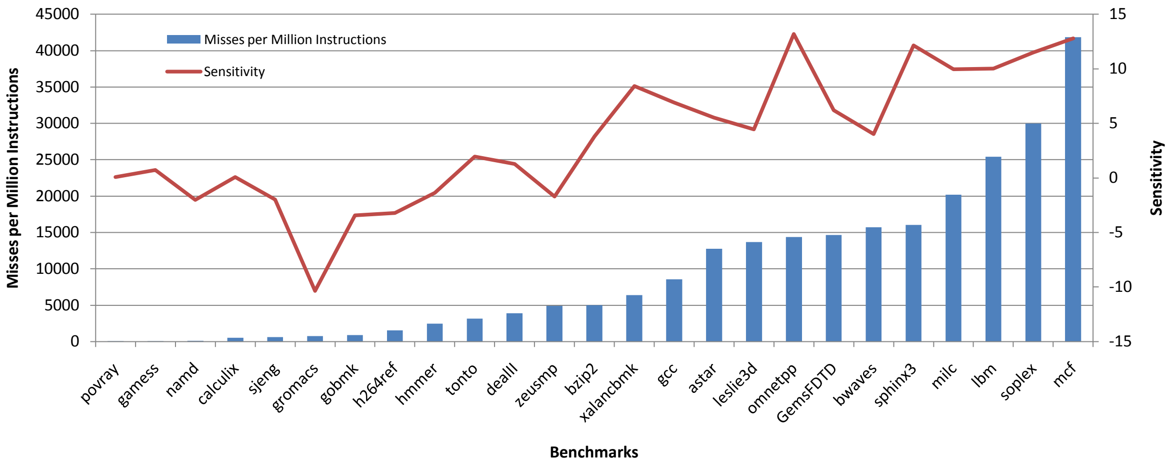


Figure 3.4: Miss Rates and Sensitivity

2006, sorted by their L2 cache miss rate and the average suffering they experienced when co-scheduled with another application.

This was done for the L1 and L3 cache events as well, to see which cache events would have the greatest correlation with suffering. We ran regression analyses on the data to obtain a correlation between suffering and cache access rate, if any. The R-square of the linear regression correlating the various cache events and IPC suffering is shown in Table 3.1.

Of these, the most promising were the L2 cache metrics, particularly the L2 misses per million instructions metric, with the highest R-square value of 63.06. Further statistical detail on this metric is provided below.

From Table 3.2 we see that there exists a correlation of 63% between the application's suffering and its cache miss rate, i.e. 63% of the variation in an application's sensitivity can be explained by the application's L2 cache miss rate. At a significance level of 0.001, i.e. α = 0.001, with 25 data points from the benchmarks, $F_{0.001,1,23} = 14.20$.

Since the observed f value, $39.2647 > F_{0.001,1,23}$, we can safely reject the null hypothesis (which states that there is no relationship between the IPC and the L2 cache miss rate) and assume that there is an approximate linear relationship between the degradation in IPC and the cache miss rate. Confirming this conclusion is the extremely low P-value of 0.000002, which strongly contradicts the null hypothesis.

This is intuitive because the more an application is affected by another application

Per Million Cycles	
Metric	**R square**
L1 accesses	40.30
L1 misses	37.23
L2 accesses	58.01
L2 misses	48.89
L3 accesses	49.34
L3 misses	35.49
Per Million Instructions	
Metric	**R square**
L1 accesses	1.91
L1 misses	38.30
L2 accesses	41.69
L2 misses	63.06
L3 accesses	53.31
L3 misses	57.58

Table 3.1: Cache Metrics and Sensitivity Correlation

Statistic	Value
R square	63.0609
f	39.2647
P-value	0.000002

Table 3.2: Sensitivity and Correlation between Miss Rate

sharing the same cache, the more misses it incurs. There is another big advantage to using the cache miss rate as a predictor for suffering: it can be easily obtained live from hardware performance counters, during the application's runtime.

Thus, it seems as though the L2 cache miss rate is a good metric for sensitivity.

3.3.3 Intensity

Following the same experimental setup as the one used for sensitivity above, we sought to correlate an application's cache miss rate and cache access rate with its intensity.

Figure 3.5 shows the L2 cache access rates and miss rates of an application, combined with the IPC decrease it causes in other applications. The degradation in IPC is measured as a percentage decrease in the IPC when compared against the IPC of a solo run.

Similarly, the L1 and L2 cache data was gathered to determine which cache event would

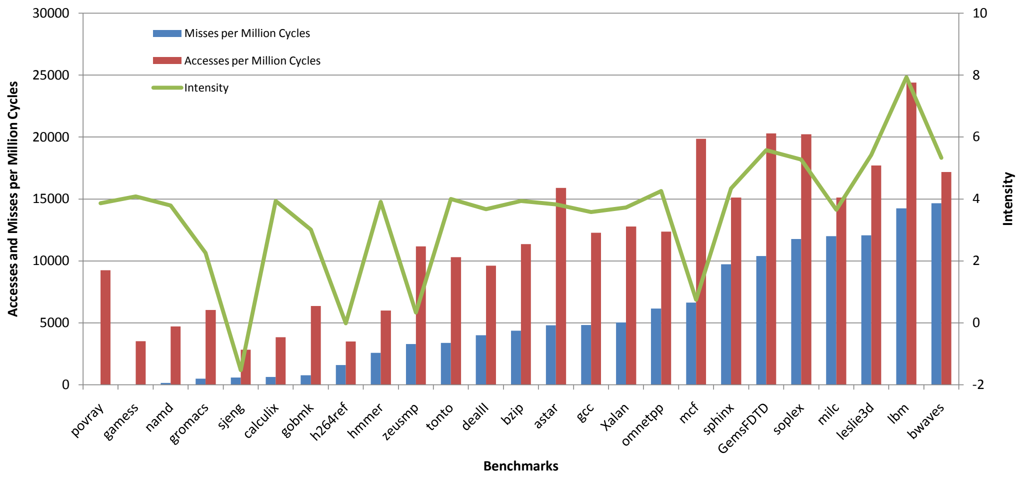


Figure 3.5: Miss Rates, Access Rates and Suffering Inflicted

have the highest correlation with intensity. Multiple regression analysis was run on the data to obtain a correlation between suffering inflicted and the pair of cache access rate and cache miss rate. This analysis produced the data present in Table 3.3.

Of these, the most promising were the L3 metrics. However, capturing both the L2 and L3 cache metrics live in a real scheduler would be cumbersome, which is why we decided on the L2 metric for intensity as well. The L2 metrics, while not as strongly correlated as the L3 metrics, will already be gathered for sensitivity and are thus a better choice.

Table 3.4 lists the statistical details of the L2 cache metric. In Table 3.4, we see that there exists a correlation of 32.20% between the application's intensity and the combination of its cache access rate and miss rate, i.e. 32% of the variability in an application's intensity can be explained by the application's cache access rate and miss rate. At a significance level of 0.050, i.e. $\alpha = 0.050$, with 25 data points from the benchmarks, $F_{0.050,2,22} = 3.44$.

Since the observed f value, $5.33 > F_{0.001,1,23}$, we can safely reject the null hypothesis, which states that there is no relationship between the intensity and the combination of the cache access rate and miss rate. Thus, we can assume that there is an approximate linear relationship between the degradation in an application's IPC and its cache access rate and miss rate. Confirming this conclusion is the low P-value of 0.01390, which strongly contradicts the null hypothesis.

The intuitive explanation for the access rate is straightforward: the more an application uses the cache, the more likely it is to occupy the cache and deny other applications a fair

Per Million Cycles	
Metric	**R square**
L1 accesses and misses	27.90
L1 misses	16.34
L2 accesses and misses	32.20
L2 misses	31.06
L3 accesses and misses	39.76
L3 misses	31.90
Per Million Instructions	
Metric	**R square**
L1 accesses and misses	0.60
L1 misses	3.04
L2 accesses and misses	29.21
L2 misses	6.08
L3 accesses and misses	26.90
L3 misses	15.57

Table 3.3: Cache Metrics and Intensity Correlation

Statistic	Value
R square	32.2030
f	5.22
P-value	0.01390

Table 3.4: Intensity and Correlation between Access Rate and Miss Rate

share of the cache. Since the miss rate determines how much data the application evicts from the cache, it is straightforward to see how a higher miss rate might negatively impact a co-scheduled application by evicting some of its data.

Thus, it seems as though the combination of the last-level cache access rate and miss rate is a reasonable metric for intensity.

3.3.4 Implications

In this section, we rank applications from the SPEC 2006 suite in terms of their access rates and miss rates and examine the sensitivity and intensity quotients of these applications.

Benchmark	Misses per million instructions
povray	7.76
gamess	23.6
namd	112.64
calculix	512.39
sjeng	627.71
gromacs	773.26
gobmk	893.52
h264ref	1530.42
hmmer	2454.6
tonto	3139.75
dealII	3900.72
zeusmp	4895.84
bzip2	5009.23
xalancbmk	6380.35
gcc	8578.31
astar	12765.85
leslie3d	13703.94
omnetpp	14357.79
GemsFDTD	14631.46
bwaves	15718.76
sphinx3	16045.37
milc	20197.31
lbm	25418.42
soplex	29994.88
mcf	41846.09

Table 3.5: L2 Cache Miss Rates of SPEC CPU 2006 Benchmarks

3.3.4.1 Sensitivity

If an application's cache miss rate determines its sensitivity, then applications with a high cache miss rate should suffer more when co-run with devils than with non-devils. This is one case of an extremum with our theory: co-scheduling an application that is most likely to suffer with an application that is most likely to inflict suffering should show us maximal suffering. This does turn out to be the case, as evinced by the data in Table 3.6.

Table 3.5 shows us the applications sorted by their cache miss rate. The median value of this set of data is *bzip2*, with a miss rate value of 5009.23. Based on the animalistic classification, we can identify *sphinx*, *milc*, *lbm*, *soplex* and *mcf* as potential devils, since they fall at the higher end of misses per million instructions. Calculating the sensitivity of applications when run with these potential devils gives us the data in Table 3.6

Benchmarks	Average sensitivity to devils	Average sensitivity to non-devils
Lower half of miss rates	2.379	2.154
Higher half of miss rates	6.241	4.450

Table 3.6: Median-separated Sensitivity Impact

From the table, we can see that applications with high cache miss rates are more sensitive, suffering at least twice as much as their low miss rate counterparts. Table 3.6 also tells us that regardless of its miss rate, an application suffers more when co-scheduled with a devil instead of a non-devil.

3.3.4.2 Intensity

Here, we examine the effects of an application on other applications as a consequence of its miss rate and access rate. Sorting the applications by their cache miss rates gives us the data in Table 3.7, with *bzip2* as the median value of 4370.28.

Similar to our calculations for sensitivity above, in Table 3.8 we see that applications with higher miss rates and access rates are more intensive than their lower miss rate counterparts, with an almost 60% more increase in intensity from the lower miss rate to the higher miss rate applications.

Benchmarks	Misses per million cycles	Accesses per million cycles	Average intensity
povray	8.96	9247.01	4
gamess	30.46	3515.19	4
namd	154.02	4724.84	4
gromacs	498.69	6045.77	2
sjeng	589.46	2828.86	-2
calculix	641.22	3836.45	4
gobmk	772.33	6359.95	3
h264ref	1596.6	3498.31	0
hmmer	2581.85	5986.97	4
zeusmp	3304.89	11175.3	0
tonto	3387.71	10305.94	4
dealII	4004.35	9620.47	4
bzip	4370.28	11358.79	4
astar	4816.75	15890.2	4
gcc	4831.18	12263.34	4
Xalan	5036.77	12776.92	4
omnetpp	6159.94	12374.06	4
mcf	6638.98	19865.33	1
sphinx	9732.63	15118.44	4
GemsFDTD	10398.59	20297.46	6
soplex	11767.45	20222.94	5
milc	12002.13	15103.68	4
leslie3d	12076.39	17709.44	5
lbm	14242.4	24390.37	8
bwaves	14665.58	17171.63	5

Table 3.7: L2 Cache Miss and Access Rates of SPEC CPU 2006 Benchmarks

Benchmarks	Average intensity
Lower half of miss rates	2.713
Higher half of miss rates	4.429

Table 3.8: Median-separated Intensity Impact

Chapter 4

Evaluation

So far, we have examined the correlation between an application's solo cache performance events and its sensitivity and intensity. Since the ultimate goal of the thesis is to build a live scheduler that can co-schedule applications based on live, runtime characteristics, we need to identify and examine the relationship between an application's intensity and sensitivity and its live cache characteristics when executing in a co-schedule.

4.1 Benchmarks

Since running all the benchmarks in the SPEC CPU 2006 suite against each other to gather live data would be prohibitively time-consuming, 10 benchmarks were chosen that served as a representative subset of the entire suite. Borrowing the clustering spanning tree methodology from Blagodurov et al. [1] gave me the following 10 benchmarks, which form a representative set spanning the entire gamut of SPEC benchmarks.

Class	**Members**
Devil	lbm, mcf, milc, sphinx, soplex
Turtles	povray, gamess, namd
Others	gcc, gobmk

Table 4.1: Representative Subset of SPEC CPU 2006 Benchmarks

In Section 3.3.1 we discovered that the metric with the highest correlation to sensitivity was the L2 cache metric. For this reason, we consider the L2 cache when gathering data for our live scheduler.

4.2 Dual Application Case

In this section, we examine the co-scheduling issues that arise when two co-scheduled applications share a last-level cache. Each pair of co-scheduled applications shares the L3 cache on the AMD Barcelona processor used in our experimental system above. The other two cores on the quad core Barcelona processor are idle, with no applications scheduled on them.

Each benchmark in the above set was run against the other nine and relevant data such as the L2 cache accesses, L2 cache misses and the Instructions Per Cycle (IPC) were gathered during a live run. An application's IPC degradation in a co-schedule was calculated as a percentage change over its solo IPC. The average IPC degradation of a benchmark was the average of its individual IPC degradations in all its co-schedules. Similarly, the L2 cache reads and misses were measured every million cycles as well. The average reads and misses per million cycles are calculated over all the co-schedules an application participates in.

Analogous to the sensitivity and intensity methodology followed in Section 3.3, this section seeks to correlate the average degradation in IPC of a benchmark to its average L2 cache accesses and misses, both per million cycles and per million instructions, when co-scheduled with another benchmark.

Table 4.2 lists the R-square values of the L2 cache parameters for the dual application case, correlated to the IPC degradation.

Per Million Cycles	
Metric	**R square**
L2 accesses	46.47
L2 misses	50.00
Per Million Instructions	
Metric	**R square**
L2 accesses	25.43
L2 misses	46.83

Table 4.2: L2 Cache Metrics and Sensitivity Correlation: Dual Case

In this case, the highest R-square was of the L2 misses per million cycles and IPC degradation. This value is 50.00, which means that 50% of the variation in the IPC can be explained by the L2 misses per million cycles figures. However, the L2 misses per million instructions metric has an R-square value of 46.83, which is the second best metric. Since my goal was to build a general-purpose scheduler that would work well for different numbers of co-scheduled applications, it made sense to stick with the L2 misses per million instructions as an indicator of sensitivity here.

Similarly, for intensity, we calculated the R-square value of linear regression of L2 cache accesses and misses, as well as on misses alone. An application's average intensity was calculated as the average of its individual IPC degradations inflicted on its co-scheduled partners. The L2 accesses and misses were gathered every million cycles and instructions as well. The results are shown in Table 4.3.

Per Million Cycles	
Metric	**R square**
L2 accesses and misses	95.96
L2 misses	95.31
Per Million Instructions	
Metric	**R square**
L2 accesses and misses	92.28
L2 misses	85.04

Table 4.3: L2 Cache Metrics and Intensity Correlation: Dual Case

We see that the highest R-square was of the L2 accesses and misses per million cycles and IPC degradation. This value is 95.96, which means that 95.96% of the variation in the IPC damage inflicted by an application can be explained by the combination of L2 accesses and misses per million cycles figures. The correlation here is higher than the correlation in Section 3.3.3, suggesting that the live numbers provide a more accurate reflection of an application's intensity.

4.3 Quad Application Case

This section examines co-scheduling four applications on a quad-core processor. All four cores on the quad-core Barcelona processor have an application scheduled on them, sharing the common L3 cache.

Selecting 4 applications at a time to co-schedule, out of 10, gives us a total of $\binom{10}{4}$ or 210 combinations. Each combination was run and each application's L2 cache reads, misses and IPC were gathered. An application's IPC degradation was calculated as before, as a percentage change over its solo IPC. The average IPC degradation of a benchmark was the average of its individual IPC degradations in all its co-schedules. The L2 cache reads and misses were measured every million cycles, with the average reads and misses, both per million cycles and per million instructions being calculated over all the co-schedules an application participated in.

Table 4.4 lists the R-square values of the L2 cache parameters for the quad application case, correlated to the IPC degradation.

Per Million Cycles	
Metric	**R square**
L2 accesses	49.76
L2 misses	49.11
Per Million Instructions	
Metric	**R square**
L2 accesses	43.48
L2 misses	67.50

Table 4.4: L2 Cache Metrics and Sensitivity Correlation: Quad Case

In this case, the highest R-square was of the L2 misses per million instructions and IPC degradation. This value was 67.50, which means that 67.50% of the variation in the IPC can be explained by the L2 misses per million instructions figures. This matches the prediction of the model in Section 3.3.2.

As done for the dual application case above, we determined the correlation between an application's intensity and its L2 cache accesses and misses, as well as its misses alone. The results are shown in Table 4.5.

Per Million Cycles	
Metric	**R square**
L2 accesses and misses	35.33
L2 misses	2.12
Per Million Instructions	
Metric	**R square**
L2 accesses and misses	49.43
L2 misses	0.21

Table 4.5: L2 Cache Metrics and Intensity Correlation: Quad Case

We see that the highest R-square was of the L2 accesses and misses per million instructions and IPC degradation. This value is 49.43, which means that 49.43% of the variation in the IPC damage inflicted by an application can be explained by the combination of L2 accesses and misses per million instructions figures. Since our goal is a general-purpose scheduler that works for both the dual-case and quad-case workloads, we consider the metric of L2 accesses and misses per million cycles, simply because it's a better match in both the model in Sections 3.3.2 and 4.2. Using the L2 accesses and misses per million instructions for the quad-case would defeat the goal of building a general scheduler and tie us down to specific cases for specific workloads. The L2 access and misses per cycle metric has an R-square value of 35.33, which implies that 35.33% of the variation in the IPC damage inflicted by an application can be explained by the combination of L2 accesses and misses per million cycles figures.

4.4 Maximum Possible Gain

If we reconsider the example in Section 1.3, reproduced below for convenience, we see that three schedules are possible.

#	Pair 1	Pair 2	Pair 1 degradation over solo	Pair 2 degradation over solo	Total IPC degradation over solo
1	mcf + milc	omnetpp + lbm	9.99 + 4.4	21.95 + 10.16	55.50
2	mcf + lbm	milc + omnetpp	19.16 + 9.13	11.79 + 13.53	53.61
3	mcf + omnetpp	milc + lbm	17.41 + 8.73	18.96 + 15.05	60.15

Table 4.6: Total Schedule Degradation over all Combinations

The optimal schedule here is the schedule with the least total IPC degradation, Schedule 2 with a total IPC degradation of 53.61. The worst schedule or the schedule with the highest IPC degradation is Schedule 3, at 60.15. The difference between the best and worst schedules in this case is $60.15 - 53.61 = 6.54$. There are 210 ways to pick 4 applications out of the 10 we chose above, and each of the 210 combinations has 3 possible co-schedules, of the form shown in Table 4.6.

Section 3.3.1 contains the pair degradation data for this subset of 10 benchmarks. In order to determine the maximum possible gain of each combination, the optimal schedule for each triplet was calculated, along with the worst-performing combination from each triplet. These schedules were calculated with a program created as part of this thesis, and described in more detail in the next section.

Figure 4.1 shows the 210 combinations, sorted by the difference between the best and the worst possible co-schedule for each combination. At the lower end of the scale, we see co-schedules where the difference between the optimal and the worst schedule is negligible. At the higher end of the scale, we see co-schedules where the difference between the optimal schedule and the worst schedule is substantial, i.e. an optimal schedule can deliver a substantial IPC improvement over a sub-optimal schedule. The possible gains are calculated using the following formula:

$$100 * (MaximumDegradation - MinimumDegradation)/(MaximumDegradation)$$

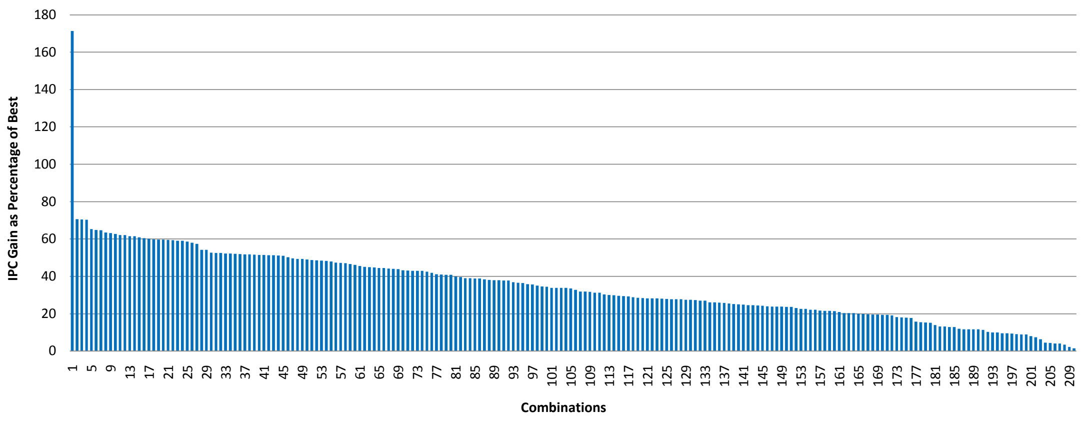


Figure 4.1: Co-schedules Sorted by Maximum Possible Gain

4.5 Optimal Scheduling with *clingo*

In Table 4.6, we saw that the optimal schedule for four applications, where two can be co-scheduled together such that they share a cache, is the schedule with the minimum IPC degradation. Finding this optimal schedule for n applications, as n gets larger, is non-trivial and results in a combinatorial explosion.

Borrowing some tools from Knowledge Representation, an area of Artificial Intelligence, I was able to create an application that would give me an optimal schedule for n pairs of applications in a reasonable amount of time. The language used was *clingo*, a declarative language that accepts conditional statements, akin to Prolog, and outputs the answer, if one exists. Such programs are referred to as Extended Logic Programs, and are used primarily as a search tool for NP-hard search problems.

My clingo application's input was the IPC degradation between two applications, modeled as a corun. Building on the example above, the IPC degradation between *mcf* and *milc* would be modeled as:

```
corun(mcf,milc,9.99)
```

The clingo application is brute force and finds all possible combinations of pairs, such that no application is left out or counted twice. The total degradation of a combination is computed as the sum of the IPC degradations of all its constituent pairs. Once all such possible combinations have been generated, the combination with the minimum total degradation is chosen as the optimal schedule.

4.6 Calculating Sensitivity and Intensity

We have seen how sensitivity and intensity are correlated to L2 cache accesses and misses. However, sensitivity and intensity need to be derived from these metrics using formulae that can be computed efficiently and quickly, in a live scheduler.

Several different methods to derive sensitivity and intensity from L2 cache accesses and misses were explored, with the two most promising being listed below:

1. Arithmetic Formulae (method A).
   (a) Sensitivity = misses, Intensity = accesses + misses
2. Equidistant Step Function (method B). This equidistant step function works by finding the maximum and minimum accesses, then dividing the interval between them into 10 equidistant intervals. Each application's sensitivity equals the interval it lies in, from 1 to 10. Similarly, the intensity is calculated based on the sum of accesses and misses.
   (a) Sensitivity = misses
   (b) Intensity = accesses + misses

4.6.1 Pain Metric

The sensitivity and intensity of each application were factored into a metric known as the pain metric. The pain metric was coined by Sergey Zhuravlev in Blagodurov et al. [1]. The pain caused by running two applications, A and B can be defined as:

$$Pain(A, B) = A.sensitivity * B.intensity + A.intensity * B.sensitivity$$

This gives us a heuristic of the damage caused by running applications A and B together. Intuitively, A's sensitivity is compromised by how intensive B is, and vice versa.

Recall that the clingo application from Section 4.5 took inputs of the form

```
corun(A,B,n)
```

where n was the IPC degradation between two applications A and B. This value n can be replaced by the pain metric to test the efficacy of the methods used to derive sensitivity and intensity. I tested each method of deriving the sensitivity and intensity of an application with the following algorithm, explained in pseudocode:

```
Run the clingo application and get the optimal schedules for all 210
   co-schedules with real IPC degradation
Calculate sensitivity and intensity according to each method above,
   from method A to E
Calculate the pain metric between two applications as described above,
   from the sensitivity and intensity
Create the corun data for each pair of applications
Feed these coruns to the clingo application
Get all 210 optimal co-schedules from these coruns created with the pain
   metric
Calculate the percentage of matches with the optimal schedule from real
   data, i.e., check how many of these optimal co-schedules are present
   in the optimal schedules created with the real data.
```

The percentage of matches against the L2 metrics that had the highest correlation to intensity and sensitivity was determined, as shown in Sections 3.3.2 and 3.3.3. The highest matches from the results of these comparisons are shown in Table 4.7.

From the table, we see that the most accurate method of deriving sensitivity and intensity is Method B using the L2 cache metrics, on a per million instructions basis. This confirms the fact that the L2 cache metrics were shown to have higher correlation to IPC degradation

Per Million Cycles		
Cache Metric	**Method**	**Percentage of Matches**
L2	A	40.65
L2	B	48.05
Per Million Instructions		
Cache Metric	**Method**	**Percentage of Matches**
L2	A	44.55
L2	B	49.15

Table 4.7: Pain Metric Accuracy

in Section 3.3.1 as well. Going forward, the L2 cache metrics and method B will be used as inputs to the live scheduler.

Thus, we now have a formula to derive sensitivity from the L2 cache misses per million instructions:

```
Bucket Size = ( Max L2 misses - Min L2 misses) / 10
For all intervals
    Sensitivity of application = Floor ( ( L2 Misses ) / Bucket Size )
```

The maximum and minimum L2 miss values will be obtained from the performance counters. Measuring the L2 misses of each application will enable us to rank them, thereby giving us a maximum and a minimum value for the L2 misses.

Similarly, intensity can be derived by summing the number of L2 cache accesses and misses per million cycles:

```
EventsSum = L2 accesses + L2 misses
Bucket Size = ( Max EventsSum - Min EventsSum ) / 10
For all intervals
    Intensity of application = Floor ( ( EventsSum ) / Bucket Size )
```

These arithmetic formulae are intuitive to understand and simple to calculate in a live scheduler.

4.7 Experimental Subset

Since it was not feasible to test all 210 combinations with the live scheduler, an evenly spaced subset of the combinations was chosen, from the list sorted by maximum possible gain. Figure 4.2 shows the combinations that were tested with the live scheduler, and includes co-schedules with different amounts of gain.

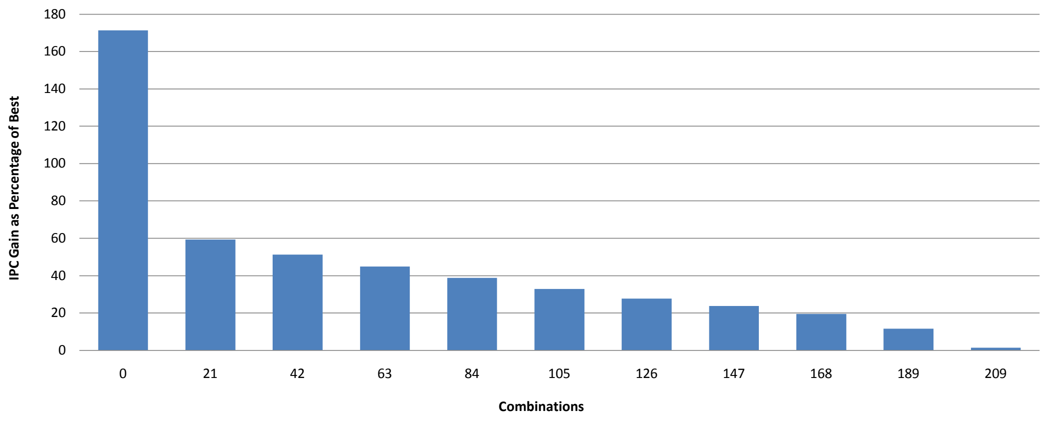


Figure 4.2: Experimental Subset of Co-schedules

Table 4.8 lists the 11 combinations that make up the experimental subset. Each pair of applications on a processor runs on its own core, with each pair sharing the L3 cache. Nothing is scheduled on the remaining two unused cores on the processor.

#	Maximum Possible Gain Percentage	Processor 1	Processor 2
1	171.34	gamess povray	gobmk namd
2	59.36	namd soplex	povray sphinx
3	51.3	gobmk milc	lbm mcf
4	44.84	gamess lbm	gcc milc
5	38.86	gcc povray	namd sphinx
6	32.81	gamess povray	lbm mcf
7	27.69	gobmk namd	milc povray
8	23.75	gcc mcf	gobmk sphinx
9	19.48	gcc mcf	gobmk milc
10	11.57	gobmk soplex	lbm sphinx
11	1.41	gamess sphinx	gobmk namd

Table 4.8: Maximum Possible Gain in Experimental Subset: Model

As previously stated, the column Maximum Possible Gain Percentage is calculated using the formula:

$$100*(MaximumDegradation-MinimumDegradation)/(MaximumDegradation)$$

4.8 Live Scheduler

The live scheduler is a user-level process that is responsible for the creation, scheduling and termination of processes. It has a static list of the ten representative benchmarks we chose in Section 4.1. It creates a new thread for each benchmark and is responsible for restarting a benchmark when it finishes, as long as the longest-running benchmark is still running. When all the benchmarks have been restarted at least once, the live-scheduler terminates all spawned threads and exits. Before exiting, the live scheduler logs all the measured values to a log file, which is used to produce the results in the next chapter.

While the benchmarks are executing, the live scheduler works in two phases: *sampling* and *placement*.

4.8.1 Sampling

In the sampling phase, the scheduler monitors the hardware performance counters of each running thread once per millisecond and tracks the following relevant events:

- L2 cache accesses
- L2 cache misses
- Instructions executed
- Cycles elapsed

These values are stored as the program executes and used in the shifting phase below. All the events are used in the placement phase that follows.

4.8.2 Placement

The placement phase handles the bulk of the scheduling decisions. Each application that is currently running is stored in a list, along with its four events measured from the performance counters. The placement phase runs once per second, but this delay is configurable via a parameter passed in to the live scheduler. During the placement phase, the scheduler wakes up and sums the values that were gathered during the sampling phase. It calculates the sensitivity and intensity of each application in the list, according to the formulae defined in Section 4.6.1. It then creates two lists of running applications and sorts them, one by sensitivity, the other by intensity. Now that we have the lists of sensitive and intensive applications, we need to schedule them based on a scheduling algorithm. An elegant, greedy algorithm for scheduling suggests itself and is described in detail below.

Running the placement phase at more frequent intervals causes more overhead in the scheduler. For this reason, the placement phase cannot be run at intervals lesser than one second. On the other hand, running the shifting phase at intervals that are too large prevents the live scheduler from migrating applications that should be co-scheduled sooner, thereby reducing the potential gains that can be observed.

4.8.3 Greedy Algorithm

Intuitively, a greedy algorithm that schedules the most intensive application with the least sensitive application should reduce potential IPC degradation. This is because co-scheduling an intensive application that has a high potential for inducing performance degradation along with a sensitive application, one that has a low potential of experiencing performance degradation should reduce the overall potential degradation. The pseudocode for such an algorithm is as follows:

```
P <- List of applications sorted by sensitivity in ascending order
Q <- List of applications sorted by intensity in descending order

While P and Q are not empty
    A <- Pop first ( P )
    B <- Pop first ( Q )
    If A != B
     and are not already co-scheduled
        Create co-schedule with A and B

For each co-schedule
    Bind applications in co-schedule to cores that share a cache
```

As described above, the live scheduler is responsible for creating and maintaining the lists of applications, sorted by their sensitivity and intensity. Each application's sensitivity and intensity are calculated using the metrics described in Section 4.6. Each application is exclusively bound to the core it is scheduled on.

4.8.4 Migration Decision

During the shifting phase, if the live scheduler decides to co-schedule two applications, one of the applications has to be migrated to share a core with the other. As per our greedy algorithm, since one of the applications is highly intensive and one is highly insensitive, the live scheduler has to decide which one to migrate. The live scheduler migrates the highly insensitive application from the core its running on, to the core the extremely intensive application is running on. This is because the insensitive application is unlikely to be affected strongly by the core migration, given its insensitive nature. This holds for both the dual-case and the quad-case workloads.

4.8.5 Replacement Decision

When a migration decision has been made by the live scheduler, a replacement decision must soon follow. This scenario is explained in the context of the dual-case and quad-case workloads below.

4.8.5.1 Dual Application Case

In the dual-case, when an application is switched from one pair to another, deciding which co-scheduled application from the other pair should take its place is trivial. Consider the dual-case workload with four applications that make up the two pairs: A, B, C and D.

1. A and B
2. C and D

Applications A and B form one pair, while applications C and D make up the second pair. During the shifting phase, if the live scheduler decides that C should now be co-scheduled with A, B must replace C to be end up being paired with D. After the switch, we end up with the two pairs:

1. A and C
2. B and D

4.8.5.2 Quad Application Case

The quad-case is not so straightforward. Since there are eight applications that form the two quartets, we have to decide which application must replace the newly co-scheduled applications. Consider the two quartets, co-scheduled as follows:

1. A, B, C and D
2. E, F, G and H

If the live scheduler determines that F is the most insensitive application and A is the most intensive, then F has to be migrated to the core A is executing on. However, in order to co-schedule F with A, one of B, C or D must take F's place in the second quartet and at the same time free up a core on A's processor. In this case, the live scheduler picks the least intensive application out of B, C and D as the replacement candidate. The rationale behind picking the least intensive is essentially the one that drives the greedy algorithm. By moving the least intensive application onto a different core, we leave the most intensive applications co-scheduled with the least sensitive application. This localizes the damage caused by the highly intensive applications to the co-scheduled applications on that core, one of which is now the least sensitive application that was migrated from another core. In our example, let the least intensive application on the first core be C. After C is swapped with F, the quartets now look like the following:

1. A, B, F and D
2. E, C, G and H

Chapter 5

Results

This chapter discusses the results of running the live scheduler against the naïve default Solaris scheduler which allocates applications to cores at random. The UMA results are described first, consisting of the outcome of the dual-case workloads. This is followed by the results from the NUMA system, which consists of both the dual-case and the quad-case workloads. In both the UMA and NUMA cases, we examine the results in terms of the group results from a combination, followed by an individual application's results. Both the NUMA and UMA systems were described in Section 3.1.

5.1 Non-NUMA

The results of running the live scheduler on the UMA system show the efficacy of the live scheduler's ability to extract performance gains from workloads, where such performance gains are extractable. When a workload is equally composed of applications that are devils and non-devils, the live scheduler is able to effect maximal performance gains. By separating the most sensitive and the most intensive applications, i.e., devils from each other, the live scheduler prevents the devils from degrading each others' performance. This is seen in all combinations with two devils and non-devils, where the live scheduler delivers up to 100% of the maximum possible gain in some combinations. Additionally, the live scheduler is also more stable with regards to application IPC, with a lower average standard deviation of 0.01, compared to the random scheduler's average standard deviation of 0.06.

When more than half the applications in a workload are intensive, the live scheduler follows its chosen strategy of separating the most intensive devils by co-scheduling them in

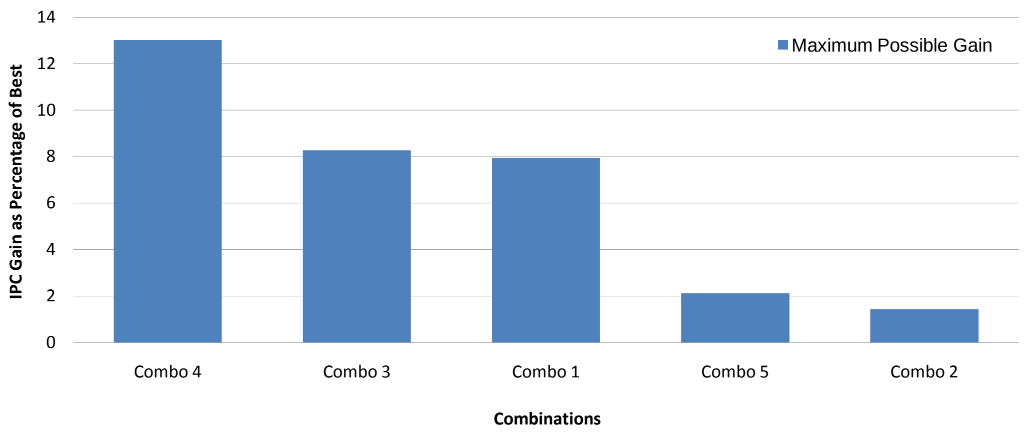


Figure 5.1: Maximum Possible Gain of Co-schedules

different co-schedules. This strategy, which is based on a simple heuristic obtained from the hardware performance counters proves to be too coarse to deliver maximal gains here. While more detailed heuristics might deliver more promising gains, these remain an avenue of future research.

In the non-NUMA section, the results are described in terms of co-scheduling four applications in the dual-case workload. Since applications display different behaviour when scheduled together on non-NUMA versus NUMA systems, we were unable to use the combinations I used on the NUMA system, described in Section 4.7. To avoid replicating the methodology on non-NUMA systems, I used the combinations used by the authors in Blagodurov et al. [1]. While their method differs from the method used in this thesis, as described in Chapter 2, the combinations they chose suit our method perfectly, being a healthy mix of devils and non-devils.

The maximum possible gain of each quartet was calculated, as seen in Table 4.8. Figure 5.1 shows the five combinations sorted by the maximum possible gain between co-schedules.

The maximum possible gain was calculated using the same formula as in Section 4.4:

$$MaximumPossibleGain$$
$$= 100 * (MaximumDegradation - MinimumDegradation)/(MaximumDegradation)$$

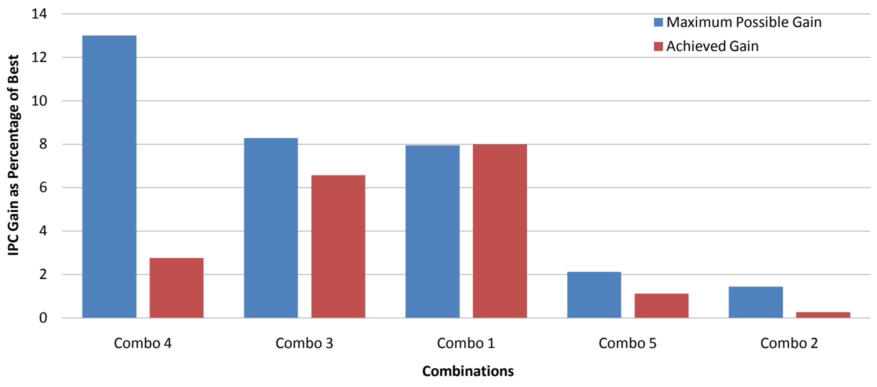


Figure 5.2: Comparison of Maximum Achievable Gain and Achieved Gain

The applications that comprise the five quartets that were run on the non-NUMA machine are shown in Table 5.1, also sorted by their maximum possible gain.

#	Maximum Possible Gain Percentage	Processor 1	Processor 2
4	13.02	lbm milc	sphinx gobmk
3	8.28	sphinx gcc	namd povray
1	7.94	soplex sphinx	povray namd
5	2.12	lbm milc	mcf namd
2	1.43	soplex mcf	povray gobmk

Table 5.1: Maximum Possible Gain in Experimental Subset: Measured

In the original work by Blagodurov et al. [1], Combinations 1, 2 and 6 used *gamess*, while I substituted *povray* for *gamess* due to compilation problems on our systems.

5.1.1 Dual-case Workload

This section describes the results of running the dual-case experiments on the non-NUMA system. For the random case, applications were started and pinned to cores at random, such that pairs of applications shared an L2 cache. The applications were not migrated during their runs, and all applications were restarted until the longest-running application terminated.

5.1.1.1 Combinations

We start off by comparing the gain achieved by a combination in a co-schedule, compared to the maximum possible gain of a co-schedule. Figure 5.2 shows the comparison of the live scheduler's performance gains against the random scheduler's performance gains. The red bars denote the average performance gains of the live scheduler, while the blue bars signify the random scheduler's maximal performance gains from Table 5.1. We see that in Combinations 3 and 1, the live scheduler nears or equals the maximum possible gain. Combinations 5 and 2 have very little gain to be extracted, valued at approximately 2% or less. For these two cases, there isn't a big margin of gains and the live scheduler does its best. However, Combination 4 has the highest possible gain and the live scheduler fails to deliver strongly in this combination. To understand this anomaly better, I decided to implement tracing in my live scheduler to see which co-schedules the live scheduler actually co-scheduled, from each combination.

Recall from Section 4.4 that each quartet has three possible co-schedules. One of these is the best-performing co-schedule with the highest IPC, one is the worst performer with the lowest IPC and one co-schedule is in between. Consider the third quartet from Table 5.1. Table 5.2 shows us the medium, best and worst performing co-schedules of this quartet along with their IPCs.

Co-schedule	Pair 1	Pair 2	Co-schedule IPC
1	soplex + sphinx	povray + namd	0.89
2	soplex + povray	sphinx + namd	0.97
3	soplex + namd	povray + sphinx	0.97

Table 5.2: Co-schedule Degradation for One Combination

Recall that both *soplex* and *sphinx* are devils, according to the taxonomic classification. We see that the two co-schedules that have the highest IPC are co-schedules where the devils are not co-scheduled together. Thus, the live scheduler should keep them apart and rarely co-schedule them on a shared cache, to effect maximal performance gains. Table 5.3 lists the co-scheduling combinations for each of the five quartets on the non-NUMA experimental system, sorted by co-schedule IPC. Co-schedule 1 has the lowest IPC, Co-schedule 2 is in the middle and Co-schedule 3 has the highest IPC for each quartet.

Figure 5.3 shows the percentage of time spent in each co-schedule for all combinations.

Name	Co-schedule 1	IPC	Co-schedule 2	IPC	Co-schedule 3	IPC
Combo 1	soplex + sphinx povray + namd	0.89	soplex + namd povray + sphinx	0.97	soplex + povray sphinx + namd	0.97
Combo 2	soplex + povray mcf + gobmk	0.62	soplex + gobmk povray + mcf	0.62	soplex + mcf povray + gobmk	0.63
Combo 3	gcc + sphinx povray + namd	0.96	gcc + povray sphinx + namd	1.04	gcc + namd povray + sphinx	1.05
Combo 4	lbm + gobmk milc + sphinx	0.46	lbm + sphinx milc + gobmk	0.48	lbm + milc gobmk + sphinx	0.52
Combo 5	milc + namd lbm + mcf	0.54	lbm + namd milc + mcf	0.54	lbm + milc mcf + namd	0.55

Table 5.3: Co-schedule Degradation for All Combinations

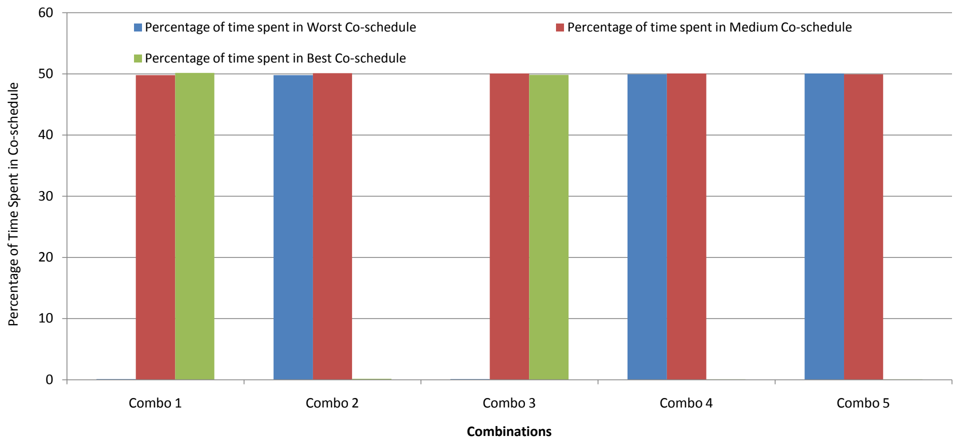


Figure 5.3: Percentage of Time Spent in Each Co-schedule

We see that in Combinations 1 and 3, the live scheduler finds the medium and best co-schedules, completely avoiding the worst. This strategy of separating the devils nets us performance gains that rival the maximum possible gain. The live scheduler schedules Combinations 2, 4 and 5 in an equal mix of their worst and medium co-schedules. As stated above, Combinations 2 and 5 have slight possible performance gains, while Combination 4 has the most possible gain. After a closer look at the applications that comprise Combination 4, we see the reason behind the live scheduler's sub-par performance. Combination 4 is composed of three devils: *lbm*, *milc*, *mcf* and one non-devil: *namd*. From Table 3.7, we see that *lbm* and *milc* are the two most intensive applications in this quartet. However,

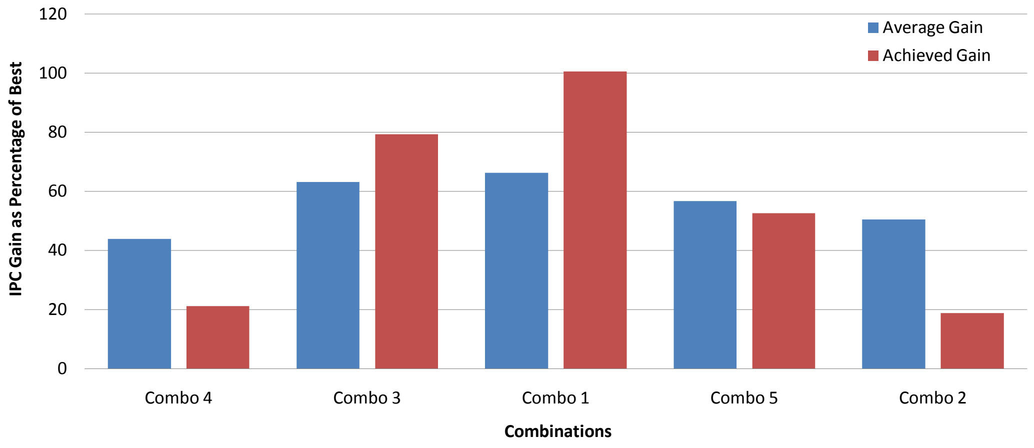


Figure 5.4: Improvement over Average Gain

from Table 5.3, we see that the highest-performing co-schedule actually has *lbm* and *milc* co-scheduled together. In this case, the live-scheduler's method works against itself by separating the most intensive devils. A less coarse-grained heuristic might be able to identify cases of devils sharing a co-schedule and benefiting each other, but the limitations of current hardware counter architectures prevent us from using such a fine-grained heuristic in a lightweight live scheduler such as this. A better heuristic that gives us deeper insight into this unexplained increase when co-scheduling certain intensive devils together is not investigated further in this thesis, but marked for future work.

Next, we compare the live scheduler to the average performance gains that the random scheduler would obtain. The average case is meant to mimic the default random scheduler in Solaris, which is agnostic of any cache characteristics of the applications it schedules. The random scheduler in Solaris doesn't pin applications to cores but occasionally migrates processes across cores if the need arises. My random case pinned the four applications to four random cores at startup, but did not migrate them at any point during their execution. All applications were restarted until the longest-running application ended. If these eleven quartets were run many times, the average IPC gain over all runs would be 50%, since the scheduler would randomly schedule either the best, the medium or the worst combination a third of the time equally. Since we have previously run all three possibilities for each combination, the best, worst and medium IPC of each combination is known to us. The medium IPC combination is not 50% for each combination, since in some cases the medium

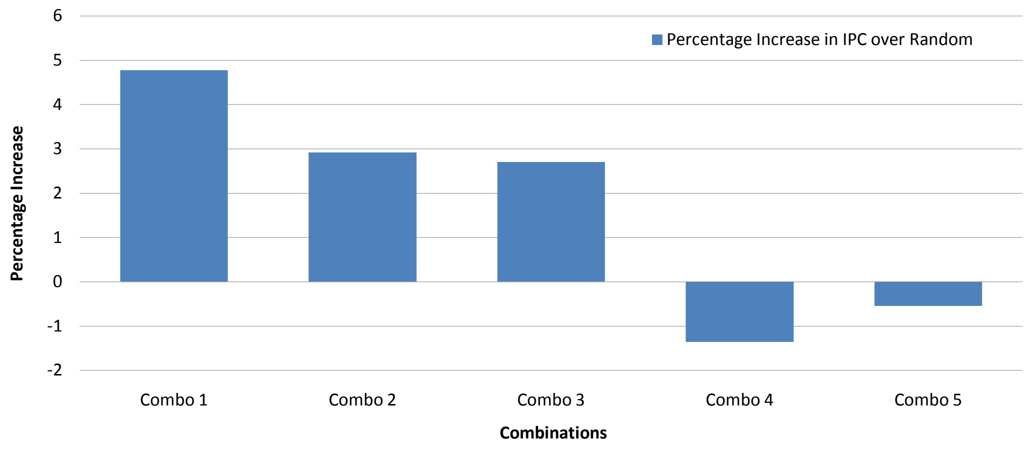


Figure 5.5: Percentage Increase in IPC: Combination

IPC is close to the best, and in others it is close to the worst IPC.

A closer look at Table 5.3 reveals that in all combinations except for Combination 3, the medium combination's IPC is almost that of the highest. This implies that the random scheduler will actually do quite well, since it will achieve close to the best IPC 66.67% of the time. However, the remaining 33.33% of the time is spent in the worst co-schedule. If the medium IPC was more evenly distributed, we would see a drop in the random IPC, since a third of its co-schedules would involve a co-schedule with lesser IPC. For these combinations, when averaged over all eleven quartets, the average gain is 56%, which is used in the equation below. Since each is equally likely to be scheduled, we get the average IPC gain when run with the random scheduler as:

Average performance gain over all runs

$= 100/3 + 68/3 + 0/3$

$= 56$

Figure 5.4 shows the comparison of the live scheduler's performance gain compared to the average performance gain that would be expected with the random scheduler. These figures mirror the ones in Figure 5.2, showing that the live scheduler exceeds the average performance in combinations 1 and 3, while failing to do so in combinations 2, 4 and 5. On average, the random scheduler will achieve 55% of the maximum possible gain, while the live scheduler will average 56% of the maximum possible gain. This shows that on average, the live scheduler's performance is just slightly below that of the random scheduler, when it comes to performance gain.

To see the percentage increase in IPC of a co-schedule when run with the live scheduler over the random scheduler, we look at Figure 5.5. We see that Combinations 1, 2 and 3 experience an increase in IPC, while Combinations 4 and 5 see a decrease when compared to the random scheduler. Combination 1 experiences the highest increase in IPC of 4.78%, while combination 4 faces the largest decline in IPC, approximately -1.26%. As stated above, this is due to the live scheduler finding the optimal co-schedules in combinations with an equal number of devils and non-devils. Dual-case workloads with more than two devils experience a decrease in IPC due to the live schedule's scheduling heuristic, which is not fine-grained enough to identify and co-schedule devils which might benefit from sharing a co-schedule. Globally, the decreases in IPC are fewer and smaller than the increases in IPC. This results in an overall average increase of 1.7% in IPC when compared to the random scheduler, which shows that the live scheduler is more effective at co-scheduling applications than the random scheduler.

Since the number of devils present in a combination plays a part in the performance improvement of the combination under the live scheduler, we now examine how combinations fare as they contain an increasing number of devils. We begin by listing the combinations classified by the number of devils they contain, in Table 5.4.

1 devil	2 devils	3 devils
Combo 3	Combo 1	Combo 4
	Combo 2	Combo 5

Table 5.4: Number of Devils in each Quartet

Figure 5.6 shows us that for combinations with one devil, the live scheduler is able to extract performance gains. The devil is isolated in a co-schedule and affects only the non-devil it is co-scheduled with. With two devils in a combination, the live scheduler separates them in two distinct co-schedules, taking care to never co-schedule devils with each other. Since there are two co-schedules and two devils, the live scheduler is able to keep them apart, based on the heuristics of sensitivity and intensity. When a combination has three devils, since there are only two pairs, the pigeonhole principle requires that the live scheduler is forced to pick a pair of devils to co-schedule. The two combinations with three devils

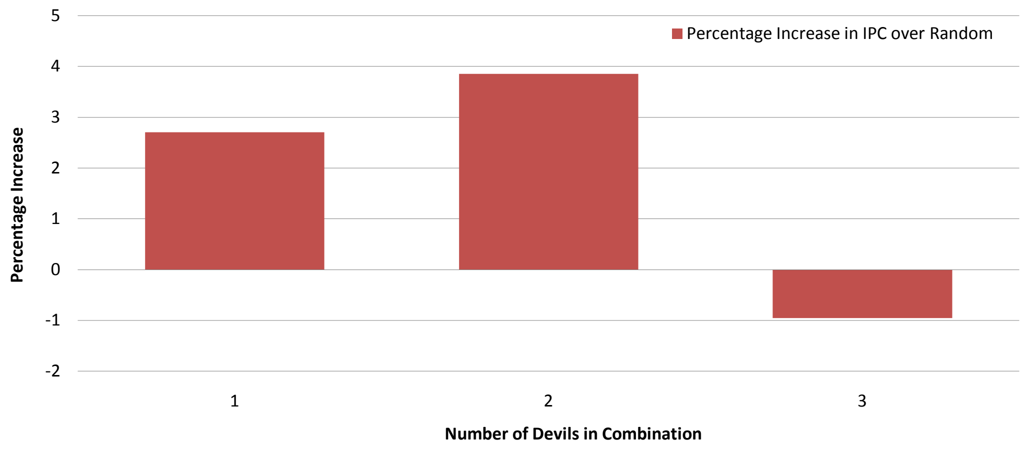


Figure 5.6: Performance as Number of Devils Varies

are combinations 4 and 5. As explained above for figure 5.3, these combinations exhibit performance gains when the most intensive devils are co-scheduled, an anomaly which the live scheduler's current heuristic is incapable of optimizing.

To confirm this, we drill down deeper and examine the breakdown of IPC increases and decreases within a combination as the number of devils varies, in Figure 5.7. When one devil is present, it experiences an increase in IPC, along with the non-devils which are co-scheduled such that the least sensitive is co-scheduled with the devil. This is exactly what happens with combination 3, which has only one devil: *sphinx*. Since *sphinx* has to be co-scheduled with the least-sensitive applications, *sphinx* should be co-scheduled with either *namd* or *povray*, but never with *gcc*. Recall from Table 3.5 that both *namd* and *povray* are at the top of the table, making them extremely insensitive applications. Proof that *sphinx* spends its co-schedules with only *namd* and *povray* but never *gcc* lies in Figure 5.3, which shows us that combination 3 spends half its time in the medium and best co-schedules, completely avoiding the worst co-schedule which pairs *sphinx* and *gcc* together. Since the devil is paired with non-devils, both categories show an IPC increase, leading to an total IPC increase of roughly 10%.

When two devils are present in a combination, the live scheduler sees to it that the devils are separated, one per co-schedule and their IPC increases significantly. Since each co-schedule has a devil now, the non-devils suffer an IPC decrease, but this decrease is of smaller magnitude than the corresponding increase in the devils' IPC. This is borne out by the two

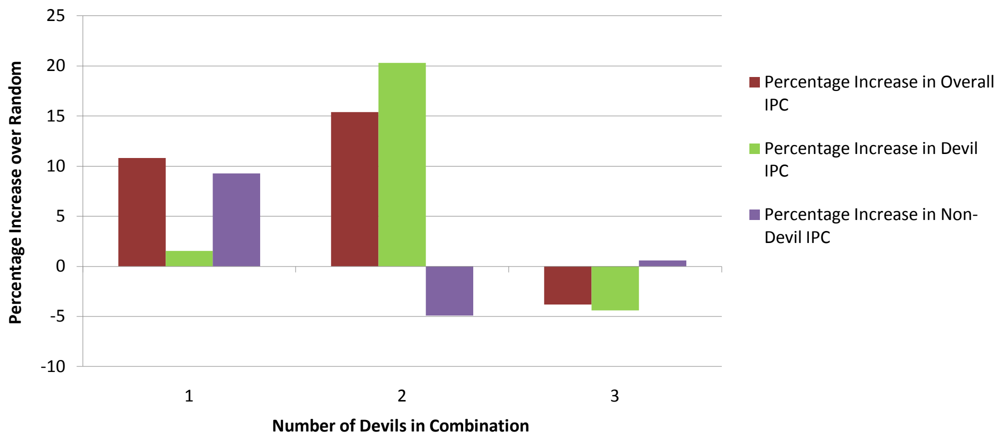


Figure 5.7: IPC Breakdown as Number of Devils Varies

combinations with two devils each, combinations 1 and 2. Again, referring to Figure 5.3, we see that at no time are the two devils co-scheduled together in both these combinations. In Combination 1, we see that the two devils, *soplex* and *sphinx* are never paired together, instead spending half their time with two non-devils, *namd* and *povray*. This story repeats itself with combination 2, where *soplex* and *mcf* are never co-scheduled together. This strategy pays off handsomely in combination 1, with both *soplex* and *sphinx* registering maximal gains. In Combination 2, by separating *mcf* and *soplex*, the live scheduler actually avoids the optimal co-schedule, but since the difference in IPC between the best co-schedule and the worst is minuscule, both *soplex* and *mcf* end up registering near-maximal gains in the end. Isolating the devils in this fashion enables them to reap the highest performance gains seen from the live scheduler. The non-devils that are co-scheduled with the devils end up experiencing a performance decrease, but this dwarfed by the magnitude of the devils' increase in IPC, leading to an overall increase of approximately 20%.

When three devils are present in a combination, the shortcomings of the live scheduler's heuristic are revealed. Since there are three devils and only two pairs of co-schedules, the live scheduler must make a decision about which pair of devils to co-schedule. Relying on its simple but herebefore effective heuristic, it simply separates the two most intensive applications. It thus falls short of recognizing cases when co-scheduling the two most intensive applications might actually deliver the maximal improvement. As stated above, this is because of the simplicity inherent in its heuristic, constrained as it is by the hardware performance counter architecture of today.

5.1.1.2 Individual Applications

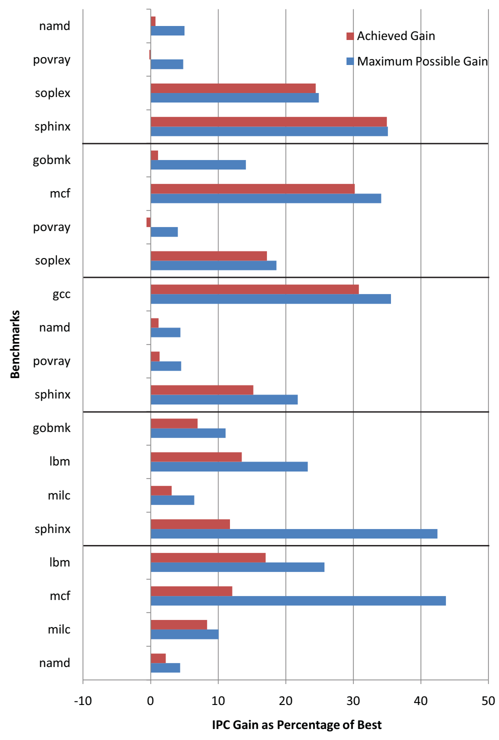
In this section, we analyze the individual performance gains of the twenty applications that comprise the five quartets. We start by comparing the performance of individual applications against the maximum possible gain of each application. The maximum possible gain of an individual application across all scheduled combinations, is calculated as stated at the beginning of this chapter:

$$MaximumPossibleGain$$
$$= 100 * (MaximumDegradation - MinimumDegradation)/(MaximumDegradation)$$

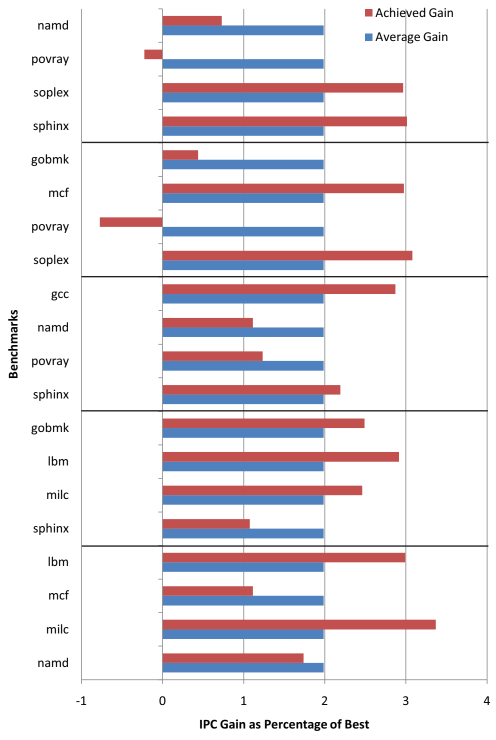
An individual application's gain is calculated by subtracting the lowest IPC of that application from its IPC under the live scheduler. Figure 5.8a shows us how each individual application's gain under the live scheduler stacks up against the maximum possible gain. This figure is the individual breakdown of the combinations in figure 5.2, and shows us which applications contributed to the overall increase or decrease of an co-scheduled combination. In combination 1, we see that *sphinx* and *soplex* achieve the maximum possible gain at the expense of *povray* and *namd*. *namd* does manage to score a slight percentage of the maximum possible gain, while *povray* actually slides in the other direction and registers a decrease in IPC. Perhaps the most interesting of these is combination 4, where we see *sphinx* falling short of the maximum possible gain by a wide margin, due to the inability of the live scheduler to optimize a combination with three devils. We also see that there are no steep declines in individual IPC. *povray* achieves a negative gain in two co-schedules, while the rest of the applications show a positive increase. Overall, the achieved gain seems to be close to the midway mark, but this is examined more accurately in the figure beside it.

The average performance gain that would be achieved by the random scheduler for individual applications is calculated by the following formula, similar to the one used for the combinations above:

Average performance gain over all runs
$$= 100/3 + 56/3 + 0/3$$
$$= 52$$



(a) Maximum Gain vs. Achieved Gain



(b) Average Gain vs. Achieved Gain

Figure 5.8: Comparison of Maximum Achievable Gain, Average Gain and Achieved Gain

Figure 5.8b shows the comparison of the normalized average gain under the random scheduler against the normalized gain from the live scheduler. An individual application's gain is calculated the same way as in Figure 5.8a. In the individual case, we see that the achieved gain surpasses the average gain in over half the applications. This figure is the individual counterpart of Figure 5.4, and shows how each application in a combination does against the average gains procured by the random scheduler. Recall that the average gain is quite high in all combinations except combination 3, since there is only a minuscule difference between the medium and the best co-schedules. This makes it easier for the random scheduler to stumble upon a higher-IPC co-schedule. As a result of that, the average gain of the live scheduler at 1.88% is close behind the average gain of the random scheduler at 1.98%. This figure shows that the the live scheduler's average performance gain is higher than the average performance gain of the random scheduler in eleven applications out of twenty.

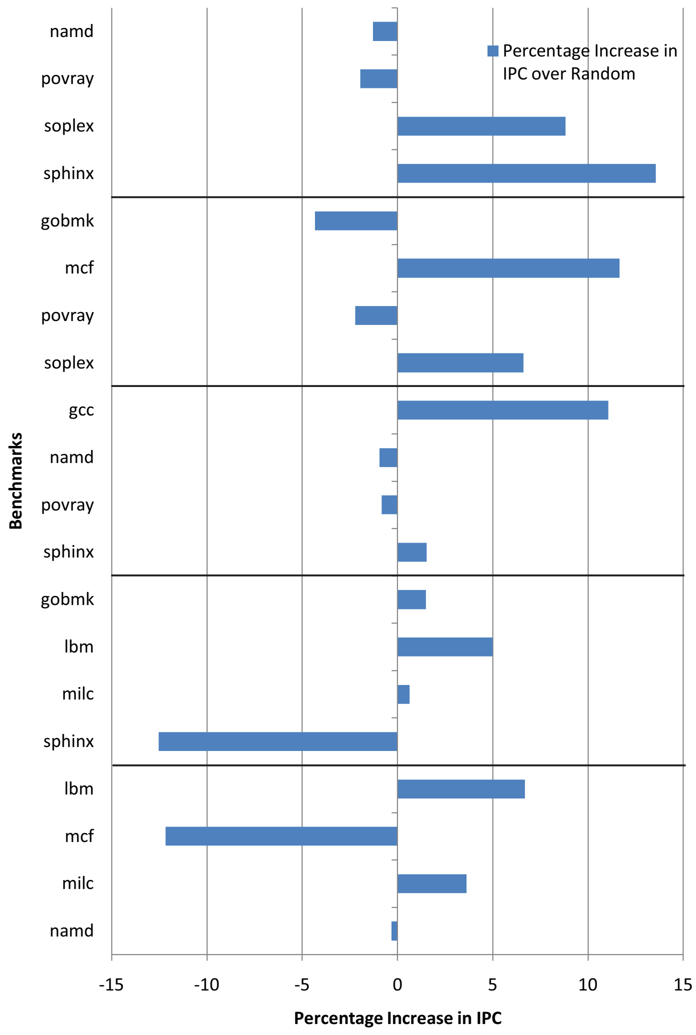


Figure 5.9: Percentage Increase in IPC: Individual

The percentage increase in the IPC of individual applications over the average IPC of the random scheduler is seen in Figure 5.9. Here, we see that that *povray* always suffers an IPC decrease, while devils in its quartet such as *soplex* or *sphinx* experience an IPC increase. This is a direct result of the greedy scheduling strategy employed by the live scheduler, which keeps devils apart from each other and co-schedules them with non-devils. In the bottom quartet we see the largest gainer, *sphinx*, registering a 13.55% increase, as a direct result of the live scheduler's greedy scheduling heuristic. Close to the top, *sphinx* is also responsible for the largest decrease, at 12.54%. This decrease occurs in Combination 4, due to the live scheduler's policy of separating the two most intensive devils. As stated before, the live scheduler's simple heuristic doesn't take into account cases where the most intensive can be co-scheduled together for mutual benefit, and thus separates the most intensive devils in Combination 4. By keeping *lbm* and *milc* apart, *sphinx* gets co-scheduled with *milc* and *lbm*, both of which contribute to its decline in IPC.

The IPC percentage increase numbers are summarized on a per-application basis in Figure 5.10. This figure shows us the increase in an individual application's IPC, averaged over all the co-schedules it participated in. The bars are colour-coded, with red bars signifying turtles, blue bars denoting devils and green bars for "others". We see that all the devils except for *mcf* post positive increases in IPC. All the turtles without exception show a decrease in IPC, no doubt due to the live scheduler's policy of pairing them with devils. The "others" category is a mixed bag, with *gcc* showing the highest overall average increase, at approximately 11%. On the other hand, despite belonging to the same category, *gombmk* shows a decrease in average IPC across all quartets. This shows that, across all combinations, the live scheduler favours devils and awards them high performance gains. Overall, turtles are subjected to a decrease in IPC, being intentionally sacrificed for the benefit of the devils.

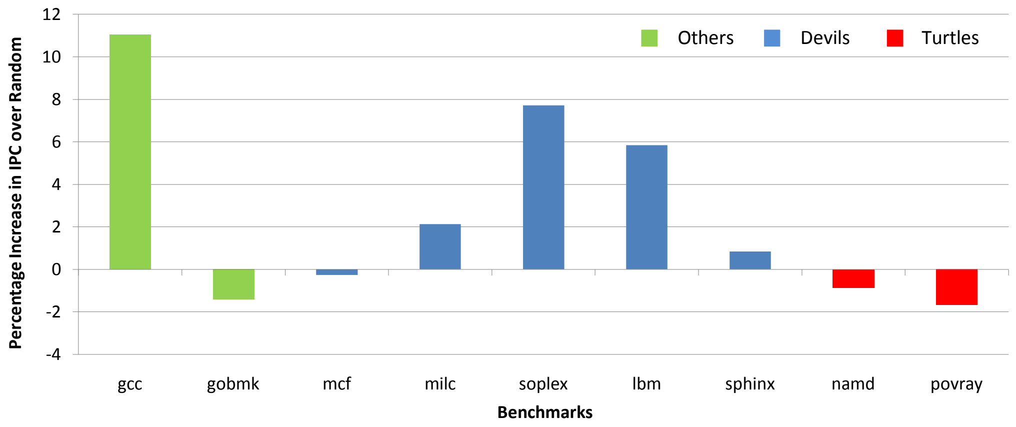


Figure 5.10: Percentage Increase in Individual Averages

A more concise representation of the animal classes' performance is seen in Figure 5.11. Under the live scheduler on an UMA system, the others and devils experience IPC increases, while the turtles experience IPC decreases. When considering clumped averages in this fashion, we see that the "others" category prospers under the live scheduler by approximately 5%. The devils experience approximately a 3% gain, while the turtles experience a 1% decrease. Thus we see that the live scheduler's scheduling policy benefits most of the animal classes, with the magnitude of increases outnumbering and exceeding the magnitude of decreases.

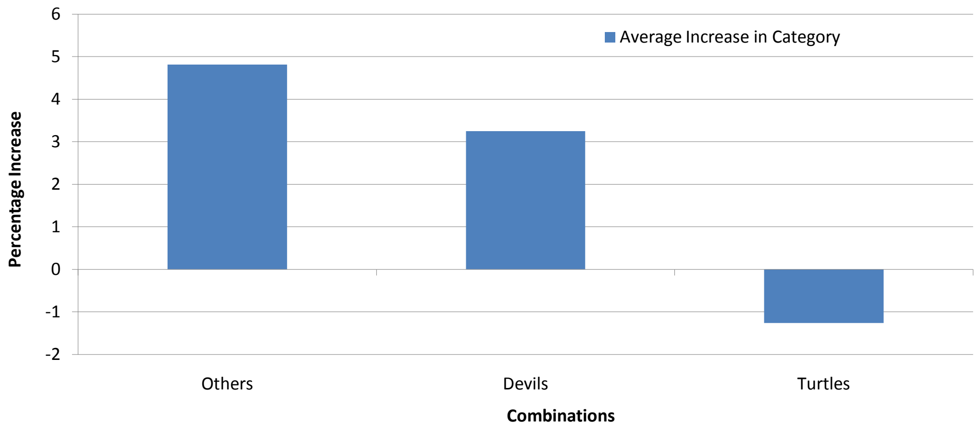


Figure 5.11: Percentage Increase in Animalistic Categories

We now compare the consistency of the live scheduler when compared to the random scheduler. In Figure 5.12, we see that the live scheduler has a smaller standard deviation than the random scheduler, calculated over all runs. This shows that the live scheduler delivers a more consistent performance. The higher standard deviation shows that in the long run, the random scheduler's results vary significantly, with it scheduling the worst-performing co-schedule a third of the time equally. We have seen that the live scheduler has a higher average IPC for both co-schedules and individual applications, as compared to the random scheduler. Coupled with the low standard deviation, the implication is that on average, the live scheduler will always deliver a consistently superior IPC, compared to the random scheduler. While the live scheduler may not always schedule the optimal co-schedule for the entirety of a workload's execution, it will almost always avoid scheduling the worst co-schedule for the entire duration of execution.

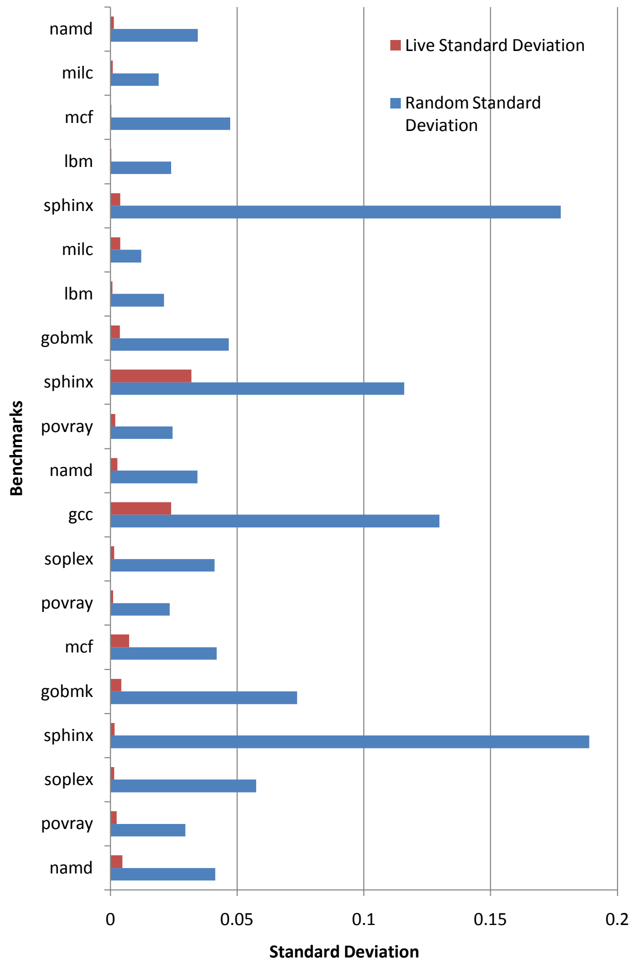


Figure 5.12: Standard Deviation of Live and Random Scheduler

5.2 NUMA

In the NUMA section, the results are described in terms of co-scheduling four applications together in the dual-case workload, and eight applications together in the quad-case workload. Recall that the NUMA system has memory node interleaving enabled, which distributes memory randomly across both local and remote memory bands. This is summed up in Figure 5.13.

We see that each core has a local and remote memory, the hallmark of a NUMA system. To access the local memory of a core, another core must go through its memory controller for that local memory. This makes the memory controller the bottleneck of the system,

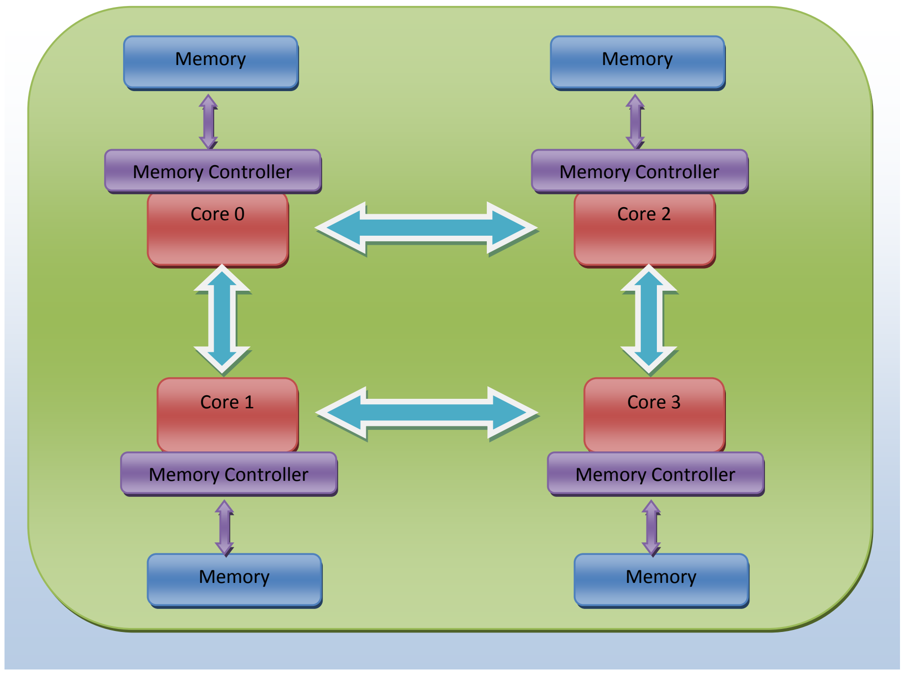


Figure 5.13: NUMA Memory Node Interleaving

with many cores contending for a particular memory controller, more so than the cache or the bus.

This effect is exacerbated when the live scheduler moves threads between cores during its placement phase. An application that began executing on core 0 would have local memory allocated to it on that core. After a subsequent move to core 1, the application would also have memory allocated to it on core 1. By accessing remote memory and local memory in a non-deterministic fashion, the application's performance would be unpredictable. Devils would be especially vulnerable to this phenomenon, being memory-intensive and cache-greedy.

With this caveat in mind, this work has not examined the NUMA results in as great a detail as the UMA results. The NUMA results are described in terms of co-scheduling four applications together in the dual-case workload, and eight applications together in the quad-case workload.

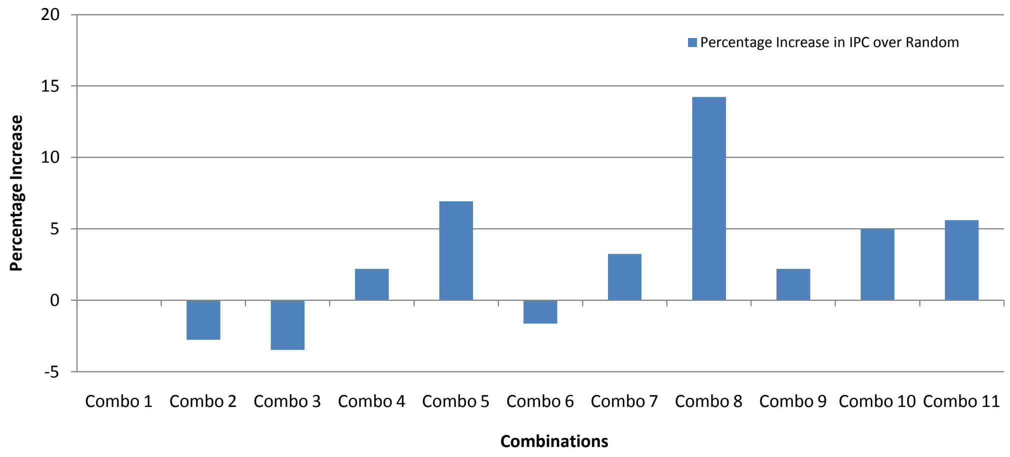


Figure 5.14: Percentage Increase in IPC: Combination

5.2.1 Dual-case Workload

Recall that in Section 4.7, we chose a subset of 11 workloads, sorted by their maximum possible gain. This gain was based on the model outlined in Section 4.4, where the sum of the IPC degradations between two co-scheduled applications became the co-scheduled degradation. However, this model was based on measuring a single pair of co-scheduled applications. In the dual-case, when we co-schedule two pairs of applications, for a total of four applications running concurrently, the model falls a little short since it doesn't take into account the system activity of all four applications. Since the model gave us a good starting point for the eleven quartets of applications with the maximum possible gain, I used those eleven combinations to test my scheduler.

5.2.1.1 Combinations

We examine the percentage increase in IPC of combinations when run with the live scheduler against the random scheduler, on the NUMA machine. Figure 5.14 shows that all but three combinations have an increase in IPC with the live scheduler. A maximum increase of 14.24% is seen in Combination 8. Some combinations experience a minor decrease in IPC, resulting in an overall average IPC percentage increase of 2.87% across all eleven quartets. These performance gains are not as reliable as the gains in the UMA section above, because of the varying degree of memory allocation to local and remote memory on a processor node.

5.2.1.2 Individual Applications

We next examine the increase in an application's IPC, averaged over the combinations it was co-scheduled in. Figure 5.15 shows us the gains in an application's IPC, colour-coded by the class it belongs too. The blue bars are devils, the red bars are turtles and the green bars denote others. We see that the turtles benefited consistently from the live scheduler, while the devils experienced decreases in IPC, albeit to a varying degree. In the others category, *gobmk* had a healthy increase, while *gcc* experienced an IPC decrease, despite belonging to the same category as *gobmk*.

With the exception of *mcf*, these results are the opposite of the results in the UMA dual-case workload. *gcc* suffers a decrease in IPC and gobmk experiences an increase, whereas their roles were reversed in the UMA case. All the devils except for *mcf* experience IPC increases with the UMA live scheduler, while all the devils suffer an IPC decrease on this NUMA system. Finally, the turtles too are not immune. All the turtles experience an IPC decrease with the UMA scheduler, while they experience increases with the live scheduler on this NUMA platform.

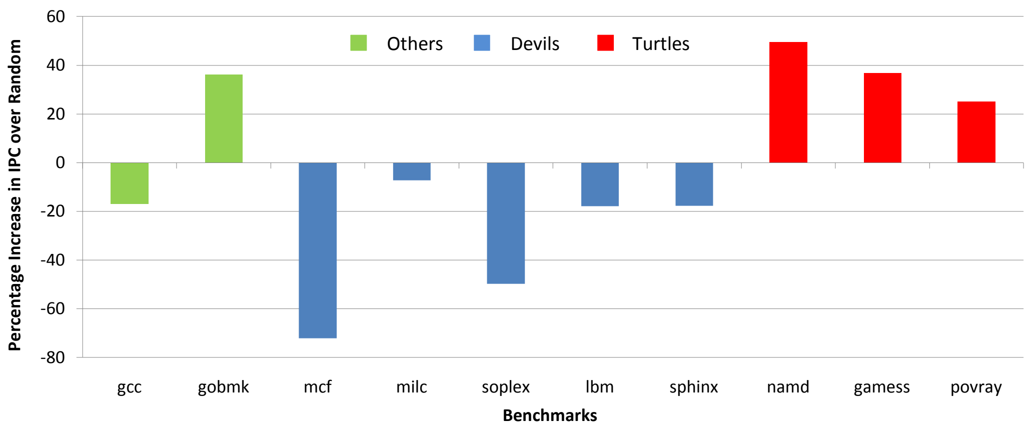


Figure 5.15: Percentage Increase in Individual Averages

A more succinct representation of the averages per category is seen in Figure 5.16. Here we see that applications in the turtle and "others" category benefited from the live scheduler, while devils experienced IPC decreases. The probable cause for the decrease in devils is contention for the NUMA memory controllers on each core. This remains the most likely hypothesis, although I have not investigated it further in this thesis and earmarked it for future work.

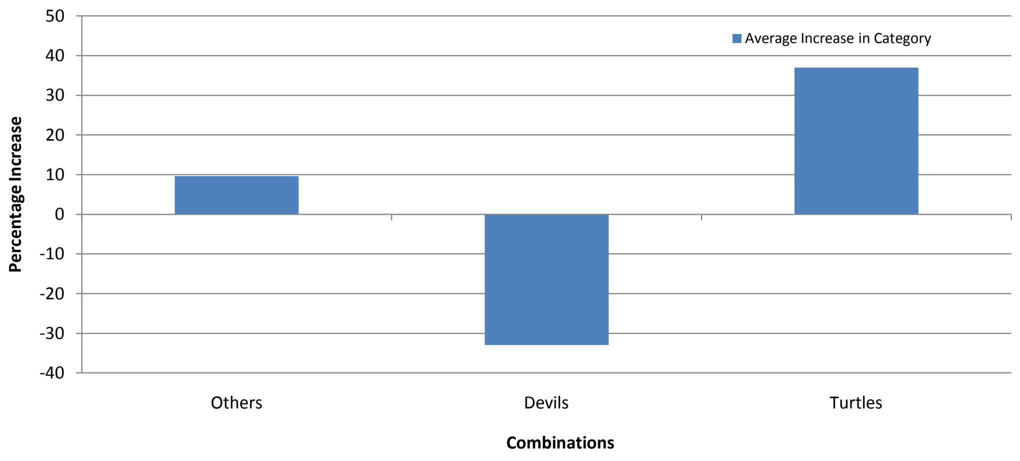


Figure 5.16: Percentage Increase in Animalistic Categories

Although these results are greater in magnitude than the corresponding dual-case UMA results, they are not what the live scheduler intended and as such, suffer from the artifacts of NUMA memory node interleaving. I have included the NUMA results for completeness, but the live scheduler is meant to effectively manage co-schedules based on cache contention, not memory-controller contention. By moving devils away from their local memory and forcing them to contend for both local and remote memory, the live scheduler introduces side-effects into the system that it is not prepared to deal with.

5.2.2 Quad-case Workload

The quad case problem was a bit more challenging, since measuring the best and worst performance of each octet was not feasible. Out of the 210 possible quartets from the experimental subset of ten, scheduling eight at a time such that the two sets of four applications have nothing in common leaves us with 3150 combinations, as calculated below.

The number of ways to schedule 8 applications out of 10 in two unique sets of four
$= \binom{10}{4} * \binom{6}{4}$
$= 3150$

Since it was not feasible to run all 3150 combinations and measure the best and worst performance, and thus obtain the maximum possible gain, I chose 10 combinations out of the 3150 at random. These 10 combinations are listed in Table 5.5 and were run with the random scheduler and my live scheduler. The random case in this experiment was set up in a similar fashion to the dual case workload. Eight applications were started and pinned to all eight cores at random. The applications were not migrated between runs and were restarted until the longest-running application finished executing. When all the applications had been restarted at least once, all executing threads were terminated and the results were written to a log file.

#	Combination
1	gcc gamess mcf milc gobmk soplex povray lbm
2	sphinx gcc gamess mcf milc gobmk povray lbm
3	gcc gamess mcf gobmk soplex povray lbm sphinx
4	gcc gamess milc namd gobmk soplex povray sphinx
5	gcc gamess milc gobmk soplex povray lbm sphinx
6	gcc mcf milc namd gobmk soplex lbm sphinx
7	gcc mcf namd gobmk soplex povray lbm sphinx
8	gcc milc namd gobmk soplex povray lbm sphinx
9	gamess mcf milc namd gobmk povray lbm sphinx
10	gamess mcf milc gobmk soplex povray lbm sphinx

Table 5.5: Quad-case Experimental Subset Combinations

5.2.2.1 Combinations

Unlike the dual-case workload where it was easy to obtain the maximum possible gain, the quad-case workload did not lend itself to an easy measurement of the best and worst performance. With eight applications in an octet, there are $\binom{8}{4} = 70$ ways to schedule each octet. Running all seventy possible combinations and finding the best and worst performers was not feasible, which is why I chose to compare the performance of the live scheduler directly against the average. Lacking the maximum possible gain data prevents us from seeing how much of the possible gain the live scheduler extracted from each scheduled octet. However, the average data gives us a better overall picture of the performance of the live scheduler compared to the random, naïve scheduler.

Figure 5.17 shows the average IPC of the octets when run with the random scheduler versus the live scheduler. In this case, we see that six out of ten of the octets experience an

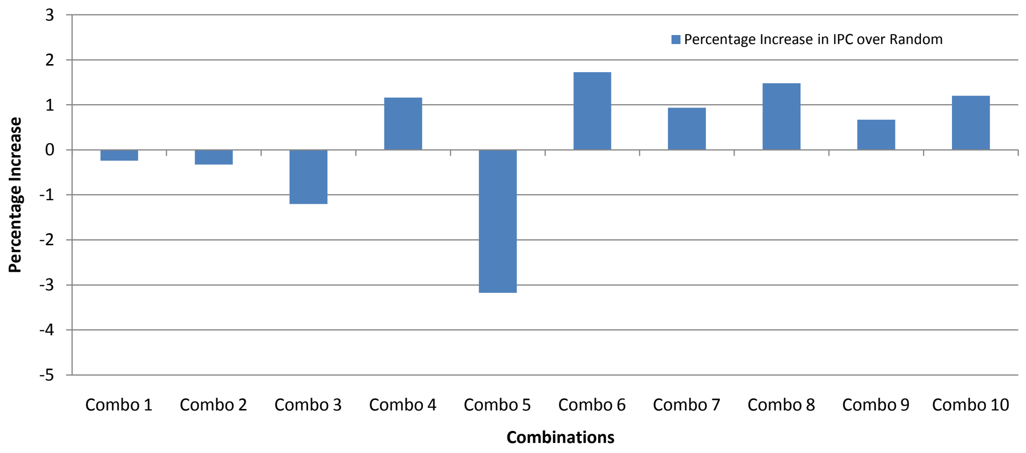


Figure 5.17: Percentage Increase in IPC: Combination

increase in IPC, leading to an overall average increase of 0.22% over the random scheduler. This very slight improvement is lesser than the gain we observed in the dual-case.

5.2.2.2 Individual Applications

In this section, we examine the effects of the live scheduler on the individual performance of the applications in the ten octets. We start by looking at the IPC increase in applications, averaged over all runs in all ten octets. Similar to the dual-case, the blue bars denote devils, the red bars denote turtles and the green bars signify others. The data in Figure 5.18 shows that no category of applications is consistently aided or inhibited by the live scheduler. In the devils, *mcf* and *sphinx* have positive net results in IPC, while *soplex* and *lbm* languish in the decreased IPC zone. The IPC gains in the turtles hover around the zero mark, with *povray* being slightly under zero, and *namd* and *gamess* rising above zero. The others category shows increases in *gobmk*, but decreases in *gcc*. This merely confirms that these results are heterogeneous, with increases and decreases being inconsistent and appearing in each category.

Averaging the results based on the animalistic classifications gives us Figure 5.19. Unlike the dual-case, the devils experience an increase in IPC here, along with the turtles. The others category ends up with a decrease in IPC compared to the random scheduler. These inconsistencies are precisely what mark the experiments on the NUMA system as being unpredictable, and not indicative of the real gains possible from the live scheduler.

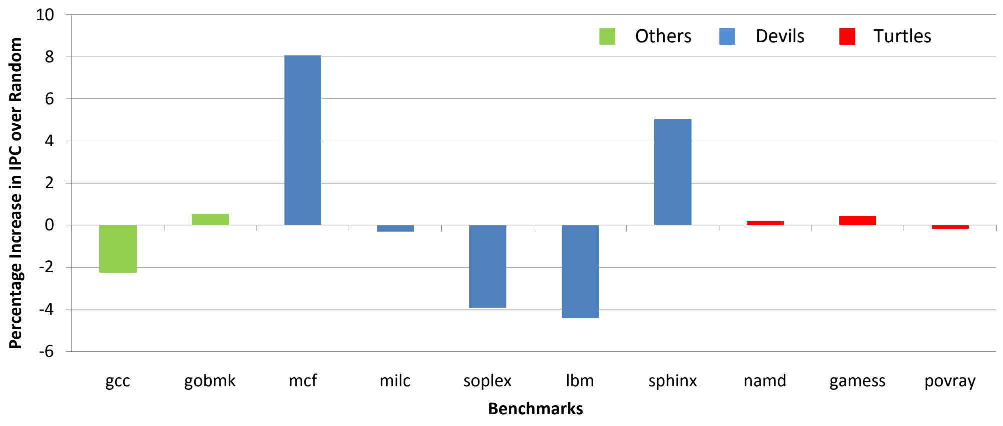


Figure 5.18: Percentage Increase in Individual Averages

Thus, on the whole we see that, on the NUMA system, factors other than cache contention play a part in the performance of applications. Contention for the memory controller arguably outweighs cache contention, making the NUMA system infertile ground for reaping the benefits of the live scheduler.

5.3 Summary

On the whole, we see that the presented approach of greedily scheduling applications such that the most intensive are paired with the least sensitive benefits certain workloads on UMA systems in a marked manner. Workloads with equal numbers of devils and non-devils show the most gain, achieving 100% of the maximal possible gain in some combinations. Workloads with less non-devils than devils also show gains of up to 79%. It is only in workloads with multiple intensive devils that the live scheduler delivers suboptimal results. Overall, devils and other applications benefit the most from the live scheduler, while turtles see a slight performance decrease, up to 1.26%. The live scheduler is more consistent than the random scheduler, delivering a higher IPC with a lesser standard deviation.

On NUMA systems, the live scheduler is rendered ineffective by the contention for memory controllers, which outweighs any benefits cache-contention-aware scheduling might provide. A better heuristic on NUMA systems that takes into account contention for memory controllers could be used in conjunction with the approach to produce higher performance

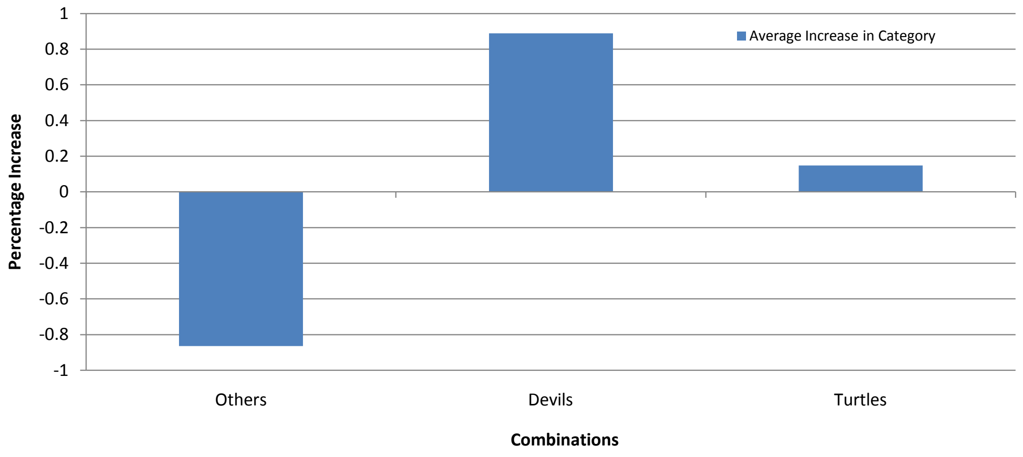


Figure 5.19: Percentage Increase in Animalistic Categories

gains. Such a heuristic would be an area of future work in this field.

Thus, we can conclude that when there are performance gains extractable from co-schedules of applications, the live scheduler will do a competent job of extracting close to the maximal gain. The only exception is when the optimal schedule consists of the two most intensive applications co-scheduled together. A finer-grained heuristic that goes beyond the simple dichotomy of devils and non-devils might be able to do a better job of picking the optimal schedule in these cases. This is intended as future work.

Chapter 6

Conclusion

Optimally scheduling applications on multicore systems to extract maximum performance while reducing contention for hardware resources remains an interesting and open problem today. The main contribution of this thesis is a user-level scheduler that demonstrates the role that cache attributes can play in making scheduling decisions. The default scheduler in an operating system on a multicore system is usually unaware of the cache characteristics of an application and co-schedules applications at random. By taking into account certain cache characteristics of an application, such as its L2 cache miss rate and access rate, the presented scheduler effects an increase in IPC over the Solaris random scheduler on an UMA system.

Workloads with equal numbers of intensive and non-intensive applications benefit the most, extracting up to a 100% of maximum possible gain in some cases, with the live scheduler. By keeping sensitive and intensive applications apart, the live scheduler is able to extract maximal performance gains in workloads of the former type. In other workloads where the intensive applications are outnumbered by non-intensive applications, the live scheduler achieves up to a 79% performance gain over the random scheduler. This comes at the cost of decreasing the IPC of the non-intensive applications by an average of 1.92%. Better, more granular heuristics that are able to pinpoint exceptions when intensive applications can be co-scheduled might deliver better performance gains, when used in conjunction with the live scheduler.

Certain other factors such as memory contention and prefetching place an upper bound on the amount of IPC increase that can be extracted with my method. These factors can form the basis of future work that might deliver more substantial gains in performance, when used in conjunction with the presented heuristic.

Bibliography

[1] Sergey Blagodurov, Sergey Zhuravlev, Serge Lansiquot, and Alexandra Fedorova. Addressing Cache Contention in Multicore Processors Via Scheduling. Technical Report TR 2009-16, July 2009.

[2] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[3] Jack Edmonds. Maximum Matching and a Polyhedron with $0, 1$ Vertices. *J. of Res. the Nat. Bureau of Standards*, 69 B:125–130, 1965.

[4] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–22, New York, NY, USA, 2006. ACM.

[5] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and Approximation of Optimal Co-scheduling on Chip Multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229, New York, NY, USA, 2008. ACM.

[6] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, 2008.

[7] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. Modeling Cache Sharing on Chip Multiprocessor Architectures. *IEEE Workload Characterization Symposium*, 0:160–171, 2006.

[8] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance,Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[9] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244. ASPLOS, 2000.

[10] Livio Soares, David Tam, and Michael Stumm. Reducing the Harmful Effects of Last-level Cache Polluters with an OS-level, Software-only Pollute Buffer. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, Washington, DC, USA, 2008. IEEE Computer Society.

[11] William Stallings. *Operating Systems (5th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[12] Soares L. Tam D., Azimi R. and M. Stumm. Managing Shared L2 Caches on Multicore Systems in Software. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33. WIOSCA, 2007.

[13] Yuejian Xie and Gabriel H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*. (CMP-MSI, held in conjunction with ISCA-35), 2008.

[14] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards Practical Page Coloring-based Multicore Cache Management. In *EuroSys '09: Proceedings of the Fourth ACM European Conference on Computer Systems*, pages 89–102, New York, NY, USA, 2009. ACM.