AKULA: A Toolset for Experimenting and Developing Thread Placement Algorithms on Multicore Systems

Sergey Zhuravlev
School of Computing Science
Simon Fraser University
Vancouver, Canada
sergey_zhuravlev@sfu.ca

Sergey Blagodurov
School of Computing Science
Simon Fraser University
Vancouver, Canada
sergey_blagodurov@sfu.ca

Alexandra Fedorova
School of Computing Science
Simon Fraser University
Vancouver, Canada
fedorova@sfu.ca

ABSTRACT

Multicore processors have become commonplace in both desktop and servers. A serious challenge with multicore processors is that cores share on and off chip resources such as caches, memory buses, and memory controllers. Competition for these shared resources between threads running on different cores can result in severe and unpredictable performance degradations. It has been shown in previous work that the OS scheduler can be made shared-resource-aware and can greatly reduce the negative effects of resource contention. The search space of potential scheduling algorithms is huge considering the diversity of available multicore architectures, an almost infinite set of potential workloads, and a variety of conflicting performance goals. We believe the two biggest obstacles to developing new scheduling algorithms are the difficulty of implementation and the duration of testing. We address both of these challenges with our toolset AKULA which we introduce in this paper. AKULA provides an API that allows developers to implement and debug scheduling algorithms easily and quickly without the need to modify the kernel or use system calls. AKULA also provides a rapid evaluation module, based on a novel evaluation technique also introduced in this paper, which allows the created scheduling algorithm to be tested on a wide variety of workloads in just a fraction of the time testing on real hardware would take. AKULA also facilitates running scheduling algorithms created with its API on real machines without the need for additional modifications. We use AKULA to develop and evaluate a variety of different contention-aware scheduling algorithms. We use the rapid evaluation module to test our algorithms on thousands of workloads and assess their scalability to futuristic massively multicore machines.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling

General Terms

Performance, Measurement, Algorithms

Keywords

Contention-aware scheduling, Multicore simulation

1. INTRODUCTION

One of the key challenges of multicore systems is minimizing contention for shared computing resources, such as caches, memory controllers, and interconnects. Modern multicore systems consist of multiple *memory-domains*, which are clusters of cores sharing computing resources (Figure 1). On such architectures, intelligent placement of threads into memory domains becomes crucial, especially for CPU-bound workloads, which are particularly sensitive to variations in CPU performance. Given T threads and M memory domains, 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. Our experiments show that a policy that avoids contention may improve performance by as much as 50% for some applications. In this context thread placement refers to an instance of the scheduling problem where a scheduler decides how to *space share* the hardware among threads. In other words, the threads which are to be run on the machine are assigned, i.e., mapped, to specific cores. In the event that the number of threads exceeds the number of cores, some cores will also be time shared by multiple threads. We will use the terms *thread placement* and *thread schedule* interchangeably. 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. We use characteristic to mean micro-architectural properties that determine how the threads will compete for shared resources if scheduled to the same memory domain. Moreover, thread characteristics may not be 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

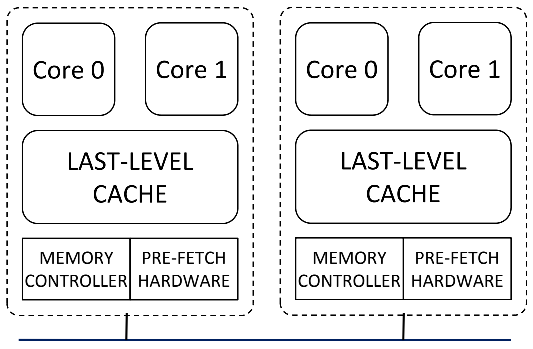

Figure 1: A schematic view of a system with two memory domains and two cores per domain. Cores in each domain share the last-level (L2) cache, the memory controller, and the pre-fetching hardware.

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 [4, 8, 25] points to the importance of finding good scheduling algorithms for CMPs.

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 between 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*.

The need for the AKULA toolset is motivated by a renewed interest in research in scheduling algorithms for multicore systems. Since scheduling algorithms were shown to be extremely effective in improving system performance by controlling resource allocation, dozens of new scheduling algorithms have emerged [25, 4, 11, 17, 1, 2, 10, 19, 22, 20, 16, 9, 15, 5, 7, 13, 24]. Commercial operating systems, such as Solaris and Linux, have also made recent updates to their schedulers. In designing AKULA, we hoped to provide a powerful tool that would aid research in this highly active and fruitful area.

Using the AKULA toolset we discovered interesting problems with existing contention-aware scheduling algorithms, and found effective solutions. For example, we found that existing algorithms, such as those proposed in [4, 11, 25], were 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 algorithms failed to effectively reduce contention. Using AKULA we were able to quickly design an algorithm that works effectively for a wider range of workloads. Using AKULA we also confirmed, using a very large number of experiments, earlier findings that while contention-aware algorithms produce modest performance improvements on average for all applications in a given workload [25], they are able to significantly reduce the worst-case execution time by preventing pathological thread placements that can be chosen by a contention-unaware scheduler. We also explored the scalability of these algorithms to futuristic massively multicore systems.

The rest of the paper is structured as follows. The AKULA toolset is discussed in detail in Section 2. We showcase AKULA's utility in Section 3 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 discusses related work, and Section 5 summarizes our findings.

2. 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 onto 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 algorithms or move on to a completely different idea without having wasted significant time or effort on this first algorithm.

Figure 2 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.

To make evaluations easier, the statistics from each ex-

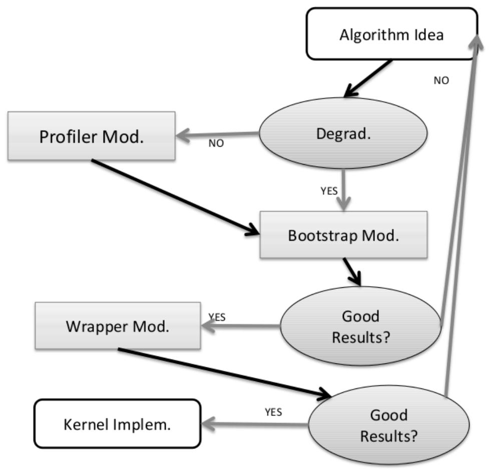

Figure 2: Using AKULA.

```
public class naive_spread {

        private int previousPop = 0;

        public void launchThread(thread readyThread, machine M, int current_time){
                readyThread.activate(current_time);
                M.addThread(readyThread);
                previousPop += 1;
        }

        public void updateSchedule(machine M, int currentTime){
                if(M.totalLoad() != previousPop){
                        M.balanceLoad();
                        M.balanceDomains();
                        previousPop = M.totalLoad();
                }
        }
}
```

Figure 3: The implementation of *Naive_Spread* algorithm with AKULA

periment are output in tabular form, which can be imported into popular statistical packages, using the *Statistics Module*.

2.1 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 a 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 [23].

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 3 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.

2.2 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* relative performance of different scheduling algorithm 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. On machines with a large number of cores in a memory domain, the process of bootstrap data collection can be especially time consuming, because many thread combinations must evaluated. To overcome this limitation, we are developing a method that would use very short representative periods from the execution of each thread to test it in each co-schedule. This method would significantly reduce the time needed to collect the bootstrap data. We omit a more detailed description of this method due to space constraints.

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 1. 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 1 and Table 2 show the necessary bootstrap data to perform this evaluation. Table 1 gives the execution times of each of the threads when they run alone on a memory domain without any contention. Table 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. The entry at row B and column A shows that in this scenario thread B executes at 0.6 of the speed it has when running alone on a memory domain.

	Solo Exec. (s)
A	100
B	150
C	175
D	200

Table 1: Solo Execution Times

	alone	with A	with B	with C	with D
A	1.00	0.75	0.50	0.98	0.99
B	1.00	0.60	0.30	0.95	0.97
C	1.00	0.99	0.98	1.00	1.00
D	1.00	1.00	1.00	1.00	1.00

Table 2: 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.

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 the example provided we specify the workload to consist of four single-phased threads A, B, C and D, whose bootstrap data is described in Table 1 and Table 2, set to be launched at time *t = 1 second.*

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 1.

$$Progress(X) = 100\% * \frac{tick}{Solo(X)} * deg(X, Neighb(X)) \quad (1)$$

$Progress(X)$ refers to the fraction of total work that a thread completes in a given scheduling interval. $Tick$ refers to the length of the scheduling time interval. $Solo(X)$ is the thread's solo completion time obtained from the bootstrap data and $deg(X, Neighb(X))$ 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 "neighbours" impose on the target thread.

In our example we set the length of the tick to be one second. Table 1 and Table 2 are the bootstrap data used in this evaluation.

Figure 4 shows the results of the evaluation step by step. The progress is calculated for each thread at each tick using Eq. 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.

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.

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. The great speed with which experiments can be performed once bootstrapping data has been gathered acts as a direct counterweight to the time needed to gather this data in the first place. Gathering bootstrapping data for a particular machine and benchmark set needs to be done only once and then an unlimited number of experiments can be done with this data. To further mitigate the *research overhead* of gathering this data we make all the bootstrap data that we have ever gathered freely available in our data repository which can be found at: *http://synar.cs.sfu.ca/akula*. We also hope that other researchers who will use AKULA in the future will contribute their data to this repository.

2.3 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 the 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. A port of AKULA to other operating systems is planned for the future. 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 3 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, 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*. To save time or memory during experiments the module can be configured to simply its launch procedures such as foregoing the data copying. 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

	Domain 1		Domain 2		Thread A		Thread B		Thread C		Thread D	
Time	Core0	Core1	Core0	Core1	PR.	Total PR.	PR.	Total PR.	PR.	Total PR.	PR.	Total PR.
0	-	-	-	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1	A	B	C	D	0.50	0.50	0.40	0.40	0.57	0.57	0.50	0.50
2	B	D	A	C	0.98	1.48	0.65	1.05	0.57	1.14	0.50	1.00
3	B	D	A	C	0.98	2.46	0.65	1.69	0.57	1.70	0.50	1.50
103	B	D	A	C	0.98	100.00	0.65	66.36	0.57	58.42	0.50	51.50
104	B	-	C	D	0.00	100.00	0.67	67.03	0.57	58.99	0.50	52.00

Figure 4: Step by step example of using the bootstrap module

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.

2.4 Code Availability

The AKULA source code is freely available to the research community under a General Public License and can be found through our website at: *http://synar.cs.sfu.ca/akula*

3. 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.

3.1 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 1 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 1, 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. Since we evaluate workloads consisting of threads with varied runtimes we encounter lengthy periods where the number of threads is less than the number of cores which makes the inclusion of both algorithms interesting.

The next class of schedulers that we consider is based on research by [25] which shows that threads with high last level cache miss rates should be scheduled far apart. In [25] the authors demonstrate that applications with a high LLC miss rate, which they call *devils*, aggressively compete for the entire memory hierarchy and as a result suffer as well as induce a high performance penalty when scheduled “near” other devils. Applications with a low LLC miss rate, which they call *turtles*, do not compete for shared resources and as such do not cause or experience performance degradation regardless of how they are scheduled. The contention-aware scheduler proposed in [25] called *Distributed Intensity (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 on any remaining cores (so long as load balance is preserved).

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 2 memory domains at random and records the average *Instructions per Cycle (IPC)* of all the threads in these 2 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.

3.2 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 machine is especially well suited for the bootstrapping experimental methodology since threads running on different memory domains (so long as their memory is also placed in different NUMA banks) have negligible effect on each other's performance. Therefore the implicit assumption made by the bootstrapping methodology that contention stops at the memory domain level holds. 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 Eq. 2.

$$Perf\_Degrad = 100\% * \frac{WorkloadTime - SoloTime}{SoloTime} \quad (2)$$

The performance of a scheduling algorithm for a given workload is evaluated based on two factors: *average performance* and *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

Category Name	# Devils	# Turtles
D0_T8	0	8
D1_T7	1	7
D2_T6	2	6
D3_T5	3	5
D4_T4	4	4
D5_T3	5	3
D6_T2	6	2
D7_T1	7	1
D8_T0	8	0

Table 3: "Well Behaved" Workloads

Category Name	# Devils	# Semi-Devils	# Turtles
D0_S8_T0	0	8	0
D1_S6_T1	1	6	1
D2_S4_T2	2	4	2
D3_S2_T3	3	2	3

Table 4: "Badly Behaved" Workloads

sooner) improving worst case performance is useful for QoS and predictable performance.

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 their ratio of devils to turtles. Table 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.

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 workloads within that category. Having high variance mean that some algorithms were able to do significantly better than others and hence there is potential for speedup in this category. 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 3. All results will be reported only for these four categories.

Figure 5 and Figure 6 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

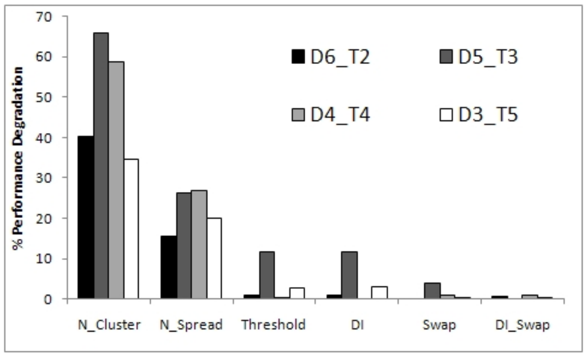

Figure 5: Worst Case Performance Degradation "Well Behaved" Workloads. Normalized to the best performing algorithm in each category

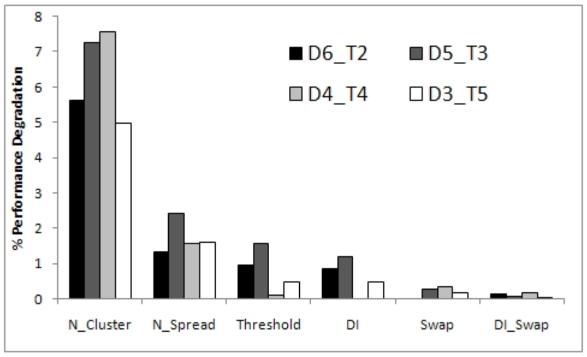

Figure 6: Average Performance Degradation "Well Behaved" Workloads. Normalized to the best performing algorithm in each category

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. 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.

3.3 Semi-Devil Workloads

Scheduling algorithms like DI and Threshold were designed specifically to handle applications which nicely fall into two categories (devils and turtles). However, in real life not all applications fit into these boxes. 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 shows the for workload categories that we considered and the number of threads of each type. Once again 100 workloads make up every category.

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

Figure 7 and Figure 8 show the worst case performance degradations and average 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.

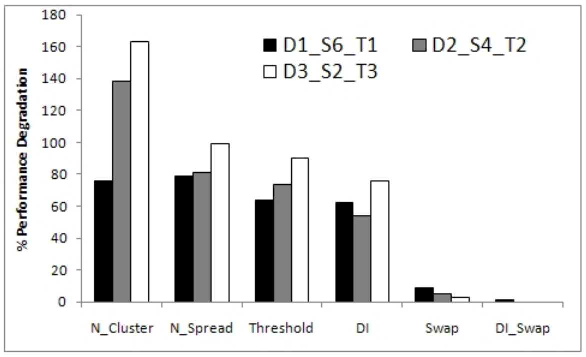

Figure 7: Worst Case Performance Degradation "Badly Behaved" Workloads. Normalized to the best performing algorithm in each category

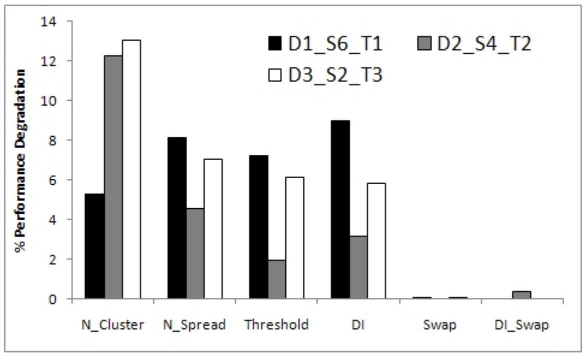

Figure 8: Average Performance Degradation "Badly Behaved" Workloads. Normalized to the best performing algorithm in each category

3.4 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

Category Name	# Devils	# Semi-Devils	# Turtles
D128_T0	128	0	0
D96_T32	96	0	32
D64_T64	64	0	64
D32_T96	32	0	96
D16_S96_T16	16	96	16
D32_S64_T32	32	64	32
D48_S32_T48	48	32	48

Table 5: Workloads for the 128 core machine

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. 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 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 focusing on online optimization we exclude threshold from all subsequent results.

Figure 9 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 7, 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 kind 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 10 shows the worst-case performance degradation averaged over all 700 workloads, from Table 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 possible for the machine in question. Figure 10 indicates that as X increases swap-X obtains better performance. However, we also see that the biggest performance jump occurs between 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.

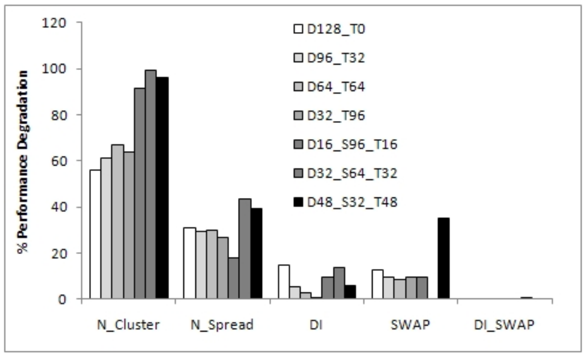

Figure 9: Worst Case Performance Degradation on 128-core machine. Normalized to the best performing algorithm in each category

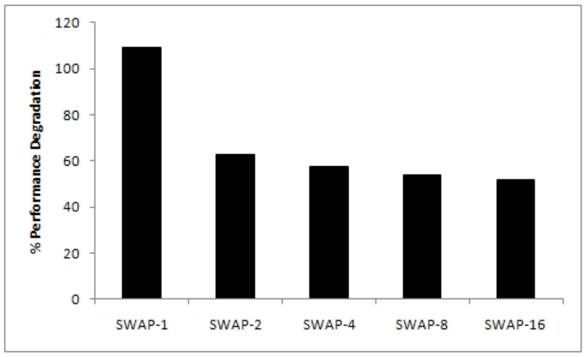

Figure 10: Worst Case Performance Degradation on 128-core machine for swap-X. Not normalized

3.5 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 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 3.2 and 3.3 at random. For each workload we selected a pair of scheduling algorithms from the six described in Section 3.1 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.

4. RELATED WORK

Multiple computer simulation projects have been proposed in the past. Perhaps the most popular among them are: Simics [12], an accurate functional full system simulator, SimpleScalar [18], which simulates the work of an out-of-order processor, and the PowerPC architecture simulator called Turandot [14]. 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.

More theoretical work on scheduling solutions for multicore processors in the presence of contention has also been extensively explored. Most notably [6, 21] have proposed methodologies for finding the optimal scheduling solution given a workload and a multicore machine. 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 [6, 21] 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 [3]. 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 [3] as the work which is complimentary to ours.

The idea of distributing programs with high miss rates across the memory domains was suggested in several prior studies [25, 8, 4]. Our work is broader in a sense that we explore the limitations of these algorithms 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.

5. CONCLUSIONS

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! After all a kernel implementation more efficient than the Wrapper Module implementation and should therefore deliver even better performance than was measured by AKULA.

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, those 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.

6. ACKNOWLEDGMENTS

The authors thank the National Science and Engineering Council of Canada for supporting this research.

7. REFERENCES

[1] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *SIGARCH CAN*, 33(2):506–517, 2005.

[2] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proc. of Computing Frontiers '06*, 2006.

[3] J. M. Calandrino, D. P. Baumberger, T. Li, J. C. Young, and S. Hahn. LinSched: The Linux Scheduler Simulator. In *ISCA PDCCS*, 2008.

[4] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: a System for Energy Efficient Computing in Virtualized Environments. In *ISLPED*, 2009.

[5] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *CF '05: Proceedings of the 2nd Conference on Computing Frontiers*, pages 199–210, 2005.

[6] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229, New York, NY, USA, 2008. ACM.

[7] V. Kazempour, A. Kamali, and A. Fedorova. AASH: An Asymmetry-Aware Scheduler for Hypervisors. In *Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2010.

[8] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, 2008.

[9] D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multicore Architectures. In *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys)*, 2010.

[10] R. Kumar, D. M. Tullsen, and P. Ranganathan et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. of ISCA '04*.

[11] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-core Architectures. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, 2007.

[12] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35, 2002.

[13] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys)*, 2010.

[14] M. Moudgill, P. Bose, and J. H. Moreno. Validation of Turandot, a Fast Processor Model for Microarchitecture Exploration. In *IPCCC*, pages 451–457, 1999.

[15] J. Saez, A. Fedorova, M. Prieto, and H. Vegas. Operating System Support for Mitigating Software Scalability Bottlenecks on Asymmetric Multicore Processors. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, 2010.

[16] J. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Processors. In *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys)*, 2010.

[17] D. Shelepov, J. C. Saez, and S. Jeffery et al. HASS: a Scheduler for Heterogeneous Multicore Systems. *ACM Operating System Review*, 43(2), 2009.

[18] C. I. Simplescalar, D. Burger, and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, 1997.

[19] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *ASPLOS 2000*, 2000.

[20] R. Teodorescu and J. Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In *Proc. of ISCA '08*, 2008.

[21] K. Tian, Y. Jiang, and X. Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 41–50, New York, NY, USA, 2009. ACM.

[22] J. A. Winter and D. H. Albonesi. Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures. In *Proc. of DSN '08*, pages 42–51, 2008.

[23] X. Zhang, S. Dwarkadas, and K. Shen. Hardware Execution Throttling for Multi-core Resource Management. In *USENIX Annual Technical Conference*, 2009.

[24] S. Zhuravlev and S. Blagodurov. Fast simulations of 1000 core system. In *SOSP 2009 Posters and Work In Progress*, 2009.

[25] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *ASPLOS*, 2010.