DESIGNING SCHEDULING ALGORITHMS FOR MITIGATING SHARED RESOURCE CONTENTION IN CHIP MULTICORE Processors

by

Sergey Zhuravlev
B.Sc., Simon Fraser University, 2009
B.Sc., University of British Columbia, 2006

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

© Sergey Zhuravlev 2011
SIMON FRASER UNIVERSITY
Spring 2011
APPROVAL

Name: Sergey Zhuravlev

Degree: Master of Science

Title of Thesis: Designing Scheduling Algorithms for Mitigating Shared Resource Contention in Chip Multicore Processors

Examining Committee: Dr. Funda Ergun
Chair

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

Dr. Lou Hafer,
Professor, Computing Science
Simon Fraser University
Supervisor

Dr. Richard Vaughan,
Professor, Computing Science
Simon Fraser University
SFU Examiner

Date Approved: 27 April 2011
Declaration of Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the “Institutional Repository” link of the SFU Library website <www.lib.sfu.ca> at: <http://ir.lib.sfu.ca/handle/1892/112>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada
Abstract

Chip multicore processors (CMPs) have become the default architecture for modern desktops and servers alike. CMPs consist of several processing cores on a single die making them ideal for multiprogramming and parallel workloads. The CMP cores, however, are not fully independent processors but rather share certain resources amongst each other. These shared resources can become points of contention for threads on neighboring cores leading to significant and often unpredictable performance impact. We propose a contention-mitigation solution via thread-level scheduling. Based on extensive studies into both the causes of contention as well as techniques to predict contention in CMPs we create an effective contention-mitigation scheduler. We highlight the vast design-space of scheduling algorithms and present a novel tool-set called AKULA for rapidly exploring this search space. We demonstrate the advantage of the AKULA tool-set by using it to prototype several new scheduling algorithms and evaluate their utility.
To my grandmother Basia Zhuravlev.
“Oh, so they have the internet on computers now!”

— Homer Simpson, The Simpsons
Acknowledgments

Words cannot express how grateful I am to my senior supervisor Dr. Alexandra Fedorova (Sasha). Without her none of this would have been possible. She has spent countless hours, guiding, teaching, editing, and helping me navigate through the twists and turns of graduate school. Dr. Fedorova taught me small specifics about computing systems as well as the big picture. She taught me how to truly read and dissect scientific papers. She also taught me how to write them. Most importantly she taught me how to be an independent researcher. Thank you Sasha!

I would also like to thank my friend and colleague Sergey Blagodurov. We spent months in the lab discussing, developing, running experiments and laying the groundwork for what would ultimately become this thesis.
# Contents

<table>
<thead>
<tr>
<th>Approval</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Quotation</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
</tbody>
</table>

## 1 Introduction and Background

1.1 Points of Contention in CMPs ............................ 4
1.2 Contention Mitigation Solutions .......................... 7
1.3 Outline of this work ........................................ 9

## 2 Designing a Contention-Aware Scheduler

2.1 Assumptions Used by Contention-Aware Schedulers ............... 13
2.2 Causes of Contention ............................................ 15
   2.2.1 Estimating Performance Degradation due to DRAM Controller Contention ........................................... 17
   2.2.2 Estimating Performance Degradation due to FSB Contention .............................................. 19
4.2.2 Well Behaved Workloads .......................... 70
4.2.3 Badly Behaved Workloads .......................... 74
4.2.4 Scalability of Algorithms .......................... 75
4.2.5 Validating the Bootstrapping Module and Data .......................... 78
4.3 Other Performance Emulation Tools .......................... 80
4.4 Summary ............................................. 81

5 Summary of Contributions .......................... 83

Bibliography ............................................. 85
List of Tables

2.1 **Pairwise degradation matrix**: The percent slowdown relative to running solo on the machine that an application (row) experiences when sharing the memory domain with another application (column). ........................................ 23

3.1 **DI test workloads**: The eight thread combinations (workloads) that were used to evaluate the DI scheduler. The number of memory- and CPU-bound applications in each workload is shown. Memory bound applications are shown italicized. The solo execution times of the applications vary between one and fifteen minutes. ......................................................... 44

4.1 **Bootstrap solos**: Solo execution times of each application. ...................... 65

4.2 **Bootstrap degradation matrix**: A degradation matrix for four applications. Each cell shows the degradation from the solo execution time when the two applications in the corresponding row and column are co-scheduled on the same memory domain. ....................................................... 65

4.3 **“Well Behaved” Workloads** ............................................................. 72

4.4 **“Badly Behaved” Workloads** .......................................................... 74

4.5 **Workloads for the 128 core machine** .............................................. 78
List of Figures

1.1 **Shared Resource Slowdowns**: The slowdowns in execution times experienced by benchmarks when sharing the LLC as compared to running alone on the machine. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. ............................................. 2

1.2 **Solo Measurement**: One target application bound to a single core of our 4-core, 2 memory domain machine. This set up is used to obtain “solo execution times”. .................................................. 3

1.3 **Contention Measurement**: One target application bound to a single core of our 4-core, 2 memory domain machine and one interfering application bound to another core in the same memory domain. This set up is used to obtain “execution times under contention”. ............................................................... 4

1.4 **Different Mappings**: Two different thread-to-core mappings for the same four applications (SOPLEX, SPHINX, GAMESS, and NAMD) on the same 4 core, 2 memory-domain machine. The mapping on the left results in significantly more performance degradation than the mapping on the right. . . . 9

1.5 **Different Mappings Results**: The per-thread as well as the average performance degradation (as compared to solo) for four applications on a 4 core, 2 memory domain machine in the globally worst and best mappings. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. Some applications experienced a small performance improvement (a few percent) due to shared cache effects. It is believed that this improvement results from the applications using shared libraries. .................................................. 10
2.1 **Single Memory Domain:** Two instances of thread-to-core mappings on a 4 core, 1 memory domain machine. The performance of the two instances is the same. ......................................................... 14

2.2 **Multiple Memory Domains:** Two instances of thread-to-core mappings on a 4 core, 2 memory domain machine. The performance of the two instances is different. ......................................................... 14

2.3 **Xeon Architecture:** Our evaluation platform. It consists of 8 cores where every pair of cores shares the LLC and prefetching logic. Each pair of LLCs is organized into sockets which share the the FSB controller. The two sockets share the DRAM controller. ......................................................... 15

2.4 **Solo on Xeon:** An application runs alone on our Xeon evaluation platform. The black line shows the contention-free portion of the memory hierarchy. .......................... 16

2.5 **Same Cache on Xeon:** Two application runs on our Xeon evaluation platform such that they share the LLC. The black line shows the contention-free portion of the memory hierarchy and the thick red line highlights the components under contention. ......................................................... 17

2.6 **Same Socket on Xeon:** Two application runs on our Xeon evaluation platform such that they share the same socket but do not share the LLC. The black line shows the contention-free portion of the memory hierarchy and the thick red line highlights the components under contention. ......................... 18

2.7 **Different Sockets on Xeon:** Two application runs on our Xeon evaluation platform such that they are not in the same socket. The black line shows the contention-free portion of the memory hierarchy and the thick red line highlights the components under contention. ......................................................... 19

2.8 **Contention Factor Breakdown:** An estimation of the percent contribution that the different contention factors (LLC-contention, FSB-contention, DRAM-controller-contention, and prefetching-resources-contention) have on the overall slowdowns that applications experience while sharing resources on the CMP. .......................................................... 22

2.9 **Graph Theoretical Representation:** The scheduling problem of four applications onto a 4 core, 2 memory domain machine as a complete graph. ........................ 23
2.10 **Best Solution:** The min-weight perfect matching solution of the graph theoretical representation of the scheduling problem is the optimal solution. The highlighted edges indicate that these applications are scheduled to the same memory domain. .......................................................... 25

2.11 **Middle Solution:** A perfect matching of the graph theoretical representation of the scheduling problem is a valid scheduling solution. The highlighted edges indicate that these applications are scheduled to the same memory domain. .......................................................... 26

2.12 **Worst Solution:** The max-weight perfect matching solution of the graph theoretical representation of the scheduling problem is the worst solution. The highlighted edges indicate that these applications are scheduled to the same memory domain. .......................................................... 27

2.13 **Creating stack distance profiles:** Shows a 4-way set associative cache and a corresponding SDP histogram. The entries at the top of the cache are the MRU position and those at the bottom are the LRU. Several address reads and the corresponding change in cache state as well as SDP are shown. .... 29

2.14 **SDP of GAMESS:** Shows excellent data locality (memory-reuse). .... 30

2.15 **SDP of MILC:** Shows very poor (non-existent data locality of reference). ... 31

2.16 **Evaluation of Classification Schemes** ............................................. 35

2.17 **DI on a 2 core per memory-domain machine:** DI first measures the miss rate of applications. It then sort them based on the miss rate. It pairs applications from opposite ends of the array and schedules them to the same memory domain. .......................................................... 39

2.18 **DI on a 4 core per memory-domain machine:** Starting at the point where the 2-core implementation left off, DI aggregates the miss rates of the application pairs. It then sort them based on the aggregate miss rate. It combines application pairs from opposite ends of the array and schedules them to the same memory domain. .......................................................... 40
3.1 **DI results on Xeon quad-core:** The percent degradation of DI above optimal as compared to the random and the worst schedules. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good). .................................................. 45

3.2 **DI results on Xeon eight-core:** The percent aggregate per workload speedup that DI delivers over the default Linux scheduler. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (High bars are good). 46

3.3 **DI worst-case speedup Xeon eight-core:** The percent worst-case per thread speedup that DI delivers over the default Linux scheduler. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (High bars are good). .................................................. 47

3.4 **DI variability Xeon eight-core:** The variability from run to run of every thread under DI and under the default Linux scheduler. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good). 48

3.5 **DI results on AMD:** The percent aggregate per workload speedup that DI delivers over the default Linux scheduler. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (High bars are good). .................. 49

3.6 **DI worst-case speedup AMD:** The percent worst-case per thread speedup that DI delivers over the default Linux scheduler. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (High bars are good). .................. 50

3.7 **DI variability AMD:** The variability from run to run of every thread under DI and under the default Linux scheduler. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good). .................. 51
4.1 **AKULA flow chart:** The intended use of the AKULA toolset to develop scheduling algorithms. A scheduling algorithm is created with the API, quickly tested with the Bootstrap Module, if successful tested with the Wrapper Module, if successful then it is safe to implement it inside the kernel.

4.2 **AKULA API example:** An implementation of Naive_Cluster using the AKULA API. The two functions `launch_thread()` and `update_schedule()` are required by all schedulers using AKULA.

4.3 **Four Core Architecture:** An example of a machine with 4 cores and 2 cores per memory domain.

4.4 **Bootstrap Example:** Shows the progress of each thread every clock tick as calculated by the Bootstrap Emulation Module.

4.5 **AMD Architecture:** The machine on which the bootstrap data was gathered and which was emulated for all experiments. It consists of 8 cores with 4 cores per memory domain. Each memory domain also features its own NUMA memory bank offering excellent inter-memory domain isolation.

4.6 **Average Performance Degradation “Well Behaved” Workloads:** Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).

4.7 **Worst Case Performance Degradation “Well Behaved” Workloads:** Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).

4.8 **Average Performance Degradation “Badly Behaved” Workloads:** Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).
4.9 **Worst Case Performance Degradation “Badly Behaved” Workloads:** Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).

4.10 **Scalability machine architecture:** The machine which was emulated to test the scalability of algorithms. It consists of 32 memory banks with 4 cores each and a per domain NUMA memory bank.

4.11 **Worst Case Performance Degradation on 128-core machine:** Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).

4.12 **Swap-X Performance:** Worst case performance degradation on 128-core machine for swap-X. Unlike previous graphs in this section the data here is NOT normalized to the best performing run. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).
Chapter 1

Introduction and Background

Moore’s law, coined by Intel co-founder Gordon Moore, predicts that the number of transistors that can be inexpensively placed on an integrated circuit doubles approximately every two years. Moore’s law has held since the early 1970s and is likely to continue to be true into the near future. For a long time computer architects have taken advantage of the growing transistor density to build progressively more complex single processor chips in order to exploit instruction level parallelism within applications. By introducing architectural features such as out of order execution, branch prediction, and wide issue instruction fetching they were able to help hide memory latency, process instructions more efficiently and deliver ever faster single-core processors. Contributing to the popularity of single-core processors was the fact that clock frequency was increasing almost in sync with the transistor density.

In recent years, however, the two major factors responsible for the growth of single-core technologies have reached their limits. Instruction level parallelism has become exhausted with ever larger and more complex architectures delivering ever more diminished speedup returns. Similarly the power budget needed to support the continued exponential growth of clock frequency and the corresponding increase in voltage needed to support such a switching frequency has become unrealistic. Furthermore, dissipating the heat associated with such clock-frequencies is well beyond the capabilities of modern thermal packaging. By some estimates if clock-frequencies continued to grow at the rate they have in the past the predicted temperature of the processor would reach that of a nuclear reactor!

Having hit the proverbial wall in terms of building more complex and faster single-core processors, architects have turned towards Chip Multicore Processors (CMPs) as a means of utilizing the still growing on-chip transistor count predicted by Moore’s law. CMPs are
characterized by several processing cores on a single die. In order to make more efficient use of transistor real estate the cores found on CMPs are not completely independent processors but rather share certain on- and off-chip resources. The most common shared resources in today’s CMPs are the last level cache (L2 or L3), the memory bus, DRAM controllers and pre-fetchers. These shared resources are managed exclusively in hardware and are thread-unaware; they treat requests from different threads running on different cores as if they were all requests from one single source. Thread-agnostic shared resource management can lead to poor system throughput as well as highly variable and workload dependent performance for threads running on the CMP.

Figure 1.1: Shared Resource Slowdowns: The slowdowns in execution times experienced by benchmarks when sharing the LLC as compared to running alone on the machine. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median.

To demonstrate the effect that shared resource contention can have on thread performance in a CMP we show the results of a simple experiment in Figure 1.1. Ten different benchmarks from the SPEC CPU2006 benchmark suite were executed on a Dell-Powerededge-2950 Intel Xeon X5365 machine. This machine has four cores, where each pair of cores shares an L2 cache. In the first set of runs each benchmark was executed alone on the machine while being bound to the first physical core Figure 1.2. In the next set of experiments each benchmark was also bound to the first core but another application “the interference application” was also executed on the machine and bound to the physical core
which shares the L2 cache with the first core to which our target application was bound
Figure 1.3. Figure 1.1 shows the percent slow down in execution time that each benchmark
experienced when running with the interfering application as compared to running alone
on the machine. We see that some benchmarks experience slowdowns in excess of 60%!
Considering that at all times the applications had exclusive control of a physical CPU such
slowdowns are impressive and highlight that shared resource contention is a serious issue
for CMPs.

Figure 1.2: **Solo Measurement**: One target application bound to a single core of our
4-core, 2 memory domain machine. This set up is used to obtain “solo execution times”.

Not surprisingly, there has been significant interest in the research community in ad-
ressing shared resource contention on CMPs. The majority of work required modifications
to hardware and falls into one of two camps: performance aware cache modification (most
commonly cache-partitioning) [85, 72, 27, 16, 47, 96, 40, 86, 37, 29, 74, 52, 105, 81, 33,
88, 76, 56, 44, 55, 19, 50, 5, 48, 87, 18, 57, 54, 101, 15, 84, 1, 7, 53, 104, 73, 70] or perfor-
mance aware DRAM controller scheduling [58, 89, 34, 75, 69, 77, 49, 12, 68, 21, 35, 67, 97].
The proposed solutions require changes to the hardware, major changes to the operating
Figure 1.3: **Contention Measurement**: One target application bound to a single core of our 4-core, 2 memory domain machine and one interfering application bound to another core in the same memory domain. This set up is used to obtain “execution times under contention”.

In modern computing systems there exists a significant lag between the processor speed and the speed with which memory can be accessed from DRAM (main memory). To compensate for this time lag and not force the processor to stall on every memory request a memory hierarchy of caches is used. Caches are hardware storage units which are smaller than but also faster to access than main memory. They are intended to hold the working data for the application. Contemporary computing systems typically have two or three levels of caches.

1.1 Points of Contention in CMPs

In modern computing systems there exists a significant lag between the processor speed and the speed with which memory can be accessed from DRAM (main memory). To compensate for this time lag and not force the processor to stall on every memory request a memory hierarchy of caches is used. Caches are hardware storage units which are smaller than but also faster to access than main memory. They are intended to hold the working data for the application. Contemporary computing systems typically have two or three levels of caches.
The Last Level Cache (LLC) which is the biggest and slowest of this hierarchy is almost always shared among two or more cores in a CMP making it one of the clearest points of contention in the CMP. The effect of simultaneously executing threads competing for space in the LLC has been explored extensively by many different researchers [85, 72, 27, 16, 47, 96, 40, 86, 37, 29, 74, 52, 105, 81, 33, 88, 76, 56, 44, 55, 101, 15, 84, 1, 7, 53, 104, 73, 70]. The most common replacement policy used in caches is an approximation to the Least Recently Used (LRU) policy [40, 88, 44]. LRU, when used with a single application, is designed to take advantage of temporal locality by keeping in cache the most recently accessed data. However, when the LLC is shared by multiple threads the LRU policy has the effect of favoring the thread with most memory accesses. This more aggressive thread is hence able to occupy a greater portion of the LLC than other threads. However, the benefit, in terms of execution time, that a thread receives from having greater cache space depends on its memory access pattern and thus varies greatly from thread to thread. It can be the case that the thread that allocates the most cache space for itself is not the one that will benefit the most from this space, and by forcing other threads to have less space it can adversely affect their performance [40].

[44, 15] both demonstrate the dramatic effects of this phenomenon. They show that the cache miss rate of a thread can significantly vary depending on the co-runner (the thread that runs on the neighboring core and shares the LLC). The increase in the miss rate, caused by cache contention, leads to a corresponding decrease in performance which also varies greatly depending on which threads share the LLC. [44] shows up to a nine-fold increase in LLC misses for the SPEC benchmark GCC when it shares the LLC as compared to when it runs alone. The corresponding performance of GCC as measured by Instructions per Cycle (IPC) drops to less than 40% of its solo performance. [15] shows an almost four-fold increase in the LLC miss rate for the SPEC benchmark MCF when it shares the LLC as compared to running alone. The corresponding performance of MCF is around 30% of its solo execution.

Another crucial shared resource and major source of contention in the CMP is the DRAM controller. The DRAM controller services memory requests which missed in the LLC. The DRAM controllers in all major commercial architectures implement the first-ready-first-come-first-serve (FRFCFS) policy where earlier memory requests are favored over later requests. Additionally, DRAM controllers employ various other techniques to
optimize throughput, such as favoring requests to the same memory bank or to the row-buffer. Like LRU for caches, FRFCFS was developed for single-threaded access and shows good results in that environment. However, when multiple threads compete for the DRAM controller this policy can result in unpredictable and poor performance.

[69] shows that the memory latency, measured as the number of stall cycles per memory request that an application experiences can increase up to ten-fold when competing for the DRAM controller with another application as compared to running solo. They also demonstrate that the IPC of the application can fall to about 40% of its solo rate when sharing the DRAM controller. [67] shows the high variability in memory latency that arises for different applications when they share the DRAM controller. The authors demonstrate that for four applications running on a four-core CMP the memory stall time for an individual application can increase eight-fold. For eight applications running on an eight-core CMP the increase in memory stall time can be as high as twelve-fold.

A far less frequently discussed but also an important source of contention in CMPs is shared memory bus. On Uniform Memory Access (UMA) Architectures with multiple LLCs, the LLCs are connected via a shared bus to the DRAM controller. This memory bus is another point of contention for threads running simultaneously on the CMP. [46] used a simulator to evaluate the effect that the shared memory bus by itself can have on the performance of threads in a CMP. Their experiments demonstrate that the reduction in IPC (normalized to solo performance) when two applications compete for the shared memory bus varies dramatically depending on which applications are used and can lead to a performance degradation of as much as 60% compared to running solo.

These studies show that competition for shared resources can lead to severe performance degradation (up to 70% reduction in IPC) but they also allude to another major problem: the reduction in IPC is not uniform and depends on the threads which compete for these shared resources. Some thread combinations experience minimal slowdown when sharing resources, other combinations lead to severe slowdowns, and still other are somewhere in between. The fact that a thread’s performance depends on the other threads in the workload makes thread performance on CMPs unpredictable which can be an even bigger problem than reduced throughput [65, 28, 38, 70, 37]. Unpredictable/workload-dependent performance means that Quality of Service (QoS) guarantees cannot be provided to threads and hence Service Level Agreements (SLAs) are very difficult to enforce. Furthermore, the OS scheduler, which relies on priority enforcement via timeslice allocation, becomes significantly less effective as the
actual progress a thread makes during a given timeslice is unknown. Contention for shared resources can lead to priority inversion as low priority threads impede the progress of high priority threads. In certain cases it may also lead to thread starvation as certain threads fail to make adequate progress [67]. The current shared resource management techniques often lead to unfair distribution of resources among competing threads and hence some threads gain unfair advantages at the expense of other threads.

1.2 Contention Mitigation Solutions

The vast majority of work addressing shared cache contention proposes some form of cache partitioning to deal with this issue. The basic idea of cache partitioning is to have a mechanism in place that will explicitly distribute LLC cache space amongst competing threads based on some metric. The cache partitioning solutions vary in the metric used to determine the distribution of cache space between the threads as well as the mechanism used to enforce this allocation decision. On the simpler end of the metric spectrum are solutions which evenly partition the cache between the threads for maximum fairness. On the other end of that spectrum are solutions which use real time measurements which are then fed into complex prediction models to determine which cache allocation will optimize overall throughput. The techniques used to enforce the partitioning decisions also vary along a wide spectrum. On the software end of the spectrum Tam proposes a page coloring technique where the OS allocates “colored” pages to applications whenever new memory is requested. Pages of the same color share the property that their addresses will map to the same cache lines. Thus by controlling the number of different colors an application is given its cache “footprint” is also controlled. On the hardware end of the enforcement spectrum are solutions which propose entirely hardware based modifications for detecting the thread which issued a given memory request and diverting it to a specific region of cache.

Hardware based solutions to address unfair and sub-optimal access to the DRAM controller use a mechanism referred to as DRAM scheduling where the number and/or order of memory requests submitted to the DRAM controller is explicitly controlled. Much like the cache partitioning solutions DRAM scheduling solutions also vary in the metric used to determine the number of requests each thread will be allowed to submit to DRAM over given a period as well as the mechanism used to enforce these limits.

Orthogonal to cache partitioning or DRAM controller scheduling another research trend
is emerging to deal with CMP shared resource contention on the level of thread scheduling [109, 45, 6, 41, 94, 64]. In this context thread scheduling refers to mapping threads to the cores of the CMP. Different mappings result in different combinations of threads competing for shared resources. Some thread combinations compete less aggressively for shared resources than others. Contention mitigation via thread scheduling aims to find the thread mappings which lead to the best possible performance.

Figures 1.4 and 1.5 show an example of how and why a contention-aware scheduler works. Figure 1.4 shows two instances of the same four SPEC CPU2006 benchmarks running on the same Xeon quad-core machine. The only difference between the left and right instance is that the applications are mapped to different cores. Mapping is achieved via user level system calls available in Linux which restrict the application to only running on one specified physical core. The difference in the mappings of the two instances in Figure 1.4 is that different applications are paired on neighboring cores which share the Last Level Cache (as well as other resources such as the prefetching logic). The different resource demands of different applications make some combinations highly competitive for shared resources resulting in high performance impact while other combinations are able to co-exists much more peacefully. Figure 1.5 shows that the average performance degradation of all four benchmarks (as compared to running alone on the machine) under the mappings of shown in Figure 1.4. The left instance of Figure 1.4 has an average slowdown of almost 30% with SOPLEX and SPHINX each slowing down by more than 50%. Conversely, the average slowdown of the right mapping in Figure 1.4 is less than 10% with no single application slowing down more than 15%. The very sizeable performance improvement that can be achieved by changing application to processor bindings makes contention-aware scheduling an interesting alternative.

Several other studies [109, 45, 6, 62, 83, 64, 25, 41, 94] have shown significant improvement in throughput, fairness, and predictability of contention-aware schedulers over contention-unaware schedulers (such as those currently in use in all commercial Operating Systems). Moreover, the simplicity of implementation, requiring no changes to the hardware and few changes to the OS\(^1\), makes the solution very attractive for the near term.

\(^1\)Some contention-aware thread schedulers were implemented entirely at user level.
1.3 Outline of this work

The purpose of this work is twofold: first the creation of a contention-aware thread level scheduler based on a systematic and methodical study of both the causes of and techniques for detecting inter-thread contention on CMPs. Second, the creation of a toolset and emulation methodology that allows for rapid exploration of the vast design space of contention-aware schedulers to make the search space simultaneously more tractable as well as much wider by facilitating the extrapolation of scheduler performance to futuristic highly multicore architectures.

The majority of the work on mitigating shared resource contention focuses on one single point of contention such as the LLC or the DRAM controller. The solution is then explored on a simulator which has the effect of creating an almost bubble effect which isolates this one point of contention from the rest of the system. On a real system the various points of contention like the LLC, the shared memory bus, the DRAM controller as well as others interact together in complex and often difficult to tease apart ways. Mitigating contention for one shared resource may exacerbate another bottle neck making performance worse. As such, we begin our design of the contention-aware scheduler by first systematically exploring what are the points of contention in the CMP and which if any are more dominant and hence should be focused on.

The only tool that the contention-aware scheduler has at its disposal to improve performance is to change the thread-to-core-mapping to one which avoids making neighbors of
Figure 1.5: **Different Mappings Results:** The per-thread as well as the average performance degradation (as compared to solo) for four applications on a 4 core, 2 memory domain machine in the globally worst and best mappings. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. Some applications experienced a small performance improvement (a few percent) due to shared cache effects. It is believed that this improvement results from the applications using shared libraries.

threads that aggressively compete for resources in favor of those that are more copacetic. To find these more favorable mappings it is necessary to classify the threads in some way as to be able to distinguish among them and decide which should and should not be scheduled together. We take a methodical approach to explore and evaluate different thread classification schemes to determine the optimality of each. We combine the results of our extensive sources of contention as well as thread classification studies to design and implement a contention-aware scheduler called Distributed Intensity (DI). We evaluate DI on physical machines as well as compare it to other contention-aware scheduler from literature.

Although DI is shown to be highly effective at mitigating shared resource contention on modern CMPs the sheer vastness of the scheduler design space suggests a need for a comprehensive methodology by which it can be efficiently explored. Consider some of the axis along which the design space can be traversed. Starting with what exactly is the desired outcome of a “good” solution. [33] brings to the forefront the discrepancy between “communist” solutions (those where the goal is to even out performance degradation among all the threads)
and “utilitarian” solutions (those where the goal is to optimize overall throughput). They show that solutions that maximize fairness are not necessarily good for overall throughput and vice versa. However, they do show that in all cases both the “communist” and “utilitarian” solutions are better than the current free-for-all “capitalist” system. Another axis is the data the scheduler uses to make its decisions. Online performance counters are the most common source of such input data. However, even the raw number of online performance counters in modern processors is very large and this number grows exponentially as combinations and permutations of the counters are considered. Yet another axis is how the decisions are enforced. Is the scheduler centralized or distributed? How often should the scheduler run? How will the scheduler handle different architectures (more cores per chip, more chips per machine)? How will the scheduler handle the case when the number of threads exceeds the number of cores?

The list of options is enormous while the time and effort required to create and test a scheduler is non-trivial and hence a major limiting factor. Furthermore, one can only explore architectures which are currently available. The final major contribution of this work is a toolset and methodology for easily and rapidly developing and testing contention-aware schedulers. We present a novel API to make coding of schedulers very simple. We also developed a unique and innovative methodology by which the performance of the scheduler can be estimated within a fraction of the time experimentation on a real machine would take. Our methodology also allows for scalability testing of schedulers for futuristic massively multicore machines. We demonstrate the utility of our toolset and methodology by exploring improvements to DI which allow it to become resilient in the face of badly behaved workloads as well as highly scalable.
Chapter 2

Designing a Contention-Aware Scheduler

The contents of Chapter 2 and 3.1 are based on work which was done collaboratively between the author Sergey Zhuravlev and his colleague Sergey Blagodurov. This work was first published in [109]. The rest of the chapters are based on work done exclusively by the author Sergey Zhuravlev. Chapter 4 was first published in [110].

In Chapter 1 we discussed our own experiments as well as a multitude of other research showing that resource contention is a serious issue in CMPs. We also discussed that contention-aware schedulers can go a long way towards mitigating some of these negative effects and unlike more intrusive solutions require minimal changes to the existing systems. In this chapter we describe the methodology used to develop our contention-aware scheduler called DI. We begin by outlining the assumptions that we make about contention-aware schedulers in general. We then describe our search for the role that different contention factors play in the CMP in order to know for which to tailor our scheduler. At the heart of the scheduling algorithm is a classification policy which is needed to distinguish different threads in order to decide which should and should not be mapped together. We discuss our methodology for evaluating the optimality of the classification schemes (both our own and those from literature). Finally, we present our scheduling algorithm DI based on the conclusions drawn from our contention factors and classification scheme studies.
2.1 Assumptions Used by Contention-Aware Schedulers

**Assumption one:** a contention-aware scheduler space-shares the machine rather than time-shares it. Space sharing refers to deciding how the runnable threads will be distributed on the machine in a given time interval: which threads will be scheduled to “neighboring” cores and which will be scheduled to distant cores. Time sharing on the other hand refers to multiplexing time on a single CPU among multiple threads. Contention-aware schedulers typically do not interfere with the traditional OS time sharing scheduler. The schedulers that we will consider are of this sort. The complex interaction of simultaneously space and time sharing the machine is outside the scope of this work.

**Assumption two:** in order for contention-aware schedulers to be effective, the underlying CMP architecture must consist of multiple cores, where subsets of cores share different resources. We refer to a subset of cores which shares a set of resources as a memory-domain. For example a machine with 4 cores where each pair of cores shares the LLC will have two memory domains corresponding to each pair of cores that share the LLC. Figure 2.1 shows a CMP architecture where all four cores all share the same resources (are in a single memory domain). As such, the fact that the threads are mapped differently in the left instance than they are in the right instance will not make any difference on performance. In contrast, Figure 2.2 shows a CMP architecture where every pair of cores shares a different LLC. Thus the different thread-to-core mapping of the left and right instances will result in different performance. Given that contention-aware schedulers focus on space sharing, they will only be effective on architectures like those in Figure 2.2 where multiple memory domains are present, and *not* on architectures like in Figure 2.1.

**Assumption three:** all resource sharing is destructive interference. This weeds out the cases where two threads from the same application can share resources such as the LLC constructively, (i.e. in such a way as the performance of the two threads is higher when the resource is shared as compared to when it is not). Contention-aware schedulers view all sharing as purely destructive. The degree of destructive interference can vary greatly from negligible to significant but it will never be negative. This assumption holds for us because all experimented are performed with workloads consisting of only single-threaded applications which cannot share data or interfere constructively\(^1\). Even if multithreaded applications

\(^1\)It has been shown that under certain circumstances even different applications may experience a small performance improvement (a few percent) due to shared cache effects. It is believed that this improvement
Figure 2.1: **Single Memory Domain**: Two instances of thread-to-core mappings on a 4 core, 1 memory domain machine. The performance of the two instances is the same.

Figure 2.2: **Multiple Memory Domains**: Two instances of thread-to-core mappings on a 4 core, 2 memory domain machine. The performance of the two instances is different.

were to be used there is significant evidence that the benefits of constructive sharing are overwhelmed by the penalties from destructive interference. [91] showed that in the best case scenario constructive sharing can improve performance by a maximum of 7%. [102] did a comprehensive and systematic study of the benefits of cache sharing for contemporary multithreaded applications and found that these benefits were altogether insignificant. The main reason behind the underwhelming impact of cache sharing, according to [102], is the mismatch between the way data sharing is performed by the threads and the underlying CMP architecture on which they run. Conversely, if we look at destructive interference, [6] results from the applications using shared libraries.
showed that shared resource contention can degrade performance by almost 200%.

2.2 Causes of Contention

In this section we attempt to quantify how much performance degradation can be attributed to each factor: cache contention, memory controller contention, memory bus contention, prefetching hardware contention.

Estimating the contribution that each factor has on the overall performance degradation is difficult, since all the degradation factors work in conjunction with each other in complicated and practically inseparable ways. Nevertheless, we desired a rough estimate of the degree to which each factor affects overall performance degradation to identify if any factor in particular should be the focus of our attention since mitigating it would yield the greatest improvements.

Figure 2.3: Xeon Architecture: Our evaluation platform. It consists of 8 cores where every pair of cores shares the LLC and prefetching logic. Each pair of LLCs is organized into sockets which share the the FSB controller. The two sockets share the DRAM controller. When two threads run on different sockets, they compete for the DRAM controller only.

We now describe the process we used to estimate the contributions of each factor to the overall degradation. Our experimental system is a two-socket server with two Intel X5365 “Clovertown” quad-core processors. The two sockets share the memory controller hub, which includes the DRAM controller. On each socket there are four cores sharing a front-side bus (FSB). There are two L2 caches on each socket, one per pair of cores. Each pair of cores also shares prefetching hardware. Figure 2.3 shows the architecture of our machine.
When they run on the same socket, but on different caches, they compete for the FSB, in addition to the DRAM controller. Finally, when they run on cores sharing the same cache, they also compete for the L2 cache and the prefetching hardware, in addition to the FSB and the DRAM controller. To estimate how contention for each of these resources contributes to the total degradation, we measured the execution times of several benchmarks under the following six conditions:

- **Solo PF ON**: Running SOLO and prefetching is ENABLED Figure 2.4.
- **Solo PF OFF**: Running SOLO and prefetching is DISABLED Figure 2.4.
- **SameCache PF ON**: Sharing the LLC with an interfering benchmark and prefetching is ENABLED Figure 2.5.
- **SameCache PF OFF**: Sharing the LLC with an interfering benchmark and prefetching is DISABLED Figure 2.5.
- **DiffCache PF OFF**: An interfering benchmark runs on a different LLC but on the same socket and prefetching is DISABLED Figure 2.6.
- **DiffSocket PF OFF**: An interfering benchmark runs on a different socket and prefetching is DISABLED Figure 2.7.

Figure 2.4: **Solo on Xeon**: An application runs alone on our Xeon evaluation platform. The black line shows the contention-free portion of the memory hierarchy.
Figure 2.5: Same Cache on Xeon: Two application runs on our Xeon evaluation platform such that they share the LLC. The black line shows the contention-free portion of the memory hierarchy and the thick red line highlights the components under contention.

Figures 2.4 through 2.7 show the placement of threads for each of the six conditions tested as well as the portion of the memory path which is contention-free (thin black line) and the path which is under contention (highlighted in red).

As an interfering benchmark for this experiment we used the SPEC CPU2006 benchmark MILC. MILC was chosen for several reasons. First, it has a very high solo miss rate which allows us to estimate one of the worst-case contention scenarios. Second MILC suffers a negligible increase in its own miss rate due to cache contention (we determined this via experiments and also by tracing MILC’s memory reuse patterns, which showed that MILC hardly ever reuses its cached data) and hence will not introduce extra misses of its own when co-run with other applications. We refer to MILC as the interfering benchmark and we refer to the tests application simply as the application.

2.2.1 Estimating Performance Degradation due to DRAM Controller Contention

We look at the difference between the solo run and the run when the interfering benchmark is on a different socket. (The difference between performance in situations shown in Figure 2.7 and Figure 2.4). When the interfering benchmark is on a different socket any performance degradation it causes can only be due to DRAM controller contention since no other resources are shared. Equation 2.1 shows how we estimate the performance degradation due
CHAPTER 2. DESIGNING A CONTENTION-AWARE SCHEDULER

Figure 2.6: **Same Socket on Xeon:** Two application runs on our Xeon evaluation platform such that they share the same socket but do not share the LLC. The black line shows the contention-free portion of the memory hierarchy and the thick red line highlights the components under contention.

\[
DRAM_{Contention} = \frac{DiffSocket_{PF.OFF} - Solo_{PF.OFF}}{Solo_{PF.OFF}} \quad (2.1)
\]

There are several complications with this approach, which make it a rough estimate as opposed to an accurate measure of DRAM controller contention. First, when the LLC is shared by two applications, extra evictions from cache cause the total number of misses to go up. These extra misses contribute to the DRAM controller contention. In our experimental technique the two applications are in different LLCs and hence there are no extra misses. As a result, we are underestimating the DRAM controller contention. Second, we chose to disable prefetching for this experiment. If we enabled prefetching and put two applications into different LLC then they would each have access to a complete set of prefetching hardware. This would have greatly increased the total number of requests issued to the memory system from the prefetching hardware as compared to the number of requests that can be issued from only one LLC. By disabling the prefetching we are once again underestimating the DRAM controller contention. As such the values that we measure should be considered a lower bound on DRAM controller contention.
CHAPTER 2. DESIGNING A CONTENTION-AWARE SCHEDULER

2.2.2 Estimating Performance Degradation due to FSB Contention

Next, we estimate the degree of performance degradation due to contention for the FSB. To that end, we run the application and the interfering benchmark on the same socket, but on different LLCs (Figure 2.6). This is done with prefetching disabled, so as not to increase the bus traffic. This scenario captures performance degradation due to both FSB and DRAM contention. To isolate the degradation due to FSB contention we subtract from the obtained value the degradation measured previously only for DRAM. Equation 2.2 shows how we estimate the degradation due to FSB contention.

\[ FSB_{\text{Contention}} = \frac{\text{DiffCache}_{PF\ OFF} - \text{DiffSocket}_{PF\ OFF}}{\text{Solo}_{PF\ OFF}} \]  

(2.2)

2.2.3 Estimating Performance Degradation due to Cache Contention

To estimate the performance degradation due to cache contention we first measure the execution time when an application is run with an interfering co-runner in the same LLC (Figure 2.5). This value, however, is the combination of DRAM, FSB, and LLC contention. To isolate the degradation due to contention for the LLC we subtract from the obtained value the execution time of the application and interference running on the same socket but different LLCs (Figure 2.6) which captures DRAM and FSB contention. This is done with
prefetching disabled so as not to increase bus traffic or contend for prefetching hardware. The difference in the execution times between the two runs can be attributed to the extra misses that resulted due to cache contention. Equation 2.3 demonstrates how we estimate performance degradation due to cache contention.

\[
LLC\_\text{Contention} = \frac{\text{SameCache}_{PF\_OFF} - \text{DiffCache}_{PF\_OFF}}{\text{Solo}_{PF\_OFF}}
\]  

(2.3)

2.2.4 Estimating Performance Degradation due to Contention for Resources Involved in Prefetching

Contention for resources involved in prefetching has received less attention in literature than contention for other resources. We were compelled to investigate this type of contention when we observed that some applications experienced a decreased prefetching rate (up to 30%) when sharing an LLC with a memory-intensive co-runner. Broadly speaking, prefetching resources include all the hardware that might contribute to the speed and quality of prefetching. For example, our experimental processor has two types of hardware that prefetches into the L2 cache. The first is the Data Prefetching Logic (DPL) which is activated when an application has two consecutive misses in the LLC and a stride pattern is detected. In this case, the rest of the addresses up to the page boundary are prefetched. This prefetcher is primarily used for bringing data arrays into cache. The second is the adjacent cache line prefetcher, also called the streaming prefetcher. It brings into cache an additional line along with every regular request. The L2 prefetching hardware is dynamically shared by the two cores using the LLC. The memory controller and the FSB are also involved in prefetching, since they determine how aggressively these requests can be issued to memory. It is difficult to tease apart the latencies attributable to contention for each resource, so our estimation of contention for prefetching resources includes contention for prefetching hardware as well as additional contention for these two other resources. This is an upper bound on the contention for the prefetching hardware itself. We can measure the performance degradation due to prefetching related resources as the difference between the total degradation and the degradation caused by cache contention, FSB, and DRAM controller contention. Equation 2.4 calculates the total degradation of an application when the LLC is shared by looking at the difference when the interfering benchmark shares the LLC Figure 2.5 and when the application runs alone Figure 2.6. Equation 2.5 shows the calculation of the prefetching degradation.
\[
\text{Total Degradation} = \frac{\text{Solo\_PF\_ON} - \text{Solo\_PF\_OFF}}{\text{Solo\_PF\_ON}}
\] (2.4)

\[
\text{Prefetch Contention} = \text{Total Degrad} - \text{DRAM Cont} - \text{FSB Cont} - \text{LLC Cont}
\] (2.5)

2.2.5 The Contention Breakdown

Finally, we calculate the degradation contribution of each factor as the ratio of its degradation compared to the total degradation. Figure 2.8 shows the percent contribution of each factor (DRAM controller contention, FSB contention, L2 cache contention, and prefetching resource contention) to the total degradation for six SPEC2006 benchmarks. The six applications shown in Figure 2.8 are the applications that experience a performance degradation of at least 45%.

We see from Figure 2.8 that the two most well researched contention factors, shared cache contention and DRAM controller contention, are NOT the dominant contention factors. In fact, there is no clear dominant contention factor for which we should exclusively tailor the contention-mitigation solution. All the contention factors explored (shared cache contention, memory bus contention, DRAM controller contention, and prefetching contention) all play a significant role in the overall shared resource contention seen in CMPs. In order for a solution to be successful, it must comprehensively address all these factors rather than focusing on only a subset.

2.3 Classification Schemes

A conventional approach to evaluate new scheduling algorithms is to compare the speedup they deliver relative to a default scheduler. This approach, however, has two potential flaws. First, the schedule chosen by the default scheduler varies greatly based on stochastic events, such as thread spawning order. Second, this approach does not necessarily provide the needed insight into the quality of the algorithms. A scheduling algorithm consists of two components: the information (classification scheme, in our case) used for scheduling decisions and the policy that makes the decisions based on this information. The most challenging part of a contention-aware scheduling algorithm is to select the right classification scheme, because the classification scheme enables the scheduler to predict the performance
Figure 2.8: **Contention Factor Breakdown:** An estimation of the percent contribution that the different contention factors (LLC-contention, FSB-contention, DRAM-controller-contention, and prefetching-resources-contention) have on the overall slowdowns that applications experience while sharing resources on the CMP.

effects of co-scheduling any group of threads in the same memory domain. Our goal was to evaluate the quality of classification schemes separately from any scheduling policies, and only then evaluate the algorithm as a whole. To evaluate classification schemes independently of scheduling policies, we have to use the classification schemes in conjunction with a “perfect” policy. In this way, we are confident that any differences in the performance between the different algorithms are due to the classification schemes, and not to the scheduling policy.

### 2.3.1 A “perfect” scheduling policy

As a perfect scheduling policy, we use an algorithm proposed by Jiang et al. [41]. This algorithm is guaranteed to find an optimal scheduling assignment, i.e., the mapping of threads to cores, on a machine with several memory domains as long as the co-run degradations for applications are known. A co-run degradation is an increase in the execution time of an application when it shares a memory domain with co-runner(s), relative to running solo.

Jiang’s methodology uses co-run degradations to construct a graph theoretic representation of the problem, where threads are represented as nodes connected by edges, and the weights of the edges are given by the sum of the mutual co-run degradations that result if
the threads are co-scheduled to the same memory domain. Figure 2.9 shows the graph theoretical representation of the scheduling problem involving four applications (MCF, MILC, GAMESS, and NAMD), a machine with 4 cores and 2 cores per memory domain, and the co-run degradations Table 2.1 obtained experimentally by running all required combinations.

![Graph Theoretical Representation](image)

**Figure 2.9: Graph Theoretical Representation:** The scheduling problem of four applications onto a 4 core, 2 memory domain machine as a complete graph.

**Table 2.1: Pairwise degradation matrix:** The percent slowdown relative to running solo on the machine that an application (row) experiences when sharing the memory domain with another application (column).

<table>
<thead>
<tr>
<th></th>
<th>mcf</th>
<th>milc</th>
<th>gameess</th>
<th>namd</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf</td>
<td>48.01%</td>
<td>65.63%</td>
<td>2.0%</td>
<td>2.11%</td>
</tr>
<tr>
<td>milc</td>
<td>24.75%</td>
<td>45.39%</td>
<td>1.23%</td>
<td>1.11%</td>
</tr>
<tr>
<td>gameess</td>
<td>2.67%</td>
<td>4.48%</td>
<td>-1.01%</td>
<td>-1.21%</td>
</tr>
<tr>
<td>namd</td>
<td>1.48%</td>
<td>3.45%</td>
<td>-1.19%</td>
<td>-0.93%</td>
</tr>
</tbody>
</table>

The optimal scheduling assignment can be found by solving the min-weight perfect matching problem over this graph theoretical representation. Although this methodology extends to arbitrarily numbers of cores per memory domain for simplicity we will discuss all examples in terms of the two cores per memory domain problem. Such as the one
The obtained graph-theoretical representation is a complete graph (one where all possible edges exist). It is a complete graph because conceptually every pair of threads can be scheduled together to the same memory domain. A matching of a graph is defined as a set of edges without common vertices. A matching for this problem means that thread pairings are selected in such a way as to never include the same thread in more than one pairing (which of course makes physical sense for the problem). A perfect-matching of a graph is a matching such that every vertex is incident on exactly one edge of the matching. In other words, such that every vertex (thread) appears in the matching. Thus, a perfect matching of the graph is equivalent to a scheduling solution where every thread has been paired up with one other thread. Figures 2.10, 2.12, and 2.11 show the three possible perfect-matchings for the graph of Figure 2.9. The weight of each solution is the sum of the weights of the edges used in that matching. This directly corresponds to the total performance degradation of that scheduling solution. Therefore, the perfect matching having the minimum weight (min-weight perfect matching) is the optimal solution while the max-weight perfect matching is the worst solution. Figure 2.10 is the min-weight perfect matching while Figure 2.12 is the max-weight perfect matching.

Although Jiang’s methodology and the corresponding algorithms would be too expensive to use online (the complexity of the algorithm is polynomial in the number of threads on systems with two cores per memory domain and the problem is NP-complete on systems where the degree of sharing is larger), it is acceptable for offline evaluation of the quality of classification schemes.

Using Jiang’s algorithm as the perfect policy implies that the classification schemes we are evaluating must be suitable for estimating co-run degradations. All of our chosen classification schemes answered this requirement: they can be used to estimate co-run degradations in absolute or in relative terms.

### 2.3.2 An Optimal Classification Scheme

To determine the quality of various classification schemes we not only need to compare them with each other, but also to evaluate how they measure up to the optimal classification scheme. All of our evaluated classification schemes attempt to approximate the relative performance degradation that arbitrary tuples of threads experience when sharing a memory domain relative to running solo. An optimal classification scheme would therefore have knowledge of the *actual* degradation, as measured on a real system. To obtain
Figure 2.10: **Best Solution:** The min-weight perfect matching solution of the graph theoretical representation of the scheduling problem is the optimal solution. The highlighted edges indicate that these applications are scheduled to the same memory domain.

these measured degradations, we selected ten representative benchmarks from the SPEC CPU2006 benchmark suite (the methodology for selection is described later in this section), ran all possible pairs of these applications and recorded their performance degradation relative to solo performance. In order to make the analysis tractable it was performed based on pairwise degradations, assuming that only two threads may share a memory domain.

### 2.3.3 Evaluating Classification Schemes

To evaluate a classification scheme on a particular set of applications, we follow these steps:

1. Find the *optimal schedule* using Jiang’s method and the optimal classification scheme, i.e., the measured degradations. Record the aggregate performance degradation of threads in this scenario.

2. Find the *estimated best schedule* using Jiang’s method and the evaluated classification scheme, i.e., the estimated degradations. Record the aggregate performance degradation of threads in this scenario.

3. Compute the difference between the degradation of the optimal schedule and of the
Figure 2.11: **Middle Solution**: A perfect matching of the graph theoretical representation of the scheduling problem is a valid scheduling solution. The highlighted edges indicate that these applications are scheduled to the same memory domain.

estimated best schedule. The smaller the difference, the better the evaluated classification scheme.

To perform a rigorous evaluation, we construct a large number of workloads consisting of four, eight and ten applications. We evaluate all classification schemes using this method, and for each classification scheme report the average difference from the optimal across all workloads.

### 2.3.4 Benchmarks and Workloads Used

We selected ten benchmarks from the SPEC CPU2006 benchmark suite to represent a wide range of memory access behaviors. The cache miss rates and access rates for every application in the SPEC CPU2006 benchmark suite were obtained from a third party characterization report [32] and a clustering technique was employed to select the ten representative applications.

From these ten applications we constructed workloads for a four-core, six-core, eight-core, and ten-core processor with 2-cores per memory domain. With the ten benchmarks we
selected, there are 210 unique four-application workloads, 210 unique six-application workloads, 45 unique eight-application workloads, and 1 unique ten-application workload to be constructed on each system (under the assumption that multiple copies of the same application are not used). There are three unique ways to schedule a four-application workload on a machine with four cores and two memory domains. The number of unique schedules grows to 15, 105, and 945 for the six, eight, and ten-core systems respectively. Using Jiang’s methodology, as opposed to running all these 9450 schedules, saves a considerable amount of time and actually makes it feasible to evaluate such a large number of workloads.

### 2.3.5 The Classification Schemes

For any classification scheme to work it must first obtain some “raw” data about the applications it will classify. This raw data may be obtained online via performance counters, embedded into an application’s binary as a signature, or furnished by the compiler. Where this data comes from has a lot to do with what kind of data is required by the classification scheme. The SDC algorithm proposed by Chandra et al. [15] is one of the most well known methods for determining how threads will interact with each other when sharing the same cache. The SDC algorithm requires the memory reuse patterns of applications, known as
stack distance profiles, as input. Likewise all but one of our classification schemes requires stack distance profiles. The one exception is the Miss Rate Classification Scheme which requires only miss rates as input. The simplicity of the Miss Rate Scheme whose input can be gathered dynamically online makes it a far more attractive option than the other classification schemes. However, in order to compare the effectiveness of the miss rate scheme against more complex and better established classification schemes we need to explore a wide variety of classification schemes and we need to use stack distance profiles to do so.

A stack distance profile is a compact summary of the application’s cache-line reuse patterns. It is obtained by monitoring (or simulating) cache accesses on a system with an N-way set associative LRU cache. Conceptually, an N-way set associative cache uses an indexing function to map all possible addresses into cache sets. Since there are significantly more addresses than sets multiple addresses will map to the same set. For each cache set there are N-ways (where N is an integer most commonly 2, 4, 8, or 16). Each way can hold one cache entry. As such an N-way set associative cache can hold N entries with addresses that map to the same set. When room is required to bring a new entry into an already full set the least recently used (LRU) policy will evict the entry in the way that has been referenced the longest time ago. We can therefore view an N-way set associative cache as a collection of stacks of length N corresponding to the N-ways of each set. At the top of the stack is the most recently used (MRU) entry for that set. At the bottom is the least recently used entry (LRU).

A stack distance profile is a histogram with an entry corresponding to each LRU stack position (the total number of entries is equal to the number of cache ways) plus an additional entry for recording cache misses. Each entry in the stack-distance profile counts the number of hits to the lines in the corresponding LRU stack position (for any of the cache sets). For example, whenever an application reuses a cache line that is at the top of the LRU stack (in the most recently used position MRU), the number of “hits” in the first entry of the stack-distance profile is incremented. If an application experiences a cache miss, the number of items in the miss entry is incremented. Figure 2.13 shows an example of a 4-way set associative cache, how several memory references are processed as well as what effect these references have on the corresponding stack-distance profile. The shape of the stack-distance profile captures the nature of the application’s cache behavior: an application with a large number of hits in top LRU positions has good locality of reference. Figure 2.14 shows
an example stack-distance profile for the SPEC CPU2006 benchmark GAMESS on a 16-way set associative cache which demonstrates excellent locality of reference. An application with a low number of hits in the top positions and/or a large number of misses has poor locality of reference. Figure 2.15 shows the stack distance profile for the SPEC CPU 2006 benchmark MILC, which has very poor locality of reference. For our study we obtained the stack-distance profiles using the Pin binary instrumentation tool [60]; an initial profiling run of an application under Pin was required for that.

We now discuss four classification schemes which are based on the information provided in the stack distance profiles.

Figure 2.13: Creating stack distance profiles: Shows a 4-way set associative cache and a corresponding SDP histogram. The entries at the top of the cache are the MRU position and those at the bottom are the LRU. Several address reads and the corresponding change in cache state as well as SDP are shown.
2.3.6 SDC

The SDC\textsuperscript{1} classification scheme was the first that we evaluated, since this is a well known method for predicting the effects of cache contention among threads [15]. The idea behind the SDC method is to model how two applications compete for the LRU stack positions in the shared cache and estimate the extra misses incurred by each application as a result of this competition. The sum of the extra misses from the co-runners is the proxy for the performance degradation of this co-schedule.

The main idea of the SDC algorithm is to construct a merged stack distance profile from the stack distance profiles of the two competing threads. The algorithm proceeds as follows: a pointer is initialized to the first stack positions of the profiles of the two competing applications as well as another pointer to the first position of the merged profile. For the next N-iterations (where N is the associativity of the cache) the entries referenced by the two pointers for the competing applications are compared. The larger of the two is copied into the merged profile entry referenced by the merged profile pointer. The merged profile pointer as well as the pointer of the “winner” of the current iteration are advanced by one position. The extra misses that each of the competing applications will incur is calculated as the difference of the entries that this application contributed to the merged profile and

\textsuperscript{1}Chandra suggested three algorithms for calculating the extra miss rates. However, only two of them (FOA and SDC) are computationally fast enough to be used in the robust scheduling algorithm. We chose SDC as it is slightly more efficient than FOA.
Figure 2.15: **SDP of MILC**: Shows very poor (non-existent data locality of reference).

all of the hits in its solo stack distance profile.

We use the sum of the extra misses incurred by both applications as the proxy co-run degradation in our classification scheme evaluation methodology.

### 2.3.7 Animal Classes

This classification scheme is based on the animalistic classification of applications introduced by Xie et al. [100]. They argue that all applications can be broken into one of four broad classes which they assign animal names based on their characteristics and behavior. The four classes are: *turtle* (low use of the shared cache), *sheep* (low miss rate, insensitive to the number of cache ways allocated to it), *rabbit* (low miss rate, sensitive to the number of allocated cache ways) and *devil* (high miss rate, tends to thrash the cache thus hurting co-scheduled applications).

In order to use this classification scheme with our evaluation methodology we created a so called “symbiosis table” which captures the performance impact of co-scheduling a particular pair of animals together. Conceptually the symbiosis table is a 2D 4x4 matrix with both rows and columns corresponding to each of the animal classes. Every entry is a number that gives the experimentally derived performance impact of co-scheduling the row-animal and the column-animal to the same memory domain.
2.3.8 Miss Rate

This classification scheme is inspired by the results of Section 2.2 where we attempted to quantify the effects that different points of contention have on the overall performance degradation of the CMP. We found that there was no one dominant cause for performance degradation. All the factors -shared cache contention, memory bus contention, DRAM controller contention, and prefetching hardware contention- play a significant role. As such, we desire a classification scheme that can identify applications which will compete for all these points of contention. We propose the LLC miss rate (the number of memory reference requests that missed in the last level cache per number of instructions retired) that an application experiences as a predictor for how much it stresses all the points of contention. This is based on the logic that the miss rate is an indicator of how much the application touches the memory hierarchy and hence the points of contention.

We hypothesized that identifying applications with high miss rates is very beneficial for the scheduler, because these applications exacerbate the performance degradation due to memory controller contention, memory bus contention, and prefetching hardware contention, and to some degree indicate cache usage. To attempt an approximation of the “best” schedule using the miss rate heuristic, the scheduler will identify high miss rate applications and separate them into different memory-domains, such that no one memory-domain will have a much higher total miss rate than any other memory domain. Since no memory domain will experience a significantly higher miss rate than any other the performance degradation factors will be stressed evenly throughout the system.

2.3.9 Pain

Although miss rate is a good heuristic for predicting how much a thread touches the memory hierarchy and in particular the contention for prefetching hardware, the memory bus, and the DRAM controller, it is not necessarily a good predictor for shared cache contention. Competition for cache space is a complex issue which is based on the application’s locality of reference as well as other memory reuse issues. As such the miss rate by itself is insufficient to model the intricacies of contention due to shared cache effects. We designed the Pain Classification Scheme to overcome these inadequacies.

The Pain Classification Scheme is based on two new concepts that we introduce in this work: cache sensitivity and cache intensity. Sensitivity is a measure of how much an
application will suffer when cache space is taken away from it due to contention. Intensity is a measure of how much an application will hurt others by taking away their space in a shared cache as well as stressing the rest of the memory hierarchy. By combining the sensitivity and intensity of two applications, we estimate the “pain” of the given co-schedule. Combining a sensitive application with an intensive co-runner should result in a high level of pain, and combining an insensitive application with any type of co-runner should result in a low level of pain. We obtain sensitivity and intensity from stack distance profiles and we then combine them to measure the resulting pain.

To calculate sensitivity $S$, we examine the number of cache hits that will most likely turn into misses when the cache is shared. To that end, we assign to the positions in the stack-distance profile loss probabilities describing the likelihood that the hits will be lost from each position. Intuitively hits to the Most Recently Used (MRU) position are less likely to become misses than hits to the LRU position when the cache is shared. Entries that are accessed less frequently are more likely to be evicted as the other thread brings its data into the cache; thus we scale the number of hits in each position by the corresponding probability and add them up to obtain the likely extra misses. The resulting measure is the sensitivity value which is shown in Equation 2.6. Here $h(i)$ is the number of hits to the $i^{th}$ position in the stack, where $i = 1$ is the MRU and $i = n$ is the LRU for an n-way set associative cache. We use a linear loss probability distribution. As such the probability of a hit in the $i^{th}$ position becoming a miss is $i/n + 1$.

$$S = \left( \frac{1}{1+n} \right) \sum_{i} i \cdot h(i)$$ (2.6)

Intesity $Z$ is a measure of how aggressively an application uses cache. As such, it approximates how much space the application will take away from its co-runner(s). Our approach to measuring intensity is to use the number of last-level cache accesses per one million instructions.

We combine sensitivity $S$ and intensity $Z$ into the Pain metric, which is then used to approximate the co-run degradations required by our evaluation methodology. Suppose we have applications $A$ and $B$ sharing the same cache. Then the Pain of $A$ due to $B$ approximates the relative performance degradation that $A$ is expected to experience due to $B$ and is calculated as the intensity of $B$ multiplied by the sensitivity of $A$ (Equation 2.7). The degradation of co-scheduling $A$ and $B$ together is the sum of the Pain of $A$ due to $B$
and the Pain of $B$ due to $A$ (Equation 2.8).

\[
\text{Pain}(A_B) = S(A) \times Z(B)
\]

(2.7)

\[
\text{Pain}(A, B) = \text{Pain}(A_B) + \text{Pain}(B_A)
\]

(2.8)

### 2.3.10 Classification Schemes Evaluation

For the evaluation of the classification schemes we collected stack distance profiles offline using Intel’s binary instrumentation tool Pin [60], an add-on module to Pin, MICA [80], and our own module extending the functionality of MICA. The stack distance profiles were converted into the four classification schemes described above: SDC, pain, miss rates, and animal. Although miss rates will later be collected online in our scheduler implementation, in order to keep the classifications as consistent as possible for this particular evaluation all input data was obtained from the same source. We estimate the extra degradation above the optimal schedule that each classification scheme produces for the four-core, six-core, eight-core and ten-core systems (lower numbers are better). The optimal solution as discussed in Section 2.3.1 is the min-weight perfect matching of the graph-theoretical formulation of the scheduling problem using experimentally obtained values as input. Additionally, we present the degradations for the worst and random schedules. A random schedule picks each of the possible assignments in a workload with equal probability. The worst schedule is the max-weight perfect matching solution to the graph-theoretical problem.

Figure 2.16 presents the results of the classification scheme evaluation. The Pain, Miss Rate and Animal schemes performed relatively well, but SDC despite being a very well known algorithm did only slightly better than random. Pain performed the best, delivering only 1% worse performance than the optimal classification scheme for all the systems.

The results of the classification scheme evaluation are in good agreement with the findings of Section 2.2 in regards to the points of contention in the CMP. They also demonstrate that the two classification schemes, miss rate and pain, that we designed as a direct consequence of the causes of contention study performed very well.

The Pain Classification scheme which was designed to explicitly take into account all the points of contention performed the best. The intensity metric used by Pain measures how aggressively the application uses the memory hierarchy and therefore considers contention.
due to the DRAM controller, the memory bus, and the prefetching hardware. The sensitivity metric used by Pain integrates the stack distance profile to capture an application’s memory reuse pattern and therefore provides a measure of the shared cache contention.

The Miss Rate classification scheme performs slightly worse than Pain although Miss Rate still performs very well and significantly better than Random or SDC. Ironically, the theory behind stack-distance based models, like SDC, seems to suggest that the miss rate should be a poor heuristic for predicting contention, since applications with a high cache miss rate may actually have very poor reuse of their cached data, and so they would be indifferent to contention. Our analysis, however, showed the opposite: miss rate turned out to be an excellent heuristic for contention.

We showed that the reason for these seemingly unintuitive results had to do with the causes of performance degradation on multicore systems. SDC, and other solutions relying on stack distance profiles such as cache partitioning [73, 82, 100], assumed that the dominant cause of performance degradation is contention for the space in the shared cache, i.e., when co-scheduled threads evict each other’s data from the shared cache. We found, however, that cache contention is by far not the dominant cause of performance degradation. Other factors, such as contention for memory controllers, memory bus, and resources involved in prefetching, dominate performance degradation for most applications. A high miss rate

![Evaluation of Classification Schemes](image)

Figure 2.16: Evaluation of Classification Schemes
exacerbates the contention for all of these resources, since a high-miss-rate application will issue a large number of requests to a memory controller and the memory bus, and will also be typically characterized by a large number of prefetch requests.

This explains why Miss Rate is such a good heuristic for performance degradations on CMPs. It also explains why Miss Rate is worse than Pain. Pain takes everything into account that Miss Rate does as well as integrates cache contention into its calculations. This further explains why SDC did so poorly and performed only negligibly better than the Random scheduler. SDC does not take into account miss rates in its stack distance competition model. So it only works well in those scenarios where the co-running threads have roughly equal miss rates (this observation is made by the authors themselves [15]). When the miss rates of co-running threads are very different, the thread with the higher miss rate will “win” more cache real estate; this fact is not accounted for by the SDC model.

Another reason that SDC performs poorly is the fact that SDC only models the performance effects of cache contention, but as we showed in Section 2.2.5, this is not the dominant cause for performance degradation and so other factors must be considered as well. We note that it is initially surprising to find that SDC, a model extensively validated in the past, failed to outperform even such a coarse classification heuristic as the miss rate. The crux of the issue is that SDC was previously validated only a simulator which simulated only cache contention. In such a system where cache contention is the only source of contention SDC will perform extremely well while Miss Rate will be a poor heuristic since it does not predict cache usage patterns. However, on a real machine the roles are reversed.

The Animalistic Classification Scheme also performed relatively well. However, the fact that as input it needs both experimental contention data to construct the symbiosis table as well as stack-distance profiles to classify threads into the appropriate animal class makes it an expensive and unattractive solution as compared to Miss Rate, which achieves very comparable results, but only needs online obtainable miss rates. The performance of the Animalistic Classification Scheme falls short of that of the other expensive classification scheme Pain making it even less attractive. The issue here is that by allowing only four classes this classification scheme does not have the granularity of Pain as different threads are given identical classifiers and hence misses opportunities that Pain is able to exploit.
2.4 The Scheduler (DI)

A scheduler consists of two parts: the classification scheme which predicts inter-thread contention properties and a scheduling policy which acts on these predictions to actually move threads around into a mapping which will minimize the contention. Having extensively explored classification schemes in the previous section we arrived at the decision to use the Miss Rate Classification Scheme for our scheduler.

While it is true that the Pain classification scheme gave the best performance, we chose not to use it in an online algorithm, instead opting to implement one using the miss rate heuristic. This was done to make the scheduler simpler, thus making it more likely that it will be adopted in general-purpose operating systems. Using Pain would require more changes to the operating system than using the miss rate for the following reason: Pain requires stack distance profiles. Obtaining a stack distance profile online requires periodic sampling of data addresses associated with last-level cache accesses using advanced capabilities of hardware performance monitoring counters, as in RapidMRC \[92\]. Although RapidMRC can generate accurate stack-distance profiles online with low overhead, there is certain complexity associated with its implementation. If a scheduler uses the miss rate heuristic, all it has to do is periodically measure the miss rates of the running threads, which is simpler than collecting stack-distance profiles. Given that the miss rate heuristic had a much lower implementation complexity but almost the same performance as Pain, we thought it would be the preferred choice in future OS schedulers.

The goal of the Miss Rate Classification Scheme is to spread the miss rate of applications evenly across memory domains such that no single memory domain has an excessively high collective miss rate and corresponding contention. The perfect policy used in Section 2.3.1 for evaluating the goodness of the classification scheme enumerated and evaluated every possible schedule to determine the optimal choice. Clearly such a solution is not practical online and is entirely non-scalable as the number of comparisons grows exponentially with both the number of threads and the number of domains. We wanted a more robust and simpler scheduling policy that could be combined with the Miss Rate Classification scheme to effectively spread the miss rate across the memory domains. To that end we chose Centralized Sort as our scheduling policy. It examines the list of applications, sorts them by their miss rates, and distributes them across cores, such that the total miss rate of all threads sharing a memory domain is equalized across all memory domains. The sorting and
distribution is performed globally on a per machine basis and as such we call this policy Centralized Sort.

We call our scheduler, which is the combination of the Miss Rate Classification scheme and the Centralized Sort scheduling policy: Distributed Intensity or DI. Below is a pseudo-code implementation of DI.

**Distributed Intensity (DI):**

if Scheduling tick expired or if the workload changed then

repeat

Measure the online miss rate of each thread in the system using performance counters.
Place each thread as an entry in the ARRAY.
Sort the entries in the ARRAY based on their miss rates.
Halve the ARRAY by combining each entry with its counter-part and aggregate the miss rate.

until The number of threads in each entry is equal to the number of cores per memory domain

Schedule the threads within each entry to the same memory domain.

end if

The DI scheduler is invoked every scheduling tick (a configurable parameter) or whenever the workload changes because new threads have been spawned or existing threads have terminated. The miss rate of each thread is measured using online performance counters available on all modern processors. The array of threads is then sorted based on their miss rate. The actual sorting can be done by any of the well known sorting algorithms such as quick sort or bubble sort for example. Threads from opposite ends of the sorted array are then paired up halving the size of the array. The thread with the highest miss rate (entry 1) is paired with the lowest miss rate (entry N). The thread with the second-highest miss rate (entry 2) is paired with the second lowest miss rate (entry N-1). And so forth in this fashion. The array is now half the size and each entry contains two threads. If the number of cores per memory domain of this particular machine is two then we are done and each pair of threads from every entry in the array will be scheduled to cores on the same memory domain. Figure 2.17 shows an example of DI applied to 8 threads for a machine with 8 cores and 2 memory domains per core. If the number of cores is however greater than two then we must aggregate the miss rate of the two threads in each array entry and repeat the algorithm. The new array is once again sorted based on this aggregate miss rate. Entries
from opposite ends of the array are paired up creating entries with 4 threads in them. If the machine has 4 cores per memory domain then we are done and each quartet of threads will be scheduled to a memory domain. Figure 2.18 shows an example of DI with a 4-cores per memory domain machine. If the machine has more than four cores the algorithm is repeated.

**Figure 2.17: DI on a 2 core per memory-domain machine:** DI first measures the miss rate of applications. It then sort them based on the miss rate. It pairs applications from opposite ends of the array and schedules them to the same memory domain.

DI handles the case when the number of threads is less than the number of cores by padding the initial array to match the number of cores by adding empty threads with a miss rate of 0. This ensures that the highest miss rate threads will be paired with these empty threads which in effect will see the highest miss rate threads scheduled alone to memory domains. The case when the number of threads exceeds the number of cores is not explicitly handled by DI as it would require time sharing of cores as well as space-sharing of
Figure 2.18: DI on a 4 core per memory-domain machine: Starting at the point where the 2-core implementation left off, DI aggregates the miss rates of the application pairs. It then sort them based on the aggregate miss rate. It combines application pairs from opposite ends of the array and schedules them to the same memory domain.

...the machine. As discussed at the beginning of this section the complex interaction between space and time sharing is well outside the scope of this work.

The evaluation of DI as well as its comparison to other contention-aware scheduling algorithms is discussed in Section 3.

2.5 Summary

In this chapter we introduced our contention-aware-thread-level-scheduler called Distributed Intensity (DI). DI was developed as a result of two major investigations into the causes of shared resource contention in CMPs. Given the vast amount of literature which chooses to focus on one specific point of contention, such as the shared Last Level Cache we first
aimed to resolve the question as to whether or not there is a dominant source of contention, the mitigation of which should be the goal of our scheduler. We performed a systematic investigation by mapping threads to different combinations of cores which share different sets of resources and from the results were able to estimate the performance degradation caused by each of LLC contention, memory bus contention, DRAM controller contention, and prefetcher contention. We showed that no contention factor by itself is dominant and any effective solution must be a comprehensive solution that deals with all of these contention factors. Using this information we designed two novel thread classification schemes. One of them, Pain, explicitly addresses all four contention factors but requires raw data as input which is difficult if not impossible to obtain online. The other, Miss Rate, addresses three of the contention factors fully and one partially however its input data can be easily obtained dynamically on modern hardware. Using a technique which we pioneered we evaluate the optimality of these classification schemes against the best possible, the worst possible, the statistical random, as well as two other well known classification schemes. We show that both Pain and Miss Rate outperform the other classification schemes and are very close to the theoretical optimum. Despite Pain performing better than Miss Rate we decided to employ Miss Rate in our DI scheduler due to its simplicity and higher likelihood of adoption into commercial Operating Systems. We combine the provably good Miss Rate classification with a scheduling policy that we have developed to create the DI scheduler. The DI scheduler mitigates shared resource contention by spreading the aggregate miss rate of threads across different memory domains therefore ensuring that no single memory domain has an excessively high miss rate which stresses the shared resource memory hierarchy and leads to poor performance.
Chapter 3

Comparing Contention-Aware Schedulers

In Section 2.4 we introduced our contention-aware thread level scheduler entitled Distributed Intensity (DI). DI is based on the Miss Rate thread classification scheme and has the goal of separating high-miss rate threads to different memory domains in order to even out the aggregate misses in each memory domain and thus minimize the overall contention for shared resources. The logic behind this strategy has been extensively validated in Section 2.3. In this section we directly evaluate the performance of DI as implemented on a real machine and compared to the default Linux contention-unaware scheduler. We also present a comprehensive review of literature available on contention-aware schedulers and discuss how the DI scheduler fits in.

3.1 Evaluation of DI

3.1.1 Evaluation Platforms

The experiments were performed on two different CMP architectures. The machines differed in their processor manufacturers, the number of cores per memory-domain as well as the cache and memory controller architectures.

Dell-Poweredge-2950 (Intel Xeon X5365) has eight cores placed on four chips. Each chip has a 4MB 16-way L2 cache shared by the two cores. Each core also has private L1 instruction and data caches. We performed two sets of experiments on this machine. In
the first, we used only half of the available cores (four cores in two memory domains) thus effectively testing our algorithm on a four-core Xeon architecture. In the second, we used the full capability of the machine (all eight cores and four memory domains) thus testing our algorithm on an eight-core Xeon architecture.

**Dell-Poweredge-R805** (AMD Opteron 2350 Barcelona) has eight cores placed on two chips. Each chip has a 2MB 32-way L3 cache shared by the four cores. Each core also has a private unified L2 cache and private L1 instruction and data caches. Experiments were conducted using all eight-cores of the machine. Since this machine only has two memory domains it was impossible to subdivide it and have a contention-aware scheduler show any improvements (as per assumption number two in Section 2.1).

Both systems were running Linux Gentoo 2.6.27 release 8. We compare performance under the DI scheduler to the default contention-unaware scheduler in Linux, which we refer to henceforth as DEFAULT.

**3.1.2 The Evaluation Workloads and Experimental Methodology**

The workloads for the evaluation were made up of 14 benchmarks from the SPEC CPU 2006 benchmark suite which were chosen via the clustering technique described earlier. Table 3.1 shows the eight distinct workloads (4-thread combinations) that were created from the 14 benchmarks. The table shows the number of memory-bound versus the number of CPU-bound threads in each workload. We ensured that various ratios of memory- to CPU-bound threads were present in the workloads.

For experiments on the four core machine each of the eight workloads were executed. For experiments on the eight-core machines eight-thread workload were created by duplicating each of the four-thread workloads. For example, for the four-thread workload (SOPLEX, SPHINX, GAMESS, NAMD) the corresponding eight-thread workload is (SOPLEX, SPHINX, GAMESS, NAMD, SOPLEX, SPHINX, GAMESS, NAMD). We note that for all experiments the number of threads was equal to the number of cores this is in agreement with assumption number 2 in Section 2.1 about contention-aware schedulers.

A single experiment proceeds as follows: All the threads of the workload are launched simultaneously. We monitor the system to determine if any threads terminated and immediately restart them. Due to the different run times of the applications, constantly restarting them is the only way to ensure that the entire workload is active on the machine. We continue this until every thread has executed at least three times. The run-time of each thread
CHAPTER 3. COMPARING CONTENTION-AWARE SCHEDULERS

is recorded as the average execution time of all of its runs.

The same set of experiments is conducted using the default Linux scheduler and using DI. When DI is used it enforces thread-to-core-mapping via system calls which set a thread’s affinity to only one specific core.

<table>
<thead>
<tr>
<th>Workloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 memory-bound</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.1: DI test workloads: The eight thread combinations (workloads) that were used to evaluate the DI scheduler. The number of memory- and CPU-bound applications in each workload is shown. Memory bound applications are shown italicized. The solo execution times of the applications vary between one and fifteen minutes.

3.1.3 Evaluation Results

Intel Xeon 4 cores:

We begin with the results for the four-thread workloads on the four-core configuration of the Intel Xeon machine. For this particular set-up we were unable to evaluate DI versus the default Linux scheduler because when we restricted the default scheduler to only use the same four cores via an affinity mask we observed that the default scheduler migrated threads at an unusually high rate causing unfair performance skew. As such we used the methodology introduced in Section 2.3.1. For every 4-thread workload we ran each of the three perfect-matchings and measured their performance; thus we obtained the performance of the BEST schedule, the WORST schedule, and the RANDOM schedule (which is a linear combination of the three schedules). We then ran DI on the same workload and were able to determine how the performance of DI compares with the optimal, the worst, and the random
solution. Figure 3.1 shows the performance degradation that each scheduler experiences for every workload above the optimal solution. Performance degradation is calculated as the aggregate performance degradation of every thread in the workload. Per thread performance degradation is calculated as the percent increase in execution time during the experiment as compared to solo execution time.

As can be seen from Figure 3.1 DI performed significantly better than both RANDOM and WORST. For all workloads DI was never more than 2% worse than the optimal solution.

![Figure 3.1: DI results on Xeon quad-core: The percent degradation of DI above optimal as compared to the random and the worst schedules. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).](image)

**Intel Xeon 8 cores:**

Since this setup does not require setting an affinity mask, and hence no unexpected thread migrations occur, we evaluated the performance of DI directly against DEFAULT. Figure 3.2 shows the aggregate percent speedup that each workload experienced under DI as compared to under the default scheduler.

We note that although generally DI improves the aggregate performance over DEFAULT, in a few cases it performed slightly worse. However, the biggest advantage of DI is that it offers much more stable results from run to run and avoids the worst-case thread assignment.
This effect is especially significant if we look at the performance of individual applications rather than at workload aggregates. To illustrate this point we measured the single worst case execution time that every thread experienced in every workload under DI and under DEFAULT. Figure 3.3 shows the relative performance improvement for every thread’s worst-case performance under DI versus under DEFAULT.

The results show that DEFAULT consistently stumbles onto much worse solutions than DI and as such there are cases when the performance of individual applications is unpredictably bad under DEFAULT. What this means is that if an application is repeatedly executed on a multicore system, running it under DEFAULT rather than under DI may occasionally cause its performance to degrade by as much as 100% in some cases! Figure 3.4 shows the deviation of the execution times of consecutive runs of the same application in the same workload under DI and under DEFAULT. We note that DEFAULT has a much higher deviation from run to run than DI.

Figure 3.2: **DI results on Xeon eight-core:** The percent aggregate per workload speedup that DI delivers over the default Linux scheduler. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (High bars are good).
Figure 3.3: **DI worst-case speedup Xeon eight-core:** The percent worst-case per thread speedup that DI delivers over the default Linux scheduler. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (High bars are good).

**AMD Opteron 8 cores:**

Finally, we report the results for the same eight-thread workloads on the AMD system. Figure 3.5 shows the aggregate workload improvements that DI delivers over DEFAULT. For this platform the improvements are as high as 11% and DI never performs worse than DEFAULT. Figure 3.6 shows the worst-case per thread improvements that DI delivers over DEFAULT. Figure 3.7 shows the run-to-run performance deviations that individual threads experience under DI and DEFAULT.

### 3.1.4 Evaluation Conclusions

We draw several conclusions from our results. First of all, using contention-aware scheduling can help improve overall system efficiency by reducing completion time for the entire workload as well as reduce worst-case performance for individual applications. In the former
Figure 3.4: **DI variability Xeon eight-core**: The variability from run to run of every thread under DI and under the default Linux scheduler. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).

In this case, DI improves performance by up to 13% relative to DEFAULT and in the isolated cases where it does worse than DEFAULT, the impact on performance is at most 4%, far smaller than the corresponding benefit. On average, if we examine performance across all the workloads we have tried, DEFAULT does rather well in terms of workload-wide performance. But if we consider the variance in completion times and the effect on individual applications, the picture changes significantly. DEFAULT achieves a much higher run-to-run deviation and it is likely to stumble onto a much worse worst-case performance for individual applications. This means that when the goal is to deliver QoS, achieve performance isolation or simply prioritize individual applications, contention-aware scheduling can achieve much larger performance impacts, speeding up individual applications by as much as a factor of two.
To understand why DEFAULT performs relatively well on average let us discuss several examples. Consider a four core, two memory domain machine. If the workload to be executed on this machine involves two memory intensive applications and two non-intensive applications and if the threads are mapped randomly to cores (which is a good approximation for DEFAULT) then there is only a 1/3 probability of stumbling onto the worst solution where the intensive applications share the same memory domain. If there are three intensive applications in the workload and only one non-intensive application then all mappings are relatively equivalent on average since two of the intensive applications will experience performance degradation and one will not (the one paired with a non-intensive application in the same memory domain). Similarly, workloads with one intensive applications show no real difference between solutions. As such DEFAULT is able to perform well on average.

3.2 How DI Fits in with Other Contention-Aware Schedulers

Though significantly less work has been done on using contention-aware schedulers to mitigate shared resource contention in CMPs than on other techniques such as cache partitioning
there is some literature on the topic. In particular the work of [45, 22, 6, 62, 64, 25, 41, 94]. We discuss how our DI scheduler fits in with the other contention-aware schedulers proposed thus far.

Knauerhase et al. use LLC misses per cycle as the input metric for their contention-aware scheduler [45]. This is similar to the LLC misses per instructions retired used in our DI scheduler as the two metrics are both measures of the application’s miss rate. They propose three scheduling policies: OBS-L, OBS-X, and OBS-C. The OBS-L policy attempts to reduce cache interference by spreading the total misses across all cache groups (a cache group consists of the shared LLC and the cores attached to it). Whenever a core becomes available, it selects a thread whose miss rate is most complementary to the other threads sharing this cache. The OBS-X policy attempts to spread the miss rate even more by adding new threads to the cache-group that has the smallest total miss rate. Furthermore, periodically
the thread with the highest miss rate is moved from the cache-group with the highest total miss rate to the group with the lowest miss rate. Finally, based on the observation that their decision mechanism pairs cache-heavy and cache-light threads and that the cache-light threads tend to suffer slightly due to this pairing, the OBS-C compensates the light weight threads by extending their time slices, similarly to an idea proposed by Fedorova et al. [25]. Although both this work and DI use a similar idea of spreading high miss applications we provide a more rigorous analysis of this idea and the reasons for its effectiveness and also demonstrate that DI operates within a narrow margin of the optimal. The other paper did not provide the same analysis, so it was difficult to judge the quality of their algorithm comprehensively. Our work is broader in a sense that we explore the limitations of the DI algorithm in Section 4.2 as well as introduce new scheduling algorithms based on online
optimization through trial and error. Furthermore, we also analyze how these algorithms would scale to futuristic multicore machines in Section 4.2.4. We note, however, that our work is also more limited than the work by Knauerhase since unlike Knauerhase we do not consider situation where multiple threads share the same core. This self-imposed limitation allowed us to simplify the scheduling problem and provide the above described theoretical treatment.

Dhiman et al. found that when Virtual Machines (VMs) running memory-intensive applications are executed on the same Physical Machine (PM) performance significantly degrades resulting in longer execution times, which also translates into higher energy use [22]. The finding that memory-intensive applications suffer performance degradations and prolonged run times matches our findings. The authors propose a VM management utility called vGreen which schedules the VMs to the PMs in such a way as to balance the MPC (misses per cycle), IPC, and CPU-utilization across the PMs. Doing such balancing, which boils down to scheduling a heterogeneous mix of compute- and memory-bound VMs to each PM, reduces shared resource contention between the VMs facilitating a faster execution time and a reduced power usage. The spreading of high miss rate VMs across different PMs is similar to the notion used by DI to spread high miss rate threads across memory domains. However, there are important differences. Once vGreen has distributed the VMs across the PMs the default OS contention-unaware scheduler manages their thread-to-core mapping which still exposes the solution to all the inter-CMP shared resource issues which DI explicitly avoids. Furthermore, the authors performed very limited experiments using only two types of SPEC benchmarks: one of the most memory-intensive applications possible and one of the least memory-intensive applications possible. Moreover, they created an equal proportion of memory-intensive and non memory-intensive VMs which is the easiest to optimize scenario. They did not explore how vGreen would perform with VMs running applications with less clearly defined properties or different ratios of memory-intensive to non memory-intensive VMs. We, on the other hand, explored both issues thoroughly both in the evaluation of DI as well as the extensions developed for DI in Section 4.2. Finally, the authors did not attempt to explain why miss rate was a good contention avoidance predictor and only hypothesized that it should lower shared cache contention. We conducted an extensive study of shared resource contention and showed the true reasons why miss rate is a good predictor.
Banikazemi et al. created a contention-aware scheduler that used a very involved performance prediction model [6]. As input they required cycles per instruction (CPI), the ratio of L1 misses to references, the ratio of L2 misses to references, L2 pre-fetching events, and the number of floating point instructions. They convert the obtained thread performance metrics into a predicted CPI for every thread in any possible mapping. To predict the performance of a particular thread in a particular mapping they first calculate the cache occupancy ratio which is the ratio of the LLC access rate of the thread of interest to the LLC access rate of all the threads that share this cache in the mapping. Next, they calculate the would-be LLC miss rate that this thread should experience given its currently measured LLC miss rate, the calculated cache occupancy ratio, and a rule of thumb heuristic. Finally, they use a linear regression model to convert the predicted LLC miss rate as well as the currently measured L1 miss rate into a predicted CPI for the thread, which directly represents the thread’s predicted performance in the proposed mapping. Performing this calculation for all the threads in the proposed mapping allows the authors to predict the performance of the entire workload given that mapping. They use a very rudimentary technique of enumerating all possible mappings and selecting the one with the best performance to find the actual thread-to-core-mapping. Although, guaranteed to be the optimal solution there are major scalability issues with such an approach when one attempts to go to a machine that is larger than that which was used by the authors (8 cores and 4 shared caches). As pointed out by [41], the number of choices explodes as the number of cores sharing the LLC increases. The computational complexity of their performance prediction model which converts the performance counters into predicted CPI for every thread in every possible mapping is also orders of magnitude more complicated and computationally costly than DI. As such the viability of such a costly and complex system being adopted in real Operating Systems is minimal.

McGregor et al. define the concept of high-pressure and low-pressure threads based on the number of bus transactions that the threads complete and attempt to schedule high and low-pressure threads together [62]. The decision mechanism proposed is however integrated with a multithreading library and is significantly more complex because it deals with multithreaded applications that provide resource requests to the OS. A complete discussion of such a scheduler is well outside the scope of this work; however we note that the contention avoidance mechanism used is similar to that of DI since they too attempt to combine threads with complementary resource usage.
While developed independently, the work by Merkel et al. can be characterized conceptually as extending ideas in DI. Merkel et al. used the number of memory-bus transactions, the number of LLC cache line requests, and the number of retired instructions per time slice as the input to their scheduler [64]. They attempted to formalize the concept of thread similarity using so-called activity vectors. A thread's activity vector records its usage of system resources during the previous time slice. The usage of the memory-bus, the LLC, and the rest of the core normalized to the theoretical maximum usage of each. The authors attempted to co-schedule threads with complementary activity vectors. They formalize this concept by measuring the variability of the activity vectors of threads within the run queue of a given core. Higher variability is an indication that the current subset of threads will yield high performance if co-scheduled. Their scheduling policy augments the standard OS scheduler to make thread migration decisions such that they increase the activity vector variability in the runqueues of both of the cores involved in the migration. They also introduce a more involved decision technique called sorted co-scheduling which groups cores into pairs and attempts to schedule only complementary threads on each core within the pair. The sorted co-scheduling technique requires focusing on only one parameter of the activity vector which is deemed most important. It then involves keeping the runqueues sorted based on that parameter; one core in the pair has the threads sorted in ascending order while the other in descending. In order to ensure synchronized scheduling of threads this technique requires manipulating the time slice mechanism of the OS scheduler.

Fedorova et al designed a cache-aware scheduler that compensates threads that were hurt by cache contention by giving them extra CPU time [25]. This algorithm provides fairness to threads that were hurt by shared cache contention. However, it does nothing to mitigate shared cache contention or improve overall throughput. In fact, by increasing the CPU time slices for threads which are experiencing performance degradation due to shared resource contention the scheduler is in effect prolonging the duration of the bad schedule and possibly hurting overall throughput even further.

[41, 94] address the contention-aware scheduling problem exclusively from the perspective of making the actual thread-to-core mapping decisions. We used concepts developed in [41, 94] as a basis for our methodology for designing contention-aware schedulers in Section 2.3.1. They assume that the exact performance degradations of any subset of threads co-scheduled to the same memory domain are known. They represent the problem in graph-theoretical form; the nodes of the graph are the threads to be scheduled. The edges between
them have weights equal to the performance degradation that would be incurred if the applications were scheduled to the same memory domain. They show that the optimal solution to this scheduling problem is a minimum weight perfect matching of the graph. For systems where memory domains are only shared by two cores the problem has many well known graph theoretical polynomial-time solutions, such as the Blossom algorithm. However, if more than two cores share a memory domain, the problem is NP-complete as proven by [41]. This means that even from a strictly theoretical perspective an exact solution cannot realistically be found online for systems with more than two cores per memory domain. Yet machines with four and six cores per cache are already widely available. [41, 94] present a series of polynomial time approximation algorithms, which they show can result in solutions very near to optimal. Although this work provides well needed theoretical background to the scheduling problem we view it as orthogonal to the DI scheduler. The unrealistic requirement that exact co-run performance degradations be known \textit{a priori} makes the proposed algorithms entire infeasible as real system schedulers.

Having reviewed the available literature on contention-aware schedulers we may conclude that the DI scheduler is a unique contribution to this field. Three factors distinguish DI from other contributions to the space of contention-aware schedulers. First, DI is based on an extensive study of shared resource contention and methods to detect and avoid it. As such, DI is based on a solid foundation which explains why this solution works. Second, DI has been extensively validated on real hardware and evaluated both against the default contention-unaware scheduler as well the theoretic optimal. The properties of DI are therefore well explored and we clearly outline the situations where it is effective and where it is not. This study is also continued in Section 4.2. Finally, DI is very light weight especially as compared to some of the other contention-aware schedulers and is therefore capable of being implemented in a commercial Operating Systems without risk of creating undo overhead. DI, however, is not a complete scheduler on its own and there are directions of future research which must be explored prior to it being able to handle arbitrary workloads. Most significantly, in or study of DI we assumed scenarios where the number of threads never exceeded the number of cores; extending DI to explicitly handle the time-sharing of cores is an important future direction. We also focused exclusively on single-threaded applications; extending DI to be aware of and deal with multithreaded applications is another future direction for development. Finally, throughout our study of DI we used only compute-bound applications which perform minimal IO; extending DI to deal with IO-bound applications
or properly interact with another scheduler like Linux’s CFS in order to handle IO-bound threads is yet another future direction for DI.
Chapter 4

Toolset for Creating Contention-Aware Schedulers

The questions that need to be answered by a contention-aware scheduler are: how to place the threads into memory domains in order to minimize contention for shared resources? Which threads should be placed in the same domain, and which threads should be placed apart? The difference between a good thread placement policy and a poor one can be quite significant. Even for a small number of cores, memory domains, and threads, there are a great many possible thread placements to choose from and this number grows exponentially with the number of cores and threads. Selecting the optimal schedule depends on the characteristics of the threads, the architecture of the machine, and the metric that the scheduler is trying to optimize.

The problem is made even more difficult by the fact that the threads to be scheduled on a machine may have vastly diverse characteristics which are not known a priori and will need to be discovered online with the use of the available performance counters. On top of this, there is no guarantee that scheduling solutions that work well on systems with two cores per memory domain will carry over to systems with four or six cores per domain, or even to future generations of the same processor. All this creates an enormous design space of scheduling algorithms for developers to explore, while the mounting pile of evidence points to the importance of finding good scheduling algorithms for CMPs.

The DI scheduler introduced in Section 2.4 and evaluated in Section 3.1 has been shown to be a good and promising solution. However, it is by no means the only solution. In fact
CHAPTER 4. TOOLSET FOR CREATING CONTENTION-AWARE SCHEDULERS

it should be regarded as a just one point in the vast design space of scheduling algorithms. Moving along one of the many dimensions within this space yields more questions, more design choices, and new schedulers. What if the scheduler is optimized for fairness instead of throughput? What if certain thread were given higher priorities? DI uses only two performance counters but dozens are available on modern hardware. Would using any of these or combinations of these enable better solutions? How will DI scale with the number of cores, the number of memory domains, and the number of cores per memory domain? Would a distributed version of DI without a centralized sorting operation be needed for futuristic highly multicore machines? These are just a sample of the questions that can and should be explored within the contention-aware scheduler search space.

We created the AKULA toolset specifically to aid developers in narrowing down and rapidly exploring the design space of scheduling algorithm. AKULA addresses what we believe to be two of the biggest difficulties in scheduling algorithm development: the difficulty of implementation and the duration of testing. The difficulty of implementation refers to the time and effort needed to convert an idea into the actual code that places threads on cores and migrates them among cores when needed. Implementing a thread placement algorithm inside the kernel is not a trivial task. Implementing a thread placement algorithm as a user level process, which relies on system calls to read performance counters and to bind threads to cores, may be easier than kernel programming but is also far from trivial. The duration of testing refers to the large amount of time required to test and validate a scheduling algorithm with sufficiently many workloads. Popular benchmark suites like SPEC CPU2006 typically have applications with run times of several minutes and hence testing a variety of workloads constructed from these benchmarks will take hours or even days.

The difficulty of implementation and the duration of testing make it infeasible to explore many different scheduling algorithms. AKULA aims to change this by making it easy to implement thread placement algorithms via our user friendly API and to rapidly test algorithms using our novel performance evaluation technique called the bootstrap method.

Using the AKULA toolset we discovered interesting problems with the DI scheduler and found effective solutions. For example, we found that DI was effective in eliminating contention only in cases where the workload consisted of threads that could be clearly categorized as either compute-intensive (those with a very low last-level cache miss rate) or memory-intensive (those with a very high cache miss rate). But when the workload had threads that could not be easily assigned to any of these coarse categories, the algorithm
failed to effectively reduce contention. Using AKULA we were able to quickly design an algorithm that works effectively for a wider range of workloads. We also explored the scalability of DI and DI-like algorithms to futuristic massively multicore systems.

The rest of this chapter is structured as follows. The AKULA toolset is discussed in detail in Section 4.1. We showcase AKULA’s utility in Section 4.2 in a series of case studies where AKULA is used to develop and evaluate different scheduling algorithms. That section also validates results obtained using AKULA against those obtained using experiments on real systems. Section 4.3 discusses how AKULA fits in with other performance emulation tools proposed in literature and Section 4.4 summarizes our findings.

4.1 The AKULA Toolset

The purpose behind the AKULA toolset is to allow algorithm developers to quickly and painlessly convert an idea for a scheduling algorithm into a working scheduling algorithm, which can then be rapidly evaluated. If the evaluation shows that the algorithm achieves the desired goals then the developer can move on to implementing this scheduler inside the kernel having high confidence that it will be a successful scheduler. On the other hand, if the evaluation shows the scheduling algorithm is lacking in certain aspects then the developer can work on improving this algorithm or move on to a completely different idea without having wasted significant time or effort on this first algorithm.

Figure 4.1 shows the flow chart for the intended use of the AKULA toolset. Once an abstract idea for a scheduling algorithm is converted into the actual implementation using the AKULA API and library, it is first rapidly evaluated using the Bootstrapping Module. This first evaluation step is done in mere seconds, thanks to our new bootstrap evaluation methodology, which we describe later. The Bootstrapping Module is a great first step for rapidly filtering out unsuccessful algorithms. Schedulers that are successful in the bootstrap evaluation are then evaluated on a real machine using the Wrapper Module, which translates the simplified implementation of the scheduler into a real one. If successfully evaluated on a real machine then the developer, confident that this scheduling algorithm will achieve the desired goals, may proceed to spend the time and effort to implement the algorithm inside the kernel or as an AKULA-independent user level scheduler.

To make evaluations easier, the statistics from each experiment are output in tabular form, which can be imported into popular statistical packages, using the Statistics Module.
Figure 4.1: **AKULA flow chart**: The intended use of the AKULA toolset to develop scheduling algorithms. A scheduling algorithm is created with the API, quickly tested with the Bootstrap Module, if successful tested with the Wrapper Module, if successful then it is safe to implement it inside the kernel.

### 4.1.1 Implementing a Scheduling Algorithm with AKULA

A scheduling algorithm is implemented in the AKULA toolset with a Java class that supports two functions: `launchThread` and `updateSchedule`. The function `launchThread(AKULAthread, Machine)` is called whenever a new thread is ready to be launched on the machine so that the scheduler can assign the new thread to a particular core. The function `updateSchedule(Machine)` is called whenever a thread terminates as well as every scheduling interval (whose length is configured by the user) so that the scheduler may modify the thread mappings. The machine is represented by an instance of the `Machine` class. This class contains members...
of the Chip class, which represent the memory domains of the machine. The Chip class contains several members of the Core class that represent the cores. All threads running on the machine are instances of the AKULAThread class.

The AKULAThread class contains basic data about a thread such as its name, source, the time launched, the time spent on processor, etc. Data about the thread’s performance obtained from performance counters is also stored in the member variables of the AKULAThread object. The developer may specify which performance counters should be monitored for each thread or opt to use one of the default performance counter combinations, such as IPC (instructions per cycle) or the last-level cache miss rate.

For example to measure the IPC for threadA the AKULA scheduler would invoke threadA.addParameter("IPC"). Then, every scheduling interval the IPC of threadA will be recorded and the scheduler may read its value by calling threadA.readParameter("IPC").

Threads are mapped to cores by adding them to the instances of the Core class. For example, to add threadA to the first core of the first memory domain of the Machine instance M, the AKULA scheduler would call M.getDomain(0).getCore(0).addThread(threadA).

AKULA provides a variety of useful functions for gathering information about cores, memory domains, threads, as well as changing their state. For example, to evenly spread all threads across the cores in memory domain zero of Machine M, the AKULA scheduler would call M.getDomain(0).balanceLoad(). Functions that manipulate the cores themselves are also available in the AKULA API. For instance, if the hardware supports dynamic frequency scaling, then it is possible to change a core’s frequency using the method Core.setFreq(). If the hardware supports dynamic enabling/disabling of pre-fetching, AKULA also allows to dynamically turn it on or off. The reason for supporting these functions is that they can be useful for alleviating contention in certain situations, as shown in previous work [103].

The AKULA scheduler enforces the assignment of threads to cores, as specified by the algorithm developer, via system calls that create an affinity between a thread and a core. System calls for enforcing affinity are available in modern operating systems. In the event that more than one thread was assigned to a given physical core, the OS will time-share the core among the threads. However, it is also possible to directly manipulate how time sharing is performed using methods in the AKULA API. Although thread mapping via affinity system calls can also be performed directly via the system calls, the main advantage of using AKULA is the simplicity of implementation. The developer can create complex scheduling algorithms without ever worrying about affinity masks, reading performance
counters or having to deal with many other system details that make thread manipulation difficult.

To demonstrate how simple it is to create a new scheduling algorithm using AKULA, Figure 4.2 shows sample code for a scheduling algorithm to which we refer as Naive_Spread. This algorithm balances the thread load across all cores. Although this particular algorithm is rather uninvolved, this example vividly demonstrates the simplicity of using AKULA. Those familiar with implementation of scheduling algorithms in the kernel, or even with their prototyping at user level, would appreciate how much simpler the AKULA code is compared to what would be needed to express the same policy in a real implementation.

```java
public void launch_thread(Thread new_thread, machine M) {
    //Put the thread on the busiest chip.
    chip target_chip = M.get_min_load_chip();
    target_chip.get_min_load_core().add_thread(new_thread);
    target_chip.update_load();
}

public void update_schedule(machine M) {
    //Check if there is a large enough difference to move threads.
    chip min_load_chip = M.get_min_load_chip();
    chip max_load_chip = M.get_max_load_chip();
    if (min_load_chip.load < (max_load_chip.load - 1)) {
        core remover_core = max_load_chip.get_max_load_core();
        core adder_core = min_load_chip.get_min_load_core();
        Thread migrant = remover_core.runQ[0];
        remover_core.migrate_thread(migrant, adder_core);
        min_load_chip.update_load();
        max_load_chip.update_load();
    }
}
```

Figure 4.2: **AKULA API example:** An implementation of Naive_Cluster using the AKULA API. The two functions `launch_thread()` and `update_schedule()` are required by all schedulers using AKULA

### 4.1.2 Bootstrapping Module: Rapid Evaluation

Once a scheduling algorithm has been created it must be evaluated to determine how it meets performance goals, such as throughput and fairness. The evaluation process can be very time consuming if one wishes to evaluate a wide range of different workloads. AKULA
provides a rapid evaluation option via its Bootstrapping Module.

The Bootstrapping Module does not actually execute any benchmarks to evaluate the scheduling algorithm (this task is done by the Wrapper Module described below). Instead, the Bootstrapping Module uses previously obtained performance data to roughly approximate the relative performance of different scheduling algorithms via a coarse simulation.

The key components of the bootstrap data are the application’s solo execution time, measured on a real system when an application runs alone, without contention from other applications, and the degradation matrix, which contains, for a set of target applications, performance degradation values when each application is co-scheduled with every other application in the same memory domain.

Bootstrap data must be obtained on a real system prior to running an evaluation. The user can obtain her own data by picking her own benchmarks and running her own experiment, or use the pre-built data available in the AKULA repository. The process of gathering the bootstrap data may be time consuming and tedious, and so to make it simple and automatic we developed the Profiler Module within AKULA. Given this module, the user can collect the bootstrap data by supplying a set of executables, which will be used as the benchmarks during the measurement.

Although the process of obtaining the bootstrap data may take anywhere from a few hours to a few days depending on the number of applications and their running time, this data must be obtained only once. After that, the user can run evaluations of sophisticated scheduling policies on large multicore systems in just seconds, saving many hours of time with each experiment.

Consider a machine that consists of two memory domains with two cores each, such as in Figure 4.3. We launch four threads A, B, C, and D on this machine simultaneously. (These four threads could be, for instance, four benchmarks from the SPEC CPU2006 suite.) Table 4.1 and Table 4.2 show the necessary bootstrap data to perform this evaluation. Table 4.1 gives the execution times of each of the threads when they run alone on a memory domain without any contention. Table 4.2 gives the slowdown that each thread would experience if sharing the memory domain with every other thread. For example, the entry at row A and column B shows that when thread A is co-scheduled to the same memory domain as thread B it executes at only 0.5 of the speed it has when running alone on a memory domain.

In addition to the bootstrap data the developer also provides to AKULA the machine
architecture on which to perform the evaluation (in this case a system with two cores and two memory domains). Other system details, such as CPU speed and memory hierarchy, need not be specified, because their effects on applications are implicitly captured in the bootstrap data. As will be explained shortly, the Bootstrapping Module does not attempt to precisely simulate the execution of workloads under different schedulers, but only roughly estimate the effects of various thread placement policies.

Finally, the user also supplies the workload configuration file. This file contains all the threads that will be run in this experiment and the wall clock time when each thread will be launched.

AKULA supports multi-phased threads that are represented as arrays of single-phased threads. In this example we specify the workload to consist of four single-phased threads A, B, C and D, whose bootstrap data is described in Table 4.1 and Table 4.2, and which are set to be launched at time $t = 1$ second.
Table 4.1: **Bootstrap solos**: Solo execution times of each application.

<table>
<thead>
<tr>
<th></th>
<th>Solo Exec. (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>150</td>
</tr>
<tr>
<td>C</td>
<td>175</td>
</tr>
<tr>
<td>D</td>
<td>200</td>
</tr>
</tbody>
</table>

Table 4.2: **Bootstrap degradation matrix**: A degradation matrix for four applications. Each cell shows the degradation from the solo execution time when the two applications in the corresponding row and column are co-scheduled on the same memory domain.

<table>
<thead>
<tr>
<th></th>
<th>alone</th>
<th>with A</th>
<th>with B</th>
<th>with C</th>
<th>with D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.00</td>
<td>0.75</td>
<td>0.50</td>
<td>0.98</td>
<td>0.99</td>
</tr>
<tr>
<td>B</td>
<td>1.00</td>
<td>0.60</td>
<td>0.30</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td>C</td>
<td>1.00</td>
<td>0.99</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>D</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

An evaluation proceeds by repeating the following four steps in a loop until all threads in the workload have terminated. The simulated time interval between each iteration of the loop, known as a tick, is set by the user prior to running an evaluation.

1. Calculate the progress of each thread in the system.
2. Determine if any threads completed; remove them from the system and send all their associated performance data to the **Statistics Module**.
3. Check if any threads in the workload file are ready for launch; if yes then call the scheduler so that it places these threads onto the system.
4. Call the scheduler to allow it to modify the current thread placement.

The key to this evaluation methodology is the ability to calculate the progress that each thread makes in each time interval. This progress is calculated using the bootstrap data and the formula shown in Equation 4.1.

\[
Progress(X) = 100\% \times \frac{\text{tick}}{\text{Solo}(X)} \times \text{deg}(X, \text{Neighbour}(X)) \tag{4.1}
\]

\(Progress(X)\) refers to the fraction of total work that a thread completes in a given scheduling interval. \(\text{Tick}\) refers to the length of the scheduling time interval. \(\text{Solo}(X)\) is the thread’s solo completion time obtained from the bootstrap data and \(\text{deg}(X, \text{Neighbour}(X))\)
is the slowdown (or degradation) that thread $X$ experiences when it shares a domain with the neighbour or neighbours assigned to it in the given thread schedule.

When a thread runs in a memory domain alone, the progress that it makes is equal to the length of the scheduling clock interval (i.e., $tick$) divided by the time to execute the entire thread. When other threads compete for resources we must scale this value by the performance degradation that these “neighbors” impose on the target thread.

Figure 4.4 shows the results of the evaluation step by step. The progress is calculated for each thread at each tick using Equation 4.1 and the total progress is tabulated over all the previous steps. At time 0 the machine is empty and threads have not been launched so they have zero progress. At time 1 the threads are loaded and the calculation of progress begins. Contention for resources is very high in this placement, and so the scheduler moves the threads into a different mapping at time 2, so threads begin making better progress. This configuration remains in place until time 103 when Thread A terminates and is removed from the machine. The evaluation now continues with three threads.

<table>
<thead>
<tr>
<th>Time</th>
<th>Domain 1</th>
<th>Domain 2</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
<th>Thread D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Core0</td>
<td>Core1</td>
<td>Core0</td>
<td>Core1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>0.98</td>
<td>1.48</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>0.98</td>
<td>2.46</td>
</tr>
<tr>
<td>103</td>
<td>B</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>0.98</td>
<td>100.00</td>
</tr>
<tr>
<td>104</td>
<td>B</td>
<td>-</td>
<td>C</td>
<td>D</td>
<td>0.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Figure 4.4: **Bootstrap Example:** Shows the progress of each thread every clock tick as calculated by the Bootstrap Emulation Module.

Since the time needed to calculate the progress of a thread over a tick is usually much shorter than the length of the tick itself, this methodology allows for a very fast evaluation of various thread schedulers. Typically, we perform evaluations which would take hours to run on a real machine in only seconds. While this example demonstrated how to perform rapid evaluation of an algorithm that aims to reduce resource contention, algorithms with many other goals can be also prototyped and evaluated using AKULA.
4.1.3 The Wrapper Module

While rapid evaluation with the Bootstrapping Module allows for quick evaluation and filtering of initial ideas, eventually the algorithm designer would want to evaluate the algorithm on a real system. AKULA provides the Wrapper Module that significantly simplifies this evaluation.

To use the Wrapper Module the developer would rely on exactly the same code that was written to express the scheduling algorithm for the Bootstrapping Module. The difference is that when this code is given to the Wrapper Module, AKULA would run the algorithm on a real system, using thread affinity bindings. When using the Wrapper Module there is no longer a need for the bootstrap data, but the user must supply to AKULA the real executables that AKULA will launch to evaluate the scheduling algorithms.

In order to use the Wrapper Module the test machine must run Linux and have the performance monitoring perfmon module installed. Additionally, if the developer wants to use power management, cpufrequtils must also be installed.

When AKULA is first loaded onto a machine the Configuration Module will learn the architecture of the machine and will automatically set up the available memory domains and cores in the development environment. This module will also detect if perfmon and cpufrequtils are installed and automatically integrate the available performance counters and frequency settings into the development environment.

The developer needs to supply the code for the scheduling algorithm, such as the one shown in Figure 4.2 and which is no different than the code that would be supplied to the Bootstrapping Module, as well as the workload file. A workload file lists the applications that will be launched and the wall clock time when each application should be launched. The user must supply the path to the executable for every application that AKULA must launch.

In order to perform the scheduler evaluation the Wrapper Module enters the following loop:

1. Determine if any threads completed and supply their runtime statistics to the Statistics Module.

2. Update the thread counters as specified by the scheduler.

3. Check the workload file to see if any threads are ready to be launched; if so, create run
directories and launch the threads; call the scheduler to allow it to assign the threads to cores.

4. Call the scheduler to allow it to modify the current thread placement.

5. Sleep until the next scheduling interval expires.

The actual implementation of the Wrapper Module relies on daemons (not Java code) which interact with the OS to perform the tasks requested by the scheduler and shared files to transfer data and instructions. New applications are launched with the AKULA API by calling `threadX.activate()` at which point the wrapper spawns a new daemon that will create a separate run directory for the application, copy its input files (if any), and launch it on the machine returning its process id (pid) and other details to the Wrapper Module. These details will be hidden from the developer. Instead, an instance of the AKULAThread class representing this new application will be created and handed to the AKULA scheduler via the `launchThread()` function. When the scheduler changes the mappings of threads to cores using the functions provided in the AKULA API, the Wrapper Module will convert these mappings into affinity masks and launch a daemon that will enforce these masks via system calls to the OS. When the AKULA scheduler adds performance counters to threads such as `threadX.addParameter("IPC")` the Wrapper Module translates IPC into the needed `perfmon` counters and launches a daemon to attach the necessary performance monitors.

Whenever the Wrapper Module returns from sleep it first launches daemons to check the state of the machine: i.e., which threads are alive, which cores they are bound to, and to gather the latest performance counters. This data is made available to the scheduler via the simplified AKULA API.

### 4.1.4 Code Availability

The AKULA toolset as well as the source code is available for unrestricted use by the community. It can be found through our website at: [http://synar.cs.sfu.ca/akula](http://synar.cs.sfu.ca/akula). As of this writing the AKULA code has been obtained and is being actively used in more than half a dozen projects at Universities on all the continents with the exception of Antarctica and Australia.
4.2 Case Studies

We show the utility of the AKULA toolset by providing case studies of scheduling algorithms we developed and evaluated with the help of AKULA. Using the Bootstrapping module we were able to evaluate our algorithms using a wide variety of workloads and test the scalability of the algorithms as the number of domains grows well beyond what is currently available. All of these things would have been impossible without the help of AKULA. We also validate the Bootstrapping Module against results produced on a real machine with the Wrapper Module.

4.2.1 The Scheduling Algorithms

The main goal of a contention-aware scheduler is to map threads onto cores of a multicore machine in such a way as to minimize the performance loss that threads experience due to competition for shared resources. Different scheduling algorithms use different strategies for determining which threads should be scheduled close together and which should be scheduled further apart. The schedulers vary in their complexity, the frequency with which they migrate threads to enforce scheduling decisions, and as a result their effectiveness for different kinds of workloads. We explore six different scheduling algorithms on a wide variety of workloads.

As the baseline for our experiment we employ two scheduling algorithms either of which can be typically used as the default scheduling policy in modern operating systems. These algorithms are contention-unaware and we refer to them as naive schedulers. *Naive_Cluster* tries to utilize as few memory domains as possible without causing load imbalance between different cores. For example, if Naive_Cluster were to schedule 2 threads onto the system shown in Figure 4.3 it would place the threads on cores 0 and 1 of one of the domains. The logic behind using policies like Naive_Cluster in the OS is that empty memory domains can be brought into a low power state. The other naive scheduler is *Naive_Spread* which attempts to limit shared resource contention by spreading threads among the memory domains as much as possible. Returning to the previous example of two threads being scheduled onto the machine in Figure 4.3 Naive_Spread would allocate them onto core 0 of the different memory domains. We note that the scheduling solutions found by the naive schedulers depend on the spawning order of threads as opposed to any properties or performance characteristics of these threads. We also note that Naive_Spread and Naive_Cluster are
essentially the same algorithm when the number of threads is equal to or greater than the number of cores. The algorithms only differ in scheduling solutions and performance if the number of threads is less than the number of cores.

The next class of schedulers that we consider are based on the DI scheduler introduced in Section 2.4. DI sorts all the threads on the machine based on their miss rate and pairs applications which are the most dissimilar ensuring that the overall LLC misses are spread out as evenly as possible among all the memory domains. We implement DI as well as a simplified version called Threshold. The threshold algorithm divides all the threads into two categories devils and turtles based on whether their miss rate is above or below the threshold value. The devils are then spread among the memory domains while turtles are placed anywhere (so long as load balance is preserved). We first introduced the terminology devils and turtles in the Animalistic Classification Scheme in Section 2.3.7.

The final class of algorithms that we consider is based on dynamic optimization. The swap algorithm schedules newly spawned threads naively on the machine much like Naive_Spread does but after some period of time when the workload has not changed (called the stability period) it begins to optimize its solution. Swap first picks two memory domains at random and records the average Instructions per Cycle (IPC) of all the threads in these two domains. Swap then picks one thread from each of these domains and swaps them (exchanges the two threads between the memory domains). It then records the new average IPC of the two domains. If the IPC has gone up then the swap is successful; otherwise swap will migrate the two threads back to their original domains. In either case the migration is recorded in a log so as to not be repeated again. The frequency of migrations is controlled by the frequency parameter which can be manually adjusted inside swap. DI_swap combines the DI and the swap scheduling algorithms. Every time the thread population changes on the machine the DI algorithm is used to sort and place threads onto cores. When the workload has stabilized the swap algorithm is activated to try and improve the solution found by DI.

4.2.2 Well Behaved Workloads

We begin our evaluation with the Bootstrapping module by emulating a machine identical to the one on which the bootstrapping data was gathered. The machine consists of two memory domains each with 4 cores where the 4 cores share the L3 cache as well as a NUMA memory bank. This architecture is pictured in Figure 4.5. All workloads which are executed on this machine consist of 8 single threaded applications. All 8 threads are spawned at the
same time \( t = 0 \) and the evaluation continues until all threads have finished. Every time a new thread is spawned or a thread terminates the scheduler is called as well as the scheduler is called every scheduling period which we set to 1 second. When a thread terminates its completion time is recorded and compared to its solo execution time in order to calculate the performance degradation that it experienced Equation 4.1.

The performance of a scheduling algorithm for a given workload is evaluated based on two factors: \textit{average performance} and \textit{worst case performance}. Average performance is calculated as the aggregate of the performance degradations of the 8 threads in the workload. The worst case performance is calculated as the maximum performance degradation of the 8 threads in the workload. While improving average performance is beneficial for the entire system and can lead to energy savings (allowing the machine to powered down sooner) improving worst case performance is useful for QoS and predictable performance.

![Diagram of AMD Architecture](https://via.placeholder.com/150)

**Figure 4.5: AMD Architecture:** The machine on which the bootstrap data was gathered and which was emulated for all experiments. It consists of 8 cores with 4 cores per memory domain. Each memory domain also features its own NUMA memory bank offering excellent inter-memory domain isolation.

Given the nature of the DI and Threshold algorithms which are designed to separate
devils and turtles we begin our exploration by creating workloads which consist of only devils and turtles we call these workloads “well behaved”. Applications are selected from the SPEC CPU2006 benchmark suite which can be clearly identified as devils (high miss rate and contention sensitive) or turtles (low miss rate and contention insensitive). Nine categories of workloads are created which differ based on the ratio of devils to turtles. Table 4.3 summarizes the workload categories. For each category we create 100 workloads. The workloads are created by randomly selecting which devils and turtles will be included in the workload as well as randomizing the spawning order of the threads.

<table>
<thead>
<tr>
<th>Category Name</th>
<th>No. Devils</th>
<th>No. Turtles</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0_T8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>D1_T7</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>D2_T6</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>D3_T5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>D4_T4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>D5_T3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>D6_T2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>D7_T1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>D8_T0</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

Each workload is then evaluated using the AKULA Bootstrapping module for each of the 6 scheduling algorithms. Results are reported as averages of the 100 workloads in a given category. Due to the sheer number of results we found it necessary to focus on the workload categories which offer the biggest potential speedup and hence are the most interesting. We evaluate the potential speedup of a category by looking at the variance of the results produced by the 6 schedulers on workload within that category. Categories with high variance mean that some algorithms were able to do significantly better than others and hence there is potential for speedup. Low variance means that all algorithms performed similarly. The categories which offer the largest speedups are those with a fair mix of devils and turtles and these are highlighted in Table 4.3. All results will be reported only for these four categories.

Figure 4.6 and Figure 4.7 show the average and worst case performance for our six algorithms respectively. The results are normalized to the best performing algorithm in each category.

The results for these experiments indicate that the contention aware algorithms that
we explored can significantly improve performance, especially worst case performance, as compared to a naive scheduler. We also note that in terms of the average performance naive_spread appears to perform almost as well as threshold and DI. This is mostly due to the fact that applications have varied run times and as such the workload soon shrinks to a level where the number of threads is less than the number of cores. In these experiments we do not restart threads after termination. If on the other hand the load was kept high naive_spread would be identical to naive_cluster which shows very poor performance. We also note that although swap and DI_swap offer the superior performance in this study the gains that they deliver over DI and threshold are small especially in terms of average performance (about 1%). Thus, we may conclude that in the case of “well behaved” workloads the 4 smart schedulers: threshold, DI, swap, and DI_swap are roughly equivalent.

Figure 4.6: **Average Performance Degradation “Well Behaved” Workloads:** Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).
Figure 4.7: **Worst Case Performance Degradation “Well Behaved” Workloads:** Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).

4.2.3 **Badly Behaved Workloads**

In real life not all applications fit nicely into the categories of devils and turtles. Some applications have an “intermediate” miss rate and less predictable behavior in terms of performance degradation. We call this class of applications *semi-devils*. We evaluate how our algorithms handle workloads which include a varied number of semi-devils. Table 4.4 shows the four workload categories that we considered and the number of threads of each type. Once again 100 workloads make up every category.

<table>
<thead>
<tr>
<th>Category Name</th>
<th>No. Devils</th>
<th>No. Semi-Devils</th>
<th>No. Turtles</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0_S8_T0</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>D1_S6_T1</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>D2_S4_T2</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>D3_S2_T3</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Once again all workloads were executed with all 6 schedulers and for each scheduler the per category averages were obtained. Analysis of potential speedup allows us to exclude the
workload category D0_S8_T0.

Figure 4.8 and Figure 4.9 show the average performance degradations and the worst case performance degradations for workloads which include semi-devils. As we can see when the workload includes semi-devils which can exhibit unpredictable behavior that is not proportional to their miss rate, algorithms like DI and Threshold are no longer sufficient. Dynamically optimizing algorithms like swap are necessary to find the best solutions. This tells us that when “badly behaved” applications are involved whose performance cannot be easily predicted a trial and error method needs to be employed to discover their behavior online.

![Figure 4.8: Average Performance Degradation “Badly Behaved” Workloads: Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).](image)

4.2.4 Scalability of Algorithms

Having determined in the previous section that online optimization is necessary in order to achieve good performance when the workload is “badly behaved” we wish to determine how online optimization scales with an increasing number of cores and threads. Algorithms
like DI and Threshold are fully centralized and are invoked whenever the thread population changes. As such, although these algorithms may become computationally inefficient as the number of cores and threads increases they will not become less effective on a bigger machine. On the other hand, algorithms like swap and DI_swap do optimizations every predetermined time period; if the number of threads and cores increases the number of optimizations performed per unit time will remain unchanged. This means that the number of possible configurations that these algorithms explore remains constant. As the number of cores and threads grows the number of all possible configurations grows rapidly and algorithms which consider only a fixed number of these configurations will become less effective.

To evaluate how our algorithms scale to highly multicore machines we use the Bootstrapping module to simulate a machine with 128 cores which are divided in 32 memory domains of 4 cores each. Figure 4.10 shows conceptually what such a machine would look like. We generate workloads with 128 threads in each. Similarly to the previous section we create seven workload categories which have different ratios of devils, turtles, and semi-devils. Table 4.5 summarizes the workload categories used. For each category we create and run 100 different workloads with each of our algorithms and report the average results. We also note that since threshold and DI produce such similar results and this is a study

---

**Figure 4.9:** Worst Case Performance Degradation “Badly Behaved” Workloads: Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).
focusing on online optimization we exclude threshold from all subsequent results.

Figure 4.10: **Scalability machine architecture:** The machine which was emulated to test the scalability of algorithms. It consists of 32 memory banks with 4 cores each and a per domain NUMA memory bank.

Figure 4.11 shows the worst case performance degradation for all the workloads. These results indicate that for a highly multicore machine the swap algorithm is less effective than the DI algorithm. This point is best highlighted by looking at the workload category D48_S32_T48. Swap performs on average 30% worse than DI for workloads of this category. Looking back at the small machine, Figure 4.9 we see that for the category D4_S2_T4 which has the same ratios of devils, turtles, and semi-devils as D48_S32_T48, swap was BETTER than DI by nearly 50%. The relative performance of DI and swap switched by almost 80% for the same kinds of workloads when the machine increased in size. This suggests that the effect described above where online optimizing algorithms become less effective as the number of cores increases is a reality. On a positive note, we see that the combination of DI and swap, DI\_swap, which contains the best of both worlds, is superior for all workload categories.

The centralized nature of DI and DI\_swap will most certainly result in large overheads as
the size of the machine increases so it would still be advantageous to be able to make the fully
decentralized swap algorithm scalable with the number of cores. To this end we experiment
with a variant of swap called swap-X, where X stands for the number of potential swaps in
a given time period. Every swap-interval swap-X exchanges up to X randomly chosen pairs
of threads, checks the resultant IPC, and undoes the exchanges that did not yield better
performance. The exact algorithm for swap-X relies on well known distributed computing
implementations and is not discussed further in this work.

Figure 4.12 shows the worst-case performance degradation averaged over all 700 work-
loads, from Table 4.5 for swap-X with X ranging from 1 to 16. Swap-1 is the original swap
algorithm which considers only 1 pair of threads at a time. Swap-16 can consider up to 16
pairs of threads every swap period. Since the threads being considered for a swap must ALL
come from different memory domains, swap-16 is the most aggressive version of swap-X pos-
sible for the machine in question. Figure 4.12 indicates that as X increases swap-X obtains
better performance. However, we also see that the biggest performance jump occurs be-
tween swap-1 and swap-2. This indicates that swap-2 would be the ideal swap algorithm for
the machine in questions since it obtains results nearly identical to more aggressive versions
of swap but performs significantly fewer migrations resulting in less overhead.

4.2.5 Validating the Bootstrapping Module and Data

The methodology that we used for gathering bootstrapping data is the short run methodology
where each combination of threads is executed for one minute and the degradation in IPC
of every application is measured. Given such a coarse method of gathering bootstrapping
data as well as the generally coarse evaluation technique employed by the Bootstrapping
Figure 4.11: Worst Case Performance Degradation on 128-core machine: Normalized to the best performing algorithm in each category. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).

Module it would be unreasonable to expect that per thread execution times could be accurately predicted. Instead we focus on validating that given a workload and two different scheduling algorithms that Bootstrapping Module can accurately predict which algorithm will perform better for that workload on average. To that end we selected 10 workloads from Sections 4.2.2 and 4.2.3 at random. For each workload we selected a pair of scheduling algorithms from the six described in Section 4.2.1 also at random. We executed the workload with the two different algorithms using both the Bootstrapping Module and the Wrapper Module (on a real machine). We obtained the difference in average degradation between the two scheduling algorithms on a real machine and compared it to the same difference obtained by the Bootstrapping Module. We found that for all workloads and algorithms tested if the difference in performance on a real machine of the two algorithms is larger than 5% then the Bootstrapping Module will correctly predict which of the two algorithms performs better. Since our goal is to predict which scheduling algorithms will perform better for different workloads these results show that the bootstrapping technique is sufficiently accurate as a first step evaluation tool.
Figure 4.12: **Swap-X Performance:** Worst case performance degradation on 128-core machine for swap-X. Unlike previous graphs in this section the data here is NOT normalized to the best performing run. Each data point on the graph represents the median of no less than 3 identical trials with a standard deviation of no larger than 5% of the median. (Low bars are good).

### 4.3 Other Performance Emulation Tools

Multiple computer simulation projects have been proposed in the past. Perhaps the most popular among them are: Simics [61], an accurate functional full system simulator, SimpleScalar [2], which simulates the work of an out-of-order processor, and the PowerPC architecture simulator called Turandot [66]. All of these projects, while being very useful for modeling the specific computing hardware, are nevertheless not ideally suited for evaluating the OS scheduling algorithms. Cycle-accurate simulations are very time consuming, and they model details, which are beyond what is needed for the evaluation of a scheduling policy. To make the evaluation of the schedulers as fast as possible, we instead focus only on the aspects of system behavior that matter in making a scheduling decision: a coarse grained configuration of the system (cores, shared caches, memory nodes), the workload distribution across the cores at any moment in time, and the degree of performance degradation that the applications will experience in a particular placement.
CHAPTER 4. TOOLSET FOR CREATING CONTENTION-AWARE SCHEDULERS

More theoretical work on scheduling solutions for multicore processors in the presence of contention has also been extensively explored. Most notably Jiang et al. [41, 94] have proposed methodologies for finding the optimal scheduling solution given a workload and a multicore machine. A subset of which we used in section 2.3.1 to devise an optimal solution against which to compare various algorithms. We view AKULA as complementary to this work as it is an actual toolset which allows developers to create and evaluate their scheduling algorithms. Although, AKULA does not facilitate finding optimal solutions it allows for a much wider exploration of potential workloads than [41, 94] by allowing multiphase applications, thread spawning at arbitrary times, and more threads than cores. Furthermore, AKULA allows the same scheduling algorithm to be evaluated on a real machine.

Calandrino et al. proposed LinSched, a user-level simulator for the Linux kernel scheduler [13]. The tool runs as a user-space program and allows specifying simulated system configuration and priorities for the simulated workload. LinSched implements the default Linux scheduler which tries to balance runqueue lengths on different cores. AKULA, on the other hand, focuses on evaluating thread placement policies that handle shared resource contention. Since these policies can be used along with load balancing policies already implemented inside Linux kernel, we see LinSched [13] as work which is complementary to ours.

4.4 Summary

We introduced the AKULA toolset which is designed to help developers create and test contention-aware scheduling algorithms for multicore machines. The AKULA API allows easy development of scheduling algorithms. The bootstrapping module and the bootstrapping evaluation technique facilitate a preliminary evaluation of the developed scheduling algorithm in a fraction of the time that the same experiments would take on a real machine. AKULA also includes the Wrapper Module which allows scheduling algorithms written with the AKULA API to manage threads on a real machine. The intended use of the toolset is to quickly and easily explore the vast search space of scheduling algorithms. The bootstrapping module is designed to detect and filter out bad solutions with minimal time invested on them. The wrapper module is designed to be applied to promising solutions to validate them on real hardware. Those scheduling algorithms which satisfy performance goals when implemented and tested with AKULA can be implemented inside the kernel with the
knowledge that the time and effort invested in kernel programming is worthwhile since these solutions work!

We demonstrated the utility of the AKULA toolset by implementing and evaluating several contention-aware scheduling algorithms using AKULA. We focused on three types of scheduling algorithms: those that schedule naively and are contention unaware, DI like algorithms that separate threads with high miss rates, and those that dynamically optimize by trial and error. We found that for workloads with “well behaved” applications (those whose performance properties can be predicted from their miss rates) both types of contention aware algorithms perform equally well and are superior to the naive algorithms. If the workloads contain “badly behaved” applications the algorithms that rely on miss rate perform significantly worse than the dynamically optimizing ones though they are still better than the naive algorithms. We evaluated how an algorithm that relies solely on dynamic optimization, called swap, would perform on a massively multicore machine and showed that it does not scale well and that its performance will drop significantly as the number of cores increases. We showed, however, that swap can be made much more scalable if the number of applications that are exchanged during every swap phase increases. For a machine with 128 cores exchanging two pairs of threads every swap-phase achieves nearly optimal performance. We also showed that a hybrid algorithm consisting of the miss rate based algorithm DI and the dynamically optimizing algorithm swap called DI_swap, will perform exceptionally well in all scenarios.

The results described above were obtained using the bootstrapping module; an evaluation of this magnitude would not have been possible without it. We also validated that results obtained using the bootstrapping technique translate into similar results if tested on a real machine.
Chapter 5

Summary of Contributions

In this section we summarize the contributions of this thesis to the field. The focus of this thesis was to explore mitigating shared resource contention in Chip Multicore Processors by means of a thread-level-scheduler.

The first contribution is a methodical study of the causes of shared resource contention in the CMP. This was particularly important since there is a significant volume of literature which focused on only one specific cause of contention such as the Last Level Cache or the DRAM controller. Since different papers focused on different issues exclusively it was not clear which of these was the main culprit and should be the main focus of a solution. With our unique methodology we broke down the contributions from the different points of contention in the CMP and we showed that there was no one single dominant factor. That all the points of contention: contention for the shared LLC, contention for the memory bus, contention for the DRAM controller, and contention for prefetching hardware all played an important role. Therefore, a comprehensive solution must address all these factors.

The second contribution is a study of different thread classification schemes. A thread classification scheme distinguishes threads from each other based on their microarchitectural properties and is needed by the scheduler to make decisions about which threads should and should not be scheduled together. We devised a novel methodology to evaluate the effectiveness of classification schemes in predicting shared resource contention. We also proposed our own classification schemes Pain and Miss Rate based on the study we conducted earlier to understand the points of contention in the CMP. We showed that our proposed classification schemes were superior to the others explored and that they were near the theoretical optimal.
CHAPTER 5. SUMMARY OF CONTRIBUTIONS

The third contribution is the creation of the Distributed Intensity (DI) contention-aware scheduler based on the highly effective Miss Rate classification scheme and our Centralized Sort algorithm. The DI scheduler was extensively validated on three different real architectures with a variety of workloads. We showed that it consistently outperformed the default contention-unaware Linux scheduler on average. We also demonstrated that the real strength of the DI scheduler was in avoiding the worst-case performance for individual threads; an essential first step in providing QoS in light of shared resource contention.

The fourth contribution is the AKULA toolset for developing contention-aware schedulers. We argue that the design space of contention-aware schedulers is enormous and that DI is by no means the ultimate solution but rather a data good point within this space. The AKULA toolset provides developers with a more tractable exploration of this vast design space. Comprised of an API which allows for simple translation of ideas into scheduling code as well as a novel performance emulation technique called the Bootstrap Module the AKULA toolset allows developers to create and rapidly evaluate scheduling algorithms. Quickly filtering out failed ideas without needless invest of time and effort.

The fifth contribution is an exploration, with the help of the AKULA toolset, of the limitations of DI. We show that when faced with workloads consisting of threads with difficult to determine/categorize properties the DI scheduler begins to perform poorly. We present a trial-and-error based technique which is able to overcome the limitations of DI and perform extremely well with any workload. We further explore the scalability of DI and this new technique to futuristic massively multicore architectures. We demonstrate the scalability issues that arise and propose extensions to the scheduling algorithms which overcome these issues. Most significantly the merging of DI and the trial-and-error scheduler into a single scheduler called DILSwap that performed well in all considered scenarios.
Bibliography


[77] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In ISCA ’00: Proceedings of the 27th annual international


