Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors

SERGEY ZHURAVLEV
Simon Fraser University
JUAN CARLOS SAEZ
Complutense University of Madrid
SERGEY BLAGODUROV
Simon Fraser University
ALEXANDRA FEDOROVA
Simon Fraser University
and
MANUEL PRIETO
Complutense University of Madrid

Chip multicore processors (CMPs) have emerged as the dominant architecture choice for modern computing platforms and will most likely continue to be dominant well into the foreseeable future. As with any system, CMPs offer a unique set of challenges. Chief among them is the shared resource contention which results because CMP cores are not independent processors but rather share common resources among cores such as the last level cache (LLC). Shared resource contention can lead to severe and unpredictable performance impact on the threads running on the CMP. Conversely, CMPs offer tremendous opportunities for mulithreaded applications which can take advantage of simultaneous thread execution as well as fast inter-thread data sharing. Many solutions have been proposed to deal with the negative aspects of CMPs and take advantage of the positive. This survey focuses on the subset of these solutions which exclusively make use of OS thread-level scheduling to achieve their goals. These solutions are particularly attractive as they require no changes to hardware and minimal or no changes to the OS. The OS scheduler has expanded well beyond its original role of time-multiplexing threads on a single core into a complex and effective resource manager. This paper surveys a multitude of new and exciting work that explores the diverse new roles the OS scheduler can successfully take on.

Categories and Subject Descriptors: D.4.1 [**Process Management**]: Scheduling

General Terms: Performance, Measurement, Algorithms

This research was funded by the National Science and Engineering Research Council of Canada (NSERC) under the Strategic Project Grant program, by the Spanish government's research contracts TIN2008-005089 and the Ingenio 2010 Consolider ESP00C-07-20811 and by the HIPEAC^2 European Network of Excellence.
Authors' addresses: S. Zhuravlev, S. Blagodurov and A. Fedorova, Simon Fraser University, 8888 University Drive, Burnaby, B.C., Canada V5A 1S6; email: {sergey zhuravlev,sergey blagodurov,alexandra fedorova}@sfu.ca. J.C. Saez and M. Prieto, ArTeCS Group, Complutense University of Madrid, Madrid 28040, Spain; email: {jcsaezal,mpmatias}@pdi.ucm.es.

Additional Key Words and Phrases: Survey, Shared Resource Contention, Thread Level Scheduling, Power-aware Scheduling, Thermal Effects, Cooperative Resource Sharing

1. INTRODUCTION

Since the dawn of multiprogramming the scheduler has been an essential part of the operating system. Tasked with distributing the limited CPU time among threads, over the years the scheduler has evolved into a highly efficient and highly effective resource-manager. Significant research efforts, both academic and industrial, have lead to OS schedulers that enforce fairness, respect priority, facilitate real-time applications and ensure that all threads make progress. OS schedulers for single processor architectures had become so optimized that the need for further improvements dramatically subsided, thereby diminishing interest in this research topic. In the late 90's, the scheduling problem was considered solved; at least it appeared that way until the advent and subsequent near ubiquitous proliferation of chip multiprocessors (CMP).

CMP architectures, consisting of multiple processing cores on a single die added new dimensions to the scheduler's role. In addition to multiplexing a single processor in time, e.g., time-sharing it among the threads, it became necessary to also *space-share* cores among the threads. Space-sharing is about deciding on which core each thread chosen to run at a given time interval will be assigned to run. This is a traditional responsibility of the OS scheduler. While on older systems space-sharing was not very important, with the advent of CMPs space sharing took on a very important role.

When CMPs first appeared, they ran unmodified schedulers that were designed for older symmetric multiprocessor (SMP) systems. Each core was exposed to the OS as an isolated processor, and so an SMP scheduler could be used without modifications on CMP systems. For the OS scheduler, this created the illusion that each core in a CMP was an independent processor. This convenient but overly optimistic abstraction caused a lot of problems.

The fact that CMP cores are not fully independent processors but share resources such as caches and memory controllers with neighboring cores results in performance degradation due to competition for shared resources. Some researchers observed that an application can slow down by hundreds of percent if it shares resources with processes running on neighboring cores relative to running alone.

Thread-level schedulers have been shown to be very effective at helping mitigate the performance losses due to shared-resource contention. Different combinations of threads compete for shared resources to different extents and as such suffer different levels of performance loss. Contention-aware schedulers determine which threads are scheduled close together (sharing many resources) and which are scheduled far apart (sharing minimal resources) in such a way as to minimize the negative effects of shared resource contention. Using thread-level schedulers to address shared resource contention is particularly attractive because the solution requires no changes to the hardware and minimal changes to the operating system itself. This is in sharp contrast to orthogonal CMP-contention minimization techniques, such as last-level-cache partitioning and DRAM-controller scheduling, that require

substantial changes to the hardware and/or the OS to enforce physical partitioning of resources among threads. The simplicity of implementation makes a contention-mitigation solution based on thread-level scheduling the most likely solution to be adopted into commercial systems in the near future.

In contrast to the negative aspects of CMPs discussed so far, CMP architectures offer tremendous opportunities for speeding up multi-threaded applications by allowing multiple threads to run at once. While threads from different applications typically compete for shared resources, threads of the same application can share these resources constructively, and actually benefit from sharing. Threads that share data can do so more productively if they also share the same LLC. Similarly such threads can share the prefetching logic and bring data into the cache for each other. Much like in the case of avoiding shared resource contention, facilitating cooperative data sharing can also be handled by a smart thread-level scheduler. Such a solution is attractive and likely to be commercially implemented for the same reason the contention-aware scheduler is.

The rest of the survey is organized as follows. Section 2 focuses on using scheduling to deal with the challenges created by CMP architectures. It describes how scheduling algorithms can be used to detect and avoid thread-to-core mappings that aggravate shared resource contention and hurt performance. Section 3 discusses the use of scheduling to better utilize the opportunities of CMPs specifically using schedulers as a tool to aid cooperate memory sharing among multiple threads of the same application. Section 4 provides an overview of current state-of-the art OS schedulers to show the current scheduling solutions and contrast them with the "smart" schedulers proposed. Section 5 concludes with a discussion of how we see the future of scheduling research.

2. CONTENTION-AWARE SCHEDULING

2.1 Introduction

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 or interconnects, 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. This means that they do not enforce any kind of fairness or partitioning when different threads use the resources. 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.

There has been significant interest in the research community in addressing 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) [Srikantaiah et al. 2009; Qureshi et al. 2006; Gordon-Ross et al. 2007; Chang and Sohi 2007; Kotera et al. 2007; Viana et al. 2008; Jaleel et al. 2008; Srikantaiah et al. 2008; Iyer 2004; Guo and Solihin 2006; Rafique et al. 2006; Liang and Mitra 2008a; Zhao et al. 2007; Shi et al. 2007; Hsu et al. 2006; Suh et al. 2004; Reddy and Petrov 2007; Lin et al. 2009; Kim et al. 2004;

Lin et al. 2008; Cho and Jin 2006a; Lee et al. 2009; Balasubramonian et al. 2000; Kotera et al. 2008; Stone et al. 1992; Chishti et al. 2005; Liu et al. 2004; Liedtke et al. 1997; Xie and Loh 2009; Chandra et al. 2005; Soares et al. 2008; Albonesi 1999; Berg and Hagersten 2004; Liang and Mitra 2008b; Zhang et al. 2009; Qureshi and Patt 2006; Nesbit et al. 2007] or performance-aware DRAM controller memory scheduling [Loh 2008; Suh et al. 2002; Hur and Lin 2004; Nesbit et al. 2006; Rixner et al. 2000; Koukis and Koziris 2005; Burger et al. 1996; Mutlu and Moscibroda 2008; Delaluz et al. 2002; Ipek et al. 2008; Mutlu and Moscibroda 2007; Wang and Wang 2006]. The proposed solutions require changes to the hardware, major changes to the operating system (OS), or both. As such, the majority of these techniques have only been evaluated in simulation and, as of this writing, none of these promising solutions have yet been implemented in commercial systems.

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 [Zhuravlev et al. 2010; Knauerhase et al. 2008; Banikazemi et al. 2008; Jiang et al. 2008; Tian et al. 2009; Merkel et al. 2010]. 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 that lead to the best possible performance.

In this section we provide background on CMP shared-resource contention. We discuss at length the research on mitigating this contention via thread-level scheduling. We then outline several other complementary techniques that are worth mentioning in the context of this survey. Finally, we conclude with a discussion of how thread-scheduling-based solutions fit together in the larger scheme of things and enumerate some possible directions going forward for mitigating shared-resource contention in CMPs.

2.2 Background

One of the clearest points of contention in the CMP is the shared last-level cache (LLC). The effect of simultaneously executing threads competing for space in the LLC has been explored extensively by many different researchers [Srikantaiah et al. 2009; Qureshi et al. 2006; Gordon-Ross et al. 2007; Chang and Sohi 2007; Kotera et al. 2007; Viana et al. 2008; Jaleel et al. 2008; Srikantaiah et al. 2008; Iyer 2004; Guo and Solihin 2006; Rafique et al. 2006; Liang and Mitra 2008a; Zhao et al. 2007; Shi et al. 2007; Hsu et al. 2006; Suh et al. 2004; Reddy and Petrov 2007; Lin et al. 2009; Kim et al. 2004; Lin et al. 2008; Cho and Jin 2006a; Lee et al. 2009; Balasubramonian et al. 2000; Kotera et al. 2008; Stone et al. 1992; Chishti et al. 2005; Liu et al. 2004; Liedtke et al. 1997; Xie and Loh 2009; Chandra et al. 2005; Soares et al. 2008; Albonesi 1999; Berg and Hagersten 2004; Liang and Mitra 2008b; Zhang et al. 2009; Qureshi and Patt 2006; Nesbit et al. 2007].

The most common replacement policy used in caches is Least Recently Used (LRU) [Suh et al. 2004; Kim et al. 2004]. 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 replacement policy treats misses from all competing threads uniformly

and allocates cache resources based on their rate of demand[1] [Jaleel et al. 2008]. As a result, the performance benefit that a thread receives from having greater cache space depends on its memory access pattern and thus varies greatly from thread to thread. Furthermore, 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 [Qureshi and Patt 2006; Suh et al. 2004]. [Kim et al. 2004; Chandra et al. 2005] 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. [Kim et al. 2004] 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. [Chandra et al. 2005] shows an almost four-fold increase in the LLC miss rate for the SPEC benchmark MCF when it shares the LCC as compared to running alone. The corresponding performance of MCF is around 30% of its solo execution.

On Uniform Memory Access (UMA) Architectures with multiple LLCs, the LLCs are usually connected via a shared bus to the DRAM controller. This memory bus is another point of contention for threads running simultaneously on the CMP. [Kondo et al. 2007] 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.

The other crucial shared resource and major source of contention in the CMP is the DRAM controller. The DRAM controller services memory requests that missed in the LLC. Like LRU for caches, existing high-performance DRAM memory controllers were optimized for single-threaded access and these controllers were designed to maximize the overall data throughput [Rixner et al. 2000]. However, they do not take into account the interference between different threads when making scheduling decisions on CMP systems. Therefore, when multiple threads compete for the DRAM controller, these conventional policies can result in unpredictable and poor performance [Mutlu and Moscibroda 2007].

[Nesbit et al. 2006] show 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. [Mutlu and Moscibroda 2007] show the high variability in memory latency that arises for different applications when they share the DRAM controller. The authors demonstrate

[1] Demand is determined by the number of unique cache blocks accessed in a given interval [Denning 1968]

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.

In the last few years major processor manufacturers have gradually abandoned[2] UMA architectures in favor of Non-Uniform Memory Access (NUMA) designs. For example, in the latest Intel and AMD products, processors and other system components such as memory banks and I/O devices, are connected via high-speed point-to-point interconnects. Instead of using a single shared pool of memory connected to all the processors through a shared bus and memory controller hubs, each processor has its own dedicated memory bank that it accesses directly through an on-chip memory controller.

Although NUMA architectures improve scalability by eliminating the competition between processors for bus bandwidth, recent work has demonstrated that the negative effect due to shared-resource contention is still significant on these systems [Zhuravlev et al. 2010; Blagodurov et al. 2011]. The authors show that on a multiprocessor NUMA system consisted of AMD "Barcelona" quad-core processors, an application's completion time when running simultaneously with others can increase by up to a factor of 2.75 compared to running alone. Several other researchers highlight additional challenges that multiprocessor NUMA systems give rise to, such as an increased overhead of inter-processor thread migrations [Li et al. 2007; Blagodurov et al. 2011] and the necessity for effective memory access scheduling algorithms specifically designed for these systems, which include per-chip memory controllers [Kim et al. 2010].

The studies presented so far 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 that 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 instruction retirement rate depends on the other threads in the workload makes performance on CMPs unpredictable which can be an even bigger problem than reduced throughput [Moreto et al. 2009; Guo et al. 2007; Iyer et al. 2007; Nesbit et al. 2007; Iyer 2004]. 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 highly variable. 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 [Mutlu and Moscibroda 2007]. 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.

The problem of shared resource contention and the resultant reduction in through-

[2]Although this is the dominant trend in most desktop and server products, NUMA designs currently coexist with more modest, low-power UMA solutions (e.g., the Intel Atom processor).

put, fairness, and predictability is further complicated by two other issues. First, the three main sources of contention – the LLC, the memory bus, and the DRAM controller – all contribute to the overall performance degradation of a system and a comprehensive solution is needed to address all these factors simultaneously. This is demonstrated by [Zhuravlev et al. 2010] where the authors quantify the percent contribution of each type of shared resource to the overall performance degradation. They show that in the majority of cases there is no single dominant contributing factor and everything plays an important role [Zhuravlev et al. 2010]. Second, these contention factors interact in complex and intertwined ways and hence attempts to deal with only a subset of the contention issues can exacerbate the other contention factors leading to poor performance [Bitirgen et al. 2008].

The research presented above makes it clear that shared resource contention is a serious problem that leads to poor and unpredictable performance. Furthermore, as previous research shows, a comprehensive solution is necessary to deal with this problem. However, the question remains what exactly is the desired outcome of a "good" solution. [Hsu et al. 2006] 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.

Contention-aware thread scheduling is emerging as a promising solution to shared resource contention on CMPs. Several studies [Zhuravlev et al. 2010; Knauerhase et al. 2008; Banikazemi et al. 2008; McGregor et al. 2005; Snavely et al. 2002; Merkel et al. 2010; Fedorova et al. 2007; Jiang et al. 2008; Tian et al. 2009] have shown significant improvement in throughput, fairness, and predictability over contention-unaware schedulers. Moreover, the simplicity of implementation, requiring no changes to the hardware and few changes to the OS[3], makes the solution very attractive for the near term. Conceptually, contention-aware schedulers map threads to cores in such a way as to minimize shared resource contention amongst the threads and improve performance. In this section we discuss the currently proposed contention-aware schedulers. We begin by explicitly stating the assumptions that contention-aware schedulers often take for granted. We then outline the four building blocks common to all contention-aware schedulers.

2.3 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. One notable exception is the work by [Merkel et al. 2010], which is discussed later.

[3]Some contention-aware thread schedulers were implemented entirely at user level.

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. Figure 1a shows a CMP architecture where all four cores share the same resources. 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 1b 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 may result in different performance. Given that contention-aware schedulers focus on space sharing, they will only be effective on architectures like those in Figure 1b, where subsets of cores share different resources, and *not* on architectures like in Figure 1a.

It is worth highlighting at this point that most existing contention-aware schedulers assume that all subsets of cores share equally-sized last-level caches, so in this survey we focus on schedulers that rely on this assumption. Recent work [Jiang et al. 2011], however, has demonstrated that asymmetric-cache CMP designs coupled with effective asymmetry-aware and contention-aware scheduling support, make it possible to deliver higher performance-per-watt than their symmetric-cache counterparts. As a result, next-generation contention-aware schedulers may also have to deal with this source of asymmetry in the future.

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 the developers of contention-aware schedulers [Zhuravlev et al. 2010; Knauerhase et al. 2008; Banikazemi et al. 2008; Jiang et al. 2008] because all experimented with workloads consisting of only single-threaded applications that cannot share data or interfere constructively[4].

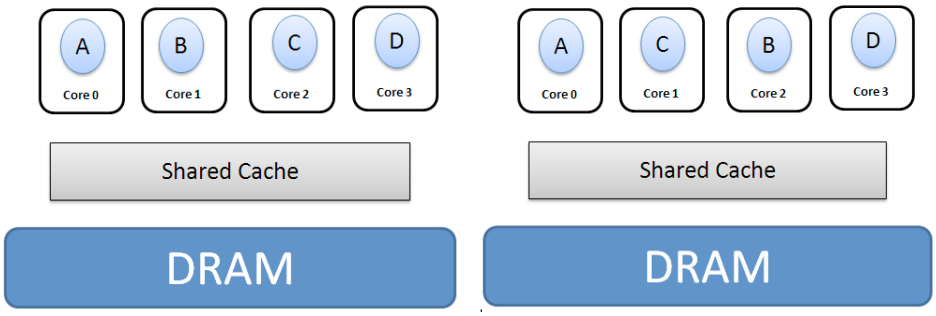
Constructive interference for multithreaded applications where data sharing is present is discussed in depth in Section 3. However, as will be discussed in that section, evidence indicates that because modern programming/compilation techniques try to purposely minimize inter-thread sharing the effects of cooperative sharing are often overshadowed by the magnitude of destructive interference even for multithreaded applications.

2.4 The building blocks of a contention-aware scheduler

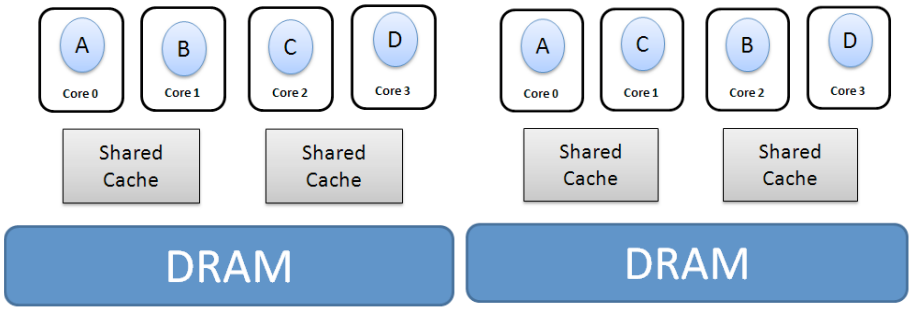
A major commonality across all contention-aware schedulers is that they can be regarded as consisting of four building blocks: the objective, the prediction, the decision, and the enforcement.

2.4.1 *The objective.* The objective describes the metric that the scheduler is trying to optimize. The most common objectives optimized by contention-aware hardware and software solutions are overall throughput and fairness/QoS. Contention-

[4]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 results from the applications using shared libraries.



(a) Two thread-to-core-mappings on a CMP architecture where all cores share the same resources



(b) Two thread-to-core-mappings on a CMP architecture where every pair of cores share different resources.

Fig. 1: Examples of thread-to-core mappings in two CMP configurations.

aware schedulers almost exclusively focus on optimizing overall throughput. These schedulers typicaly strive to protect threads that "suffer" the most from contention and thus lower the overall performance degradation for the workload. A pleasant side effect is that these strategies yield performance stability when compared to the default OS scheduler [Zhuravlev et al. 2010].

Default OS schedulers, which are contention unaware at the time of this writing, are primarily tasked with balancing the runqueues across cores. Since the number of ways to map threads to cores in a given time interval is combinatorial in the number of cores, the actual mappings of threads to cores can vary dramatically from run to run of the *same workload* under the default scheduler. Some of these mappings will offer low contention and high performance. Others will offer high contention and low performance. As such, the performance of workloads under the default OS scheduler can be varied and unpredictable [Zhuravlev et al. 2010]. Contention-aware schedulers, on the other hand, have a policy in place regarding which threads should be mapped closer together and which should be mapped further apart. As such, the mappings that these schedulers produce are similar if not identical from run to run. Predictable mappings mean predictable and stable performance. Although, so far there have been no explicit solutions offering QoS

via contention-aware scheduling alone, the predictability and repeatability provided by such scheduling algorithms makes it a clear possibility for the future.

2.4.2 *The prediction.* Having established an objective (in this case, improved overall throughput) one now needs to find a thread-to-core mapping that will facilitate this objective. The search space is too vast for a dynamic trial and error exploration. Consider a machine with 8 cores where every pair of cores shares the LLC. There are 8! = 40,320 ways to map 8 threads onto the 8 available cores. A lot of these mapping are redundant, for example if cores 0 and 1 share an LLC then the mapping (A to 0 and B to 1) or (A to 1 and B to 0) are equivalent in terms of performance. Nevertheless, there are still 105 performance-unique mappings to explore. Furthermore, 8-core machines are now on the lower end of server capabilities. Machines with 16, 32, and even 64 cores are becoming available with 100+ cores expected in the near future. The sheer number of mappings makes trial and error infeasible. As such, it becomes necessary to be able to predict the performance of different mappings without actually trying them.

The prediction is the second building block common to all contention-aware schedulers. The prediction problem can be best illustrated with a simple example. Consider a machine with 4 cores where each pair of cores shares the LLC, like the machine pictured in Figure 1b. There are three performance-unique ways to map four threads A, B, C, and D onto this system: (AB CD), (AC BD), and (AD BC). The prediction model should be able to predict the relative "goodness" of each of the three mappings so that the scheduler can choose the one that will result in the best performance.

Many different techniques have been proposed for modeling how performance of applications degrades as they share resources on multicore systems [Weinberg and Snavely 2008; Cascaval et al. 2000; Zhou et al. 2004; Azimi et al. 2009; Hoste and Eeckhout 2007; Azimi et al. 2007; Zhong et al. 2009]. The majority focus on sharing the LLC, which many researchers believed was the primary source of contention. The best known techniques use Stack Distance Profiles (SDP) and Miss Rate Curves (MRC) to predict the performance of multiple threads sharing the LLC. SDPs were first proposed by Mattson [Mattson et al. 1970] and first used for prediction purposes by [Chandra et al. 2005]. They are a concise description of the memory reuse patterns of an application and a measure of the benefit derived from additional cache space.

The caches used on modern machines are set-associative, often with a pseudo-LRU replacement policy. An SDP shows the distribution of cache hits among the different positions in the LRU stack. In other words it is a histogram depicting the number of cache hits that were satisfied from the Most Recently Used (MRU) position, the number satisfied from the Least Recently Used (LRU) position and each of the positions in-between. If an SDP was obtained for an N-way set associate cache then it can be used to estimate the extra misses that an application would incur if running on an (N-1) set associate cache by counting the hits to the LRU position on an N-way associative cache as misses on an (N-1)-way associative cache. Thus, it is possible to calculate the extra misses a thread will encounter on a variety of different cache architectures using SDPs. The resultant curves depicting miss rate as a function of cache size are known as Miss Rate Curves (MRC).

It is more challenging to predict how two threads sharing a cache will perform than it is to predict performance based on cache size, because in the case of sharing it is not clear how much space each thread will obtain for itself. Nevertheless there are models that predict the resultant miss rate when sharing the LLC based on the applications' SDP or MRC. The best known such model is called Stack Distance Competition and comes from Chandra et al. [Chandra et al. 2005]. They use an algorithm to merge two SDPs into a single profile and estimate the resultant extra misses for each application. The model has been extensively evaluated on a cycle accurate simulator and shown to be very accurate [Chandra et al. 2005].

While SDPs and MRCs were shown to be an effective tool for modeling contention in the LLC (and so they could help contention-aware schedulers to significantly improve the accuracy of the prediction), their main limitation is that they are difficult to obtain online on current systems. Existing hardware counters do not provide sufficient data to obtain an SDP and the required changes to the hardware [Qureshi and Patt 2006], though simple enough to be implemented, have not yet been adopted by commercial systems. Similarly, the methods proposed so far for generating MRCs (e.g., RapidMRC [Tam et al. 2009]) rely on certain hardware counters that are only available on some platforms[5].

One approach to get around the need to use SDP is to approximate cache occupancy of competing threads using simple *performance metrics* (such as the LLC miss and access rate) that be easily collected using the hardware performance monitoring infrastructure available in most modern processors. [Banikazemi et al. 2008], for instance, predict the performance of a particular thread in a particular mapping by first calculating 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 a 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.

The method proposed by [Banikazemi et al. 2008] is rather complex, but fortunately the most recently proposed contention-aware schedulers are able to avoid this complexity. Schedulers proposed by [Knauerhase et al. 2008; Zhuravlev et al. 2010; Merkel et al. 2010] approximate contention with one simple heuristic: the LLC miss rate (with small differences in how they measure it). Based on observation and experiments, these researchers concluded that applications that frequently miss in the LLC will stress the entire memory hierarchy for which they are competing; as such, these applications should be kept apart. Their prediction model can be summed up as follows: putting applications with numerous LLC misses together leads to poor performance. They do not attempt to convert this intuition into a numerical prediction for the resultant CPI. Instead they develop concepts like cache light/heavy

[5]The IBM Power V processor is the exception which offers facilities to dynamically collect the SDP. However, Intel and AMD processors, which constitute the majority of the market, do not.

threads [Knauerhase et al. 2008] and turtles/devils [Zhuravlev et al. 2010] to represent low-miss and high-miss applications respectively. The terminology devil/turtle is borrowed from [Xie and Loh 2008], in which the authors divided applications into categories that were assigned animal names. The intention is that pairing light and heavy users of the memory hierarchy (i.e., turtles and devils) yields the best performance. Similarly, [McGregor et al. 2005] use the concept of high pressure and low-pressure threads based on the number of bus transactions that they complete and attempt to schedule high and low-pressure threads together.

The fact that such a simple and coarse heuristic works well to approximate contention ([Zhuravlev et al. 2010] showed that a scheduler based on this heuristic performs within 3% of the optimal "oracular" scheduler) seems at a first glance very surprising. Indeed, the LLC miss rate is a very coarse heuristic for approximating cache contention, because it does not reflect key factors like cache locality or reuse. The reason why it does work is that it captures contention for the other parts of the memory hierarchy, such as memory controllers and interconnects. Intuitively, a thread with a high LLC miss rate will put pressure on memory controllers and interconnects and will thus suffer from contention for these resources and cause interference with other threads. The key to this puzzle is that contention for memory controllers and interconnects is dominant on existing systems. [Zhuravlev et al. 2010; Blagodurov et al. 2011] performed experiments that demonstrated this fact on two different AMD and Intel platforms. As a result, a simple and easy-to-obtain LLC miss rate heuristic provides sufficient information to predict contention on existing multicore systems.

Taking advantage of this result, the most recently proposed schedulers (and those that were also implemented and evaluated on real hardware) use very simple prediction algorithms as described above [Knauerhase et al. 2008; Zhuravlev et al. 2010; Merkel et al. 2010]. Should cache contention become more dominant in future systems, a combination of techniques relying on LLC miss rates and on heuristics approximating cache contention will be needed to implement a successful contention-aware scheduler.

2.4.3 *The decision.* Once a prediction model is developed, the scheduler must select the actual thread mapping that will be used. This task is handled by the third building block of the scheduler: the decision. One might imagine that if the prediction mechanism did its job well then it should be trivial to pick the best solution among the possible choices. However, as [Jiang et al. 2008; Tian et al. 2009] show that even if the exact performance for every possible mapping is known it is still a challenging task (NP-complete in the general case) to find a good, let alone the best possible mapping.

[Jiang et al. 2008; Tian et al. 2009] address the contention-aware scheduling problem exclusively from the perspective of making the scheduling decision. They assume that the exact performance degradations of any subset of threads co-scheduled on the same LLC are known. Going back to our earlier example of mapping four threads A, B, C, and D to a system with 4 cores where each pair of cores share the LLC, [Jiang et al. 2008; Tian et al. 2009] assume that they have the performance degradations for every possible thread pairing sharing the LLC: AB, AC, AD, BC, BD, and CD.

Assuming that this performance degradation information is 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 LLC. Since all pairings are possible this is a complete graph. They show that the optimal solution to this scheduling problem is a *minimum weight perfect matching* of the graph.

For systems where caches 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 an LLC, the problem is NP-complete as proven by [Jiang et al. 2008]. 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 cache. Yet, machines with four and six cores per cache are already widely available. [Jiang et al. 2008; Tian et al. 2009] present a series of polynomial time approximation algorithms, which they show can result in solutions very near to optimal.

The scheduler presented by [Banikazemi et al. 2008] (as discussed above) uses a complex model that predicts the performance of each thread in a given schedule and hence presents the decision mechanism with the kind of input assumed by [Jiang et al. 2008; Tian et al. 2009]. They, however, use a very rudimentary technique of enumerating all possible mappings and selecting the one with the best performance. 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 [Jiang et al. 2008], the number of choices explodes as the number of cores sharing the LLC increases.

[Zhuravlev et al. 2010; Knauerhase et al. 2008; McGregor et al. 2005; Merkel et al. 2010] put more stress on their decision mechanism because the prediction mechanism used was rather simple and never predicted direct performance of threads but rather provided metrics that hint at the relative performance of each schedule.

The major decision mechanism proposed by [Zhuravlev et al. 2010] is called *distributed intensity (DI)*. It sorts all threads to be scheduled based on their miss rate. It then begins pairing applications from opposite ends of the list. The most intense application is paired with the least intense. The second most intense is paired with the second least intense, and so forth. This is done every time a new thread is spawned, a thread terminates, and every predefined time period.

The decision mechanisms proposed in [Knauerhase et al. 2008] are threefold OBS-L, OBS-X, and OBS-C. The OBS-L 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 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. 2007].

The decision mechanism in [Merkel et al. 2010] is based on the so-called activity vectors. A thread's *activity vector* records its usage of system resources during the previous time slice; mainly the memory-bus, the LLC, and the rest of the core; normalized to the theoretical maximum usage. The proposed OS-level scheduler exploits thread migrations to enforce co-scheduling of 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. 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 run queues 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.

The decision mechanism in [McGregor et al. 2005] is 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 the other schedulers that combined threads with complementary resource usage [Zhuravlev et al. 2010; Knauerhase et al. 2008; Merkel et al. 2010]. Once a job has been selected the remaining cores are filled with threads that will be the most complementary to this job. In other words, threads with a high memory bus usage are combined with those with a low memory bus usage.

2.4.4 *The enforcement.* Once the decision mechanism has settled on a thread placement, this placement must be enforced by binding or migrating the threads to the cores specified in the placement. Modern operating systems provide functionality that allows the binding of threads to cores from the user level via system calls with no modification to the kernel. [Zhuravlev et al. 2010; Banikazemi et al. 2008; McGregor et al. 2005] all go this route making a scheduler that runs as a user level process and enforces its decisions via Linux system calls.

The enforcement mechanism of [Zhuravlev et al. 2010] binds every thread to a specific core while [Banikazemi et al. 2008] uses a more flexible mechanism of CPU-sets. They bind a thread to a set of CPUs, for example the set of CPUs that do not share an LLC. The enforcement mechanism is a design choice that determines how much of the onus for thread scheduling is moved into the contention-aware scheduler and how much remains inside the traditional OS scheduler. The CPU-set version chosen by [Banikazemi et al. 2008] allows the OS scheduler to retain more control as it is still able to move threads within the CPU-sets to enforce load balance. The direct thread-to-core mapping technique employed by [Zhuravlev et al. 2010] places all load balancing responsibilities with the contention aware scheduler leaving the OS scheduler only the time domain.

The schedulers proposed by [Knauerhase et al. 2008; Merkel et al. 2010] were directly integrated into the kernel. They enforce scheduling decisions by directly manipulating the run queues. Such an approach is more suited for situations where the number of threads exceeds the number of cores, as was the case only in experimentation performed by [Knauerhase et al. 2008; Merkel et al. 2010].

2.5 Discussion of complementary techniques

Two techniques related to the management of shared-resource contention are worth mentioning in the context of this survey: DRAM controller scheduling and cache partitioning. Although these techniques do not fall under the category of thread scheduling algorithms, they can certainly help contention-aware scheduling algorithms accomplish their goals.

2.5.1 *DRAM controller scheduling.* One of the most critical shared resources in a chip multiprocessor is the DRAM memory. The DRAM memory system in modern computing platforms is organized into multiple banks. Because of this organization, DRAM systems are not truly random access devices (equal access time to all locations) but rather are three-dimensional memory devices with dimensions of bank, row, and column. While accesses to different banks can be serviced in parallel, sequential accesses to different rows within one bank have high latency and cannot be pipelined.

Existing controllers for DRAM memory systems typically implement variants of the first-ready-first-come-first-serve (FR-FCFS) policy [Rixner et al. 2000]. FR-FCFS prioritizes memory requests that hit in the row-buffers associated with DRAM banks (a.k.a. *row-buffer hits*) over other requests, including older ones. If no request is a row-buffer hit, then FR-FCFS prioritizes older requests over younger ones. This policy has been shown to significantly improve the system throughput in single-core systems [Rixner et al. 2000]. However, other authors [Moscibroda and Mutlu 2007; Mutlu and Moscibroda 2007] have shown that the fact that memory accesses requested by all cores are treated uniformly by FR-FCFS leads this policy to poor system throughput in CMPs and to potential starvation of threads with low row-buffer locality.

There have been several recently proposed memory aware schedulers for CMP systems that aim to "equalize" the DRAM-related slowdown experienced by each thread due to interference from other threads, and at the same time avoid hurting the overall system performance. Inspired by the fair queuing algorithms widely used in computing networks, [Nesbit et al. 2007] proposed the *Fair queuing memory scheduler* (FQM). For each thread, in each bank, FQM keeps a counter called virtual runtime; the scheduler increases this counter when a memory request of the thread is serviced. FQM prioritizes the thread with the earliest virtual time, trying to balance the progress of each thread in each bank.

Another algorithm that deserves special attention is the *parallelism-aware batch scheduling algorithm* (PAR-BS) [Mutlu and Moscibroda 2008], which delivers higher memory throughput and exploits per-thread bank-level parallelism more effectively than FQM. To make this possible, the PAR-BS scheduler attempts to minimize the average stall time by giving a higher priority to requests from the thread with the shortest stall time at a given instant. This scheduling algorithm relies on the

concept of batches. The idea of batching is to coalesce the oldest k outstanding requests from a thread in a bank request buffer into units called *batches*. When a batch is formed, PAR-BS builds a ranking of threads based on their estimated stall time. The thread with the shortest queue of memory requests (number of requests to any bank) is heuristically considered to be the thread with the shortest stall time and its requests are serviced preferentially by PAR-BS.

Although PAR-BS is currently the best known scheduling algorithm in terms of memory throughput for systems with a centric DRAM controller, recent research has highlighted that it does not have good scalability properties, so it does not turn out suitable for systems with multiple[6] DRAM controlers [Kim et al. 2010]. To the best of our knowledge, ATLAS (*Adaptive per-Thread Least-Attained-Service*) [Kim et al. 2010] is the first memory scheduling algorithm specifically designed for systems with multiple DRAM controllers. In order to reduce the coordination between memory controllers, ATLAS divides execution time into long time intervals or *quanta*, during which each controller makes scheduling decisions locally based on a system-wide thread ranking. At the beginning of each quantum, a new system-wide ranking is generated by exchanging information across memory controllers. Such a ranking dictates that requests from threads that received the least service so far from the memory controllers will be serviced first. These working principles enable ATLAS to provide superior scalability and throughput than PAR-BS on systems with multiple memory controllers.

2.5.2 *Cache partitioning.* As we stated earlier, the LRU replacement policy, which is typically used in existing caches, does not always guarantee an efficient distribution of the shared last-level cache resources across applications on CMPs. Some researchers [Suh et al. 2002; Suh et al. 2004; Chandra et al. 2005; Qureshi and Patt 2006] proposed solutions to address this problem by explicitly *partitioning* the cache space based on how much applications are likely to benefit from the cache, rather than based on their rate of demand.

In the work on Utility Cache Partitioning [Qureshi and Patt 2006], a custom monitoring circuit is designed to estimate an application's number of hits and misses for all possible number of ways allocated to the application in the cache (the technique is based on stack-distance profiles (SDPs)). The cache is then partitioned so as to minimize the number of cache misses for the co-running applications. UCP minimizes cache contention given a particular set of co-runners.

Tam et al. [2009] similarly to other researchers [Cho and Jin 2006b; Lin et al. 2008; Zhang et al. 2009] address cache contention via software-based cache partitioning. The cache is partitioned among applications using *page coloring*. A portion of the cache is reserved for each application, and the physical memory is allocated such that the application's cache lines map only into that reserved portion. The size of the allocated cache portion is determined based on the marginal utility of allocating additional cache lines for that application. Marginal utility is estimated via an application's reuse distance profile, which is approximated on-

[6] As we mentioned earlier, cutting-edge processors from main hardware manufacturers integrate on-chip memory controllers and so multiprocessor systems based on these processors include multiple DRAM controllers, each controlling requests to a different portion of main memory.

line using hardware counters [Tam et al. 2009]. Software cache partitioning, like hardware cache partitioning, is used to isolate threads that degrade each other's performance. While this solution delivers promising results, it has two important limitations: first of all, it requires nontrivial changes to the virtual memory system, a complex component of the OS. Second, it may require copying of physical memory if the application's cache portion must be reduced or reallocated. Although smarter implementations can in part overcome the latter limitation by tracking and coloring only frequently used "hot" pages, these work well only when recoloring is performed infrequently [Zhang et al. 2009]. Given these limitations, it is desirable to explore options like scheduling, which are not subject to these drawbacks.

It is also worth highlighting that solutions based exclusively on cache partitioning only help alleviate LLC contention, but not memory controller and interconnect contention, which were shown to be the dominant on modern systems [Zhuravlev et al. 2010; Blagodurov et al. 2011]. As a result, these solutions cannot replace contention-aware schedulers, which usually approximate contention for memory controllers and interconnects, but not for the LLC. Cache partitioning, however, can be complementary to thread scheduling.

Beyond exploiting cache partitioning to mitigate contention on CMPs, other researchers have demonstrated the potential of cache partitioning to make effective utilization of Non-Uniform Cache Access (NUCA) designs [Dybdahl and Stenstrom 2007; Hardavellas et al. 2009; Awasthi et al. 2009; Chaudhuri 2009]. Recent research has highlighted that the increase of the number of cores and cache banks, coupled with the increase of global wire delays across the chip in future manufacturing technologies, make current on-chip last-level-cache designs (centric caches with single, discrete latency) undesirable for upcoming systems [Kim et al. 2002]. In NUCA designs the shared cache is statically organized into private per-core partitions, but partitions can keep blocks requested by other cores (this usually happens when cores run out of cache space on its private partition). Although this cache organization promises lower access latencies than centric designs, making effective use of upcoming NUCA architectures is a challenging task. Recent research has proposed promising solutions to address this problem using novel cache-partitioning schemes, either purely hardware-based solutions [Dybdahl and Stenstrom 2007; Hardavellas et al. 2009] or OS-assisted ones [Awasthi et al. 2009; Chaudhuri 2009].

2.6 Discussion and conclusions

Shared resource contention on CMPs is a serious problem that leads to overall performance degradation as well as makes it difficult if not impossible to provide QoS to individual applications. A significant amount of work has been done exploring potential solutions to this problem with the two main directions focusing on either shared cache partitioning or modifications to the DRAM controller's request scheduling system. Although these solutions seem promising they require significant changes to the underlying hardware and the operating system. Research on contention-aware scheduler has emerged as a simpler alternative for mitigating shared resource contention.

There are several contention-aware schedulers in existence today. Even though they approach the problem in different ways, use different prediction, decision, and enforcement mechanisms, the end result is very similar. A contention-aware

scheduler lives either in user or kernel space and it augments the decisions taken by the tradition OS scheduler to find thread-to-core mappings that result in less shared resource contention. Existing contention-aware schedulers can be separated into two categories: those that focus on modeling cache contention (using stack-distance profiles or miss rate curves, or by approximating cache space occupancy using simple hardware performance counters) and those that approximate contention for the entire memory hierarchy using a simple heuristic: the last-level cache miss rate. The latter algorithms have a much simpler implementation, do not require additional hardware support and are readily portable to different systems[7]. The former algorithms, while being more complex, do model contention for the last-level cache, and so might become more relevant if cache contention takes on a higher importance than memory controller and interconnect contention.

The contention-aware schedulers have been shown to be effective at mitigating shared resource contention and improving performance as well as predictability, however schedulers do not solve shared resource contention but rather try to *avoid* it. As such, there are limitations to what schedulers can accomplish. Schedulers rely on the heterogeneity of workloads. To be truly effective they require a workload that consists of both memory-intensive as well as compute-intensive threads; that way, by co-scheduling threads with complementary resource usage, they are able to avoid contention as compared to contention-unaware schedulers. A contention-aware scheduler becomes less effective as the workload becomes more homogeneous, becoming completely pointless if all the threads are the same. Furthermore, as contention-aware schedulers are not able to actually eliminate shared resource contention in any way even the best possible thread-to-core mapping may result in high overall contention and performance degradation.

It is therefore unrealistic to believe that contention-aware thread schedulers will solve the problem of shared resource contention on CMPs by themselves. To solve this problem we need a truly comprehensive solution that addresses all the bottlenecks in the memory hierarchy and most likely involves hardware techniques to which we alluded in the beginning of this section. Considering the ease with which contention-aware schedulers can be implemented and their demonstrated effectiveness, makes us believe that any comprehensive contention-mitigating solution should involve contention-aware schedulers. Using a contention-aware scheduler as the starting point means having to run a race not from the beginning but rather from half-way. Contention-aware schedulers will avoid as much unnecessary contention as possible. Only what cannot be avoided needs to be solved by other means.

3. COOPERATIVE RESOURCE SHARING

Cooperative sharing becomes important primarily for multithreaded applications, where threads share data. Threads may be concurrently updating the same data or simply reading the same data either concurrently or one after another. Another popular pattern is producer-consumer, where one thread writes some data, and then another thread reads it. In this scenario, it may be beneficial to co-schedule

[7]For instance, an algorithm implemented on Linux by Blagodurov et al [2011] was ported to Solaris by [Kumar et al. 2011].

threads on cores that share a part of the memory hierarchy (e.g., a last-level cache), so as to minimize the cost of inter-core data exchange and take advantage of data pre-fetching that threads can do for each other.

Since data sharing occurs primarily in multithreaded applications, a lot of sharing-aware scheduling algorithms where built in parallel runtime libraries, such as Phoenix MapReduce [Chen et al. 2010]. We omit discussing this vast area of research, in the interest of staying true to our focus on operating system schedulers.

The main difference between schedulers implemented in parallel runtime libraries and OS schedulers is that the former typically have a lot of information about how threads or tasks in the application share data, while the latter sees threads as black boxes and has no information about inter-thread data sharing. The main challenge for the OS scheduler, therefore, is to detect whether threads share data and whether the degree of sharing warrants changing thread-to-core mappings.

The first research in this area was done long before multicore systems became mainstream, in 1994 by Thekkath and Eggers [Thekkath and Eggers 1994]. They examined parallel scientific applications on a multithreaded system, where like on multicore systems thread contexts share portions of the memory hierarchy. To their surprise they discovered that sharing-aware co-scheduling of threads – e.g., placing threads that share data on thread context that share a cache – did not yield performance benefits. When they investigated the cause for this surprising finding they discovered that the culprit was the *pattern* in which the threads shared the data. It turns out that the applications that they examined (they used the SPLASH benchmark suite) exhibit the so-called *coarse-grained sharing*. That is, they do not ping-pong data back and forth, but rather there is a relatively long period where one thread accesses the data followed by a long period where another thread accesses the same data. As a result, the cost associated with bringing the data into the cache of another processor when threads are not co-scheduled is amortized over a long period during which this data is accessed.

This finding becomes less surprising when one puts into perspective the environment for which SPLASH (and other scientific applications of that time) were created. They were designed for the most part by experienced programmers for multiprocessor systems, where cross-chip data sharing was extremely expensive. Therefore, these applications were optimized to avoid fine-grained sharing – exactly the kind of sharing that would benefit from sharing-aware co-scheduling.

More than a decade later, a study by Zhang et al. arrived at a similar finding, upon examining a more "modern" PARSEC benchmark suite: these applications are simply not designed to share data in a way that sharing-aware co-scheduling would make a difference [Zhang et al. 2010]. Although PARSEC is a relatively recent benchmark suite, it is based on scientific applications similar to those used in SPLASH. Zhang's study presented another interesting observation: if applications were specifically redesigned with the assumption that they would run on a multicore machine with a shared memory hierarchy and that threads sharing data would be co-scheduled on cores sharing a cache, the application can run faster than the original version that tried to avoid sharing. The key is to redesign data access patterns such that threads pre-fetch data for each other. A sharing-aware scheduler is crucial for such applications, because threads that share data must be co-scheduled

to run on cores sharing a cache.

From this research we can conclude that for older scientific applications, sharing-aware schedulers are not likely to bring any performance benefits because of the nature of sharing among the application threads. However, as parallel computing enters the mainstream and new kinds of parallel applications emerge, this could change. Sharing-aware schedulers could be important for other types of applications.

As a confirmation of this possibility, Tam et al. looked at database and server applications, such as SPEC JBB and VolanoMark [Tam et al. 2007]. They found that sharing-aware thread placement did have an effect on these applications, and so the sharing-aware scheduler they proposed (described below) did yield performance improvement, albeit only as much as 7%. In a similar note, recent work on multicore operating systems and on multicore profilers (e.g., Corey [Boyd-Wickizer et al. 2008], Barrelfish [Baumann et al. 2009], DProf [Pesterev et al. 2010]) demonstrated examples of applications, including the OS kernel itself, where cross-cache sharing of data is extremely expensive due to the overhead of coherency protocols. In these cases, co-scheduling applications threads that share data on cores that share a cache can eliminate that overhead. For example, Kamali, in his work on sharing-aware scheduling, demonstrated that memcached, which is one of the applications used in the DProf study, experienced a performance improvement of 28% when managed by a sharing-aware scheduler [Kamali 2010].

We now describe two sharing-aware thread schedulers implemented for modern multicore systems. Recall that we are focusing on schedulers that see threads as "black boxes" in a sense that they do not assume any knowledge about how threads share data. One such scheduler was designed by Tam et al. [Tam et al. 2007]. Their scheduler relied on hardware performance counters specific to the IBM Power processors to detect when threads running on different cores satisfy last-level cache misses from a remote cache. The scheduler profiled this data and analyzed the intensity of remote cache accesses. If remote accesses were deemed to be sufficiently frequent the threads that were considered to be the cause of remote misses were co-scheduled to run on cores that share a last-level cache. The benefits of this approach is that it assumed no advance knowledge about sharing patterns in the application and could adapt if sharing patterns changed over the lifetime of the application. The downside is that this algorithm could only be implemented on IBM Power processors.

Kamali attempted to implement a similar algorithm such that it would rely only on hardware counters that are available on most modern server processors [Kamali 2010]. However, like Tam he found this challenging and ended up implementing an algorithm that worked only on the latest Intel processors. His algorithm needed hardware counters that measured cross-cache coherency misses, and they were not available on AMD systems. Contrary to Tam, however, Kamali went further and developed a highly accurate performance model that could predict how much performance improvement an application would experience if its threads were co-scheduled on cores that share a cache. The model could be used online, without any prior knowledge of applications and without imposing any measurable overhead on application runtime. It relied on a simple linear regression and required simpler

data analysis than that used by Tam in his algorithm. Armed with this model, a sharing-aware scheduler could be implemented very simply. It could simply examine sharing among all pairs of caches, run the model to predict the effects of co-scheduling threads running on cores sharing those caches, and decide whether migrating threads is worthwhile based on those predictions.

Another benefit of having an accurate model that estimates benefits of sharing is that a scheduler that uses this model could be combined with a contention-aware scheduler – something that has not yet been done in any scheduling algorithm and is a major shortcoming of existing contention-aware and sharing-aware algorithms. In particular, a scheduler could estimate the benefit of sharing-based co-scheduling as well as assess the degradation from resource contention if the threads were co-scheduled. Based on these two estimates it could then make a decision whether or not the benefits of co-scheduling (cooperative data sharing) outweigh its costs (contention for shared resources).

3.1 Summary

In summary, while sharing-aware scheduling was shown to have little impact on older applications created for multiprocessor systems, this may change as new multithreaded applications are emerging. Sharing-aware schedulers that would benefit these applications do exist, but on some systems they are limited by the lack of hardware performance counters that would help the operating system detect inter-thread sharing. So the importance and effectiveness of sharing-aware OS scheduling on future systems will be determined by the kinds of sharing patterns found in future multithreaded applications, by the availability of certain hardware performance counters, and whether or not the task of sharing-aware scheduling will be taken over by application runtime libraries.

4. STATE-OF-THE-ART SCHEDULING TECHNIQUES IN LINUX AND SOLARIS

In this section we discuss the state-of-the-art scheduler implementations available in the latest releases of two popular operating systems: Linux and Solaris. It is important to note that the main goal of these schedulers on multicore systems is load balancing across cores. Hence, the strategies employed to place threads on cores aim to balance the runnable threads across the available resources to ensure fair distribution of CPU time and minimize the idling of cores.

Schedulers built with primarily load balancing in mind are very different from those surveyed in this work. These load-balancing schedulers do not take advantage of the available hardware performance counters, they do not construct performance-models, or in any other way attempt to characterize the runnable threads to determine more optimal thread-to-core mappings. With the exception of balancing threads across caches when the load is light, they do not strive to avoid shared resource contention or conversely take advantage of cooperative inter-thread sharing. The primary goal is to fairly distribute the workload across the available cores. As discussed in Section 2 such a rudimentary approach has the potential to lead to poor and unpredictable thread performance.

This section is organized into two subsections. The first discusses the Linux scheduler and the latter focuses on the Solaris scheduler. Each subsection is further divided into three parts. The first part answers the question: given several threads

mapped to a core how does the scheduler decide which thread to run next? The second discusses how the scheduler decides on the thread-to-core-mappings. The final part addresses how the schedulers incorporate the actual physical layout of the machine, particularly in regards to NUMA architecture, when making scheduling decisions.

4.1 The Linux scheduler

Contemporary multiprocessor operating systems, such as Linux 2.6 (but also Solaris 10, Windows Server, and FreeBSD 7.2), use a two-level scheduling approach to enable efficient resource sharing. The first level uses a distributed run queue model with per core queues and fair scheduling policies to manage each core. The second level is a load balancer that redistributes tasks across cores. The first level schedules in time, the second schedules in space. We discuss each level independently in the subsections that follow.

4.1.1 *Run queue management.* The default scheduler in Linux is the *Completely Fair Scheduler (CFS)* which was merged into mainline Linux version 2.6.23. Since then, there have been some minor improvements made to CFS scheduler. We will focus on the CFS scheduler in Linux kernel 2.6.32.

The threads that are currently mapped to a core are stored in the core's *run queue*; a run queue is created for each core. Task scheduling (deciding which thread from the run queue will run next) is achieved via *CFS run queue management*. A novel feature of the CFS scheduler is that it does not require the concept of time slices. Classical schedulers compute time slices for each process in the system and allow them to run until their time slice is used up. CFS, in contrast, considers only the waiting time of a process (how long it has been in the run queue and was ready to be executed). The task with the gravest need for CPU time is always scheduled next.

This is why the scheduler is called completely fair: the general principle is to provide maximum fairness to each task in the system in terms of the computational power it is given. Or, put differently, it tries to ensure that no task is treated unfairly. The unfairness is directly proportional to the waiting time. Every time the scheduler is called, it picks the task with the highest waiting time and assigns it to the processor. If this happens often enough, no large unfairness will accumulate for tasks, and the unfairness will be evenly distributed among all tasks in the system.

Also unique to the CFS is that unlike other OS schedulers or even prior Linux implementations it does not maintain an array with run queues for each priority level. Instead the CFS maintains a time-ordered *red-black tree* (a self-balancing data structure, with the property that no path in the tree will ever be more than twice as long as any other).

Different priority levels for tasks must be taken into account when considering which task should run next, as more important processes must get a higher share of CPU time than less important ones. CFS does not use priorities directly but instead uses them as a decay factor for the time a task is permitted to execute. Lower-priority tasks have higher factors of decay, while higher-priority tasks have lower factors of delay. This means that the time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task.

4.1.2 *The load balancer.* In the CMP environment, each CPU has its own run queue. As a result of uneven blocking intervals and completion times among threads some queues may run out of work, leaving the corresponding processors idle, while other processors would have threads waiting in their run queues. This is called load imbalance, and the load balancer is designed to address this issue. The "load" on each core is defined as the number of tasks in the per-core run queue. Linux attempts to balance the queue lengths system wide by periodically invoking the load balancing algorithm on every core to migrate tasks from the busiest CPU to less-loaded CPUs.

When selecting potential migration candidates, the kernel ensures that the candidate thread is not running at the moment or has not been running on a CPU recently (so as not to lose its cache affinity). Another consideration, is that the candidate process must be permitted to execute on the destination CPU (i.e., CPU affinity has not been set to prohibit that).

4.1.3 *Topology and locality awareness.* The Linux scheduler has topology awareness which is captured by organizing all run queues into a hierarchy of *scheduling domains*. The scheduling domains hierarchy reflects the way hardware resources are shared: Simultaneous Multithreading (SMT) contexts, last-level caches, socket and NUMA domains [Hofmeyr et al. 2010].

Balancing is done progressing up the hierarchy. At each level the load balancer determines how many tasks need to be moved between two groups to balance the cumulative loads in those groups. If the balance cannot be improved (e.g. one group has 3 tasks and the other 2 tasks) no threads will be migrated. The frequency with which the load balancer is invoked is dependent on both the scheduling domain kernel settings and the instantaneous load.

The default parameter settings are established such that, the frequency of balancing and the number of migrations decreases as the level in the scheduling domain hierarchy increases. This is based on the idea that migrations between domains further up on the hierarchy, such as between two different NUMA banks, are more costly than those between domains lower down, such as between SMT contexts.

4.2 The Solaris scheduler

In some respects the Solaris scheduler is similar to the Linux scheduler. They both make distributed per-core decisions as well as global load balancing decisions, and both support topology and locality awareness. There are, however, stark differences between the two. For example, in Solaris, a thread's priority exclusively determines how soon it will run.

The Solaris scheduler has a modular design and consists of two components: the *scheduling classes* and the *dispatcher*. While the dispatcher is in charge of key tasks such as load balancing and global run queue management, the scheduling classes handle CPU accounting, time-slice expiration and priority boosting. Every thread in the system is in one of several possible scheduling classes[8]; this arrangement

[8]Solaris supports several scheduling classes: time-share, fair-share, real-time, fixed-priority and interactive. The scheduling classes are implemented as separate, dynamically loadable scheduling modules.

determines the range of priorities for the thread, as well as which class-specific scheduling algorithms that will be applied as the thread runs.

4.2.1 *Run queue management.* The Solaris kernel implements a *global priority scheme*, which guarantees that a runnable thread's global priority solely determines how soon it will be selected to run. The dispatcher makes this possible by enforcing that threads with the highest priorities run first than other threads at any time.

Runnable threads are assigned a global priority ranging from 0 to 169. The global priority range is divided into several priority subranges associated to the different scheduling classes. Solaris's per-core run queues are organized as an array, where a separate linked list of threads is maintained for each global priority.

4.2.2 *The load balancer.* Apart from being in charge of finding the highest-priority runnable thread and dispatching it onto the target CPU, the Solaris dispatcher is also responsible for making load balancing decisions. These decisions are performed based on the topology of the platform. For example on a two-core chip, even if only one of the two cores is busy, it is still better to try to find a chip where both CPUs are idle. Otherwise, we could end up having a chip with two busy cores and another chip on the same system with two idle cores. More succinctly, for systems with multicore chips, the scheduler tries to load-balance across physical chips, spreading the workload across CPUs on physical chips rather than filling up cores on chip 0, then the cores on chip 1, and so on. This is the case on Uniform Memory Access (UMA) systems. On NUMA systems, topology-aware load-balancing is based on the MPO framework, described in the next section.

4.2.3 *Topology and locality awareness.* In addition to priority, other conditions and configuration parameters factor into scheduling decisions on Solaris. A good example of this is the Memory Placement Optimization (MPO) framework [Chew 2006]. MPO mitigates the effects of systems with non-uniform memory access latencies by scheduling threads onto processors that are close to the thread's allocated physical memory. MPO uses the abstraction of *locality groups (lgroups)* that represent the set of CPU-like and memory-like hardware resources which are within some latency of each other. A hierarchy of lgroups is used to represent multiple levels of locality. On Uniform Memory Access (UMA) machines there is only one level of locality represented by one lgroup since the latency between all CPUs and all memory is the same. On NUMA systems there are at least two levels of locality. On such systems the child lgroups capture CPUs and memory within same latency of each other while the root (parent) lgroup contains CPUs and memory within remote latency (e.g., all CPUs and memory).

Solaris uses lgroups to enforce topology awareness in the following way:

—Each thread is assigned a home lgroup upon creation.

—The kernel always tries to get memory and processing resources from its home locality group and then, if this fails, from the parent lgroup(s).

—The load balancing is performed across CPUs in the same lgroup.

5. CONCLUSIONS AND FUTURE DIRECTIONS OF SCHEDULING RESEARCH

The advent of the multicore era has compelled us to reconsider the role of scheduling. On the one hand, multicore architectures introduce tightly coupled resources sharing, which the scheduler can manage. On the other hand, abundance of cores makes one wonder: is thread scheduling as important as it used to be in the past, when CPU resources were scarce and expensive? Indeed, with the growing abundance of cores on a single chip, we may be able to assign a dedicated core to each thread, and then there is not much left for the scheduler to do; at least not for the traditional time-sharing schedulers. However, as the research surveyed in this work shows shared-resource-aware schedulers can play a tremendous role in this situation by selecting thread-to-core mappings that either facilitate cooperative inter-thread data sharing or mitigate shared resource contention between competing threads. Dealing with shared resources in CMPs is just one of the many roles future schedulers will take on.

In the rest of this section we look ahead at what scheduling challenges lie just beyond the horizon as well as highlight some of the most recent research on future scheduling architectures.

While schedulers have traditionally managed individual threads, in the future schedulers will have to manage *application containers*. The drive towards increased parallelism and the advent of new parallel programming environments, as well as"modernization" of the old ones, will create a large class of applications that "schedule themselves". In particular, applications will run on top of runtimes that have their own thread or task schedulers. Many established parallel programming frameworks already perform their own thread scheduling (OpenMP [Chandra et al. 2001], MPI, Cilk [Blumofe et al. 1995]), and new parallel libraries certainly follow suit (e.g., Intel TBB [Reinders 2007], CnC [Knobe 2009], Serialization Sets [Allen et al. 2009], Cascade [Best et al. 2011], Galois [Kulkarni et al. 2007], Dryad [Isard et al. 2007], job managers in commercial game engines [Leonard 2007]).

We have already seen a trend that is somewhat similar to management of application containers in the virtual machine domain, where a hypervisor manages virtual machines as independent entities. It assigns to them a fixed amount of CPU and memory resources, which the VM then manages on its own. A similar kind of framework will have to be adopted at the level of individual applications. Tessellation [Colmenares et al. 2010] and ROS [Klues et al. 2010] are two emerging scheduling infrastructures whose primary schedulable entities are application containers.

For example [Colmenares et al. 2010] proposed the Cell Model as the basis for scheduling. A Cell is a container for parallel applications that provides guaranteed access to resources, i.e., the performance and behavior close to an isolated machine. Resources are guaranteed as space-time quantities, such as "4 processors for 10% of the time" or "2 GB/sec of bandwidth". Although Cells may be time-multiplexed, hardware thread contexts and resources are gang-scheduled such that Cells are unaware of this multiplexing. Resources allocated to a Cell are owned by that Cell until explicitly revoked. Once the Cell is mapped, a user-level scheduler is responsible for scheduling hardware contexts and other resources within Cells. Inter-cell communication is possible and occurs through special channels.

If the future is about managing application containers, then what must be done to manage these containers effectively? The first and obvious challenge is to slice machine resources such that the demands of individual containers are satisfied, while avoiding resource waste and internal fragmentation. The need to satisfy resource demands of multiple application containers implies the need to perform some kind of admission control, which means that an application must be able to assess its resource requirements *a priori* and present them to the scheduler. Building such applications is going to be a challenge.

As researchers from ETH Zurich and Microsoft Research point out in their position paper *Design Principles for End-to-End Multicore Schedulers* [Peter et al. 2010], future systems will run a very dynamic mix of interactive real-time applications. Satisfying performance goals of these applications will demand new schedulers.

In traditional scheduling infrastructures, the way for an application to ask for more resources was to increase its priority. This is a very coarse and inexpressive mechanism that does nothing to guarantee that the application will meet its quality-of-service goals. While some of the recent research attempted to move beyond this simplistic mechanism and suggested that an application should inform the scheduler of its target instruction throughput, we believe that this mechanism is still very rudimentary. First of all, instruction throughput is a machine-dependent metric, so an application attempting to specify its target throughput will not be portable across a large number of systems. Second, it is very difficult for the programmer to determine the target throughput, because an average programmer is not trained to understand performance at such a fine level of detail. Finally, if performance of the application is limited by bottlenecks other than instruction throughput (for example if the application is I/O- or synchronization-bound), specifying performance goals in terms of instructions becomes useless.

We believe that the scheduler should demand from the application no more than to specify its QoS goals in terms of high-level metrics that make sense for the application developer (e.g., frames per second, response time or transactions per second). The scheduler would then attempt to satisfy the application performance goal by monitoring how its actual performance deviates from the target and by providing to the application more of a bottleneck resource if its performance is below the target.

Putting everything together both from the works surveyed and looking forward, this is how we view the *ideal* scheduler in future systems: it is a scheduler that balances system resources among application containers while satisfying applications' individual performance goals and at the same time considering global system constraints such as thermal envelope and power budget. If we are to succeed in designing such a scheduler, we need to address not only the scheduling algorithms themselves, but also design new interfaces between application runtimes and the scheduler, create better performance monitoring infrastructures (both in hardware and in software) that will permit better "observability" of what is happening inside the system, as well as create better mechanisms for fine-grained resource allocation in hardware. Addressing these problems will require inter-disciplinary effort of operating system designers, hardware architects and application developers.

REFERENCES

Albonesi, D. H. 1999. Selective cache ways: on-demand cache resource allocation. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. 248–259.

Allen, M. D., Sridharan, S., and Sohi, G. S. 2009. Serialization Sets: a Dynamic Dependence-based Parallel Execution Model. In *Proc. of PPoPP '09*. 85–96.

Awasthi, M., Sudan, K., Balasubramonian, R., and Carter, J. 2009. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *HPCA 2009. IEEE 15th International Symposium on High Performance Computer Architecture, 2009.* 250 –261.

Azimi, R., Soares, L., Stumm, M., Walsh, T., and Brown, A. D. 2007. Path: page access tracking to improve memory management. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*. 31–42.

Azimi, R., Tam, D. K., Soares, L., and Stumm, M. 2009. Enhancing operating system support for multicore processors by using hardware performance monitoring. *SIGOPS Oper. Syst. Rev. 43,* 2, 56–65.

Balasubramonian, R., Albonesi, D., Buyuktosunoglu, A., and Dwarkadas, S. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 245–257.

Banikazemi, M., Poff, D., and Abali, B. 2008. Pam: a novel performance/power aware meta-scheduler for multi-core systems. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. 1–12.

Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 29–44.

Berg, E. and Hagersten, E. 2004. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*. 20–27.

Best, M. J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., and Brownsword, A. 2011. Synchronization via scheduling: Techniques for efficiently managing shared state in video games. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*.

Bitirgen, R., Ipek, E., and Martinez, J. F. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. 318–329.

Blagodurov, S. and Fedorova, A. 2011. User-level scheduling on NUMA multicore systems under Linux. In *Proceedings of the 13th Annual Linux Symposium*.

Blagodurov, S., Zhuravlev, S., Dashti, M., and Fedorova, A. 2011. A case for NUMA-aware contention management on multicore processors. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*. 207–216.

Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. 2008. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. 43–57.

Burger, D., Goodman, J. R., and Kägi, A. 1996. Memory bandwidth limitations of future microprocessors. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*. 78–89.

Cascaval, C., Rose, L. D., Padua, D. A., and Reed, D. A. 2000. Compile-time based performance prediction. In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*. 365–379.

Chandra, D., Guo, F., Kim, S., and Solihin, Y. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. 340–351.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. 2001. *Parallel programming in OpenMP*. EuroPar'09.

Chang, J. and Sohi, G. S. 2007. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. 242–252.

Chaudhuri, M. 2009. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA 2009. IEEE 15th International Symposium on High Performance Computer Architecture, 2009*. 227 –238.

Chen, R., Chen, H., and Zang, B. 2010. Tiled MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling. In *Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT 2010)*. Vienna, Austria.

Chew, J. 2006. *Memory Placement Optimization (MPO)*.

Chishti, Z., Powell, M. D., and Vijaykumar, T. N. 2005. Optimizing replication, communication, and capacity allocation in CMPs. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. 357–368.

Cho, S. and Jin, L. 2006a. Managing distributed, shared l2 caches through OS-level page allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 455–468.

Cho, S. and Jin, L. 2006b. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 455–468.

Colmenares, J. A., Bird, S., Cook, H., Pearce, P., Zhu, D., Shalf, J., Hofmeyr, S., Asanovic', K., and Kubiatowicz, J. 2010. Resource management in the tessellation manycore OS. In *Poster session at 2nd USENIX Workshop on Hot Topics in Parallelism*.

Delaluz, V., Sivasubramaniam, A., Kandemir, M., Vijaykrishnan, N., and Irwin, M. J. 2002. Scheduler-based DRAM energy management. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*. 697–702.

Denning, P. J. 1968. The working set model for program behavior. *Commun. ACM 11*, 323–333.

Dybdahl, H. and Stenstrom, P. 2007. An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 2–12.

Fedorova, A., Seltzer, M., and Smith, M. D. 2007. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. 25–38.

Gordon-Ross, A., Viana, P., Vahid, F., Najjar, W., and Barros, E. 2007. A one-shot configurable-cache tuner for improved energy and performance. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*. 755–760.

Guo, F., Kannan, H., Zhao, L., Illikkal, R., Iyer, R., Newell, D., Solihin, Y., and Kozyrakis, C. 2007. From chaos to QoS: case studies in CMP resource management. *SIGARCH Comput. Archit. News 35,* 1, 21–30.

Guo, F. and Solihin, Y. 2006. An analytical model for cache replacement policy performance. In *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*. 228–239.

Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A. 2009. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th annual international symposium on Computer architecture*. ISCA '09. 184–195.

Hofmeyr, S., Iancu, C., and Blagojević, F. 2010. *Load balancing on speed*. ACM.

Hoste, K. and Eeckhout, L. 2007. Microarchitecture-independent workload characterization. *IEEE Micro 27,* 3, 63–72.

Hsu, L. R., Reinhardt, S. K., Iyer, R., and Makineni, S. 2006. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. 13–22.

Hur, I. and Lin, C. 2004. Adaptive history-based memory schedulers. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. 343–354.

Ipek, E., Mutlu, O., Martínez, J. F., and Caruana, R. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. 39–50.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

Iyer, R. 2004. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*. 257–266.

Iyer, R., Zhao, L., Guo, F., Illikkal, R., Makineni, S., Newell, D., Solihin, Y., Hsu, L., and Reinhardt, S. 2007. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 25–36.

Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely, Jr., S., and Emer, J. 2008. Adaptive insertion policies for managing shared caches. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 208–219.

Jiang, X., Mishra, A. K., Zhao, L., Iyer, R., Fang, Z., Srinivasan, S., Makineni, S., Brett, P., and Das, C. R. 2011. Access: Smart scheduling for asymmetric cache cmps. In *HPCA*. 527–538.

Jiang, Y., Shen, X., Chen, J., and Tripathi, R. 2008. 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*. 220–229.

Kamali, A. 2010. Sharing Aware Scheduling on Multicore Systems. M.S. thesis, Simon Fraser University, Burnaby, BC, Canada.

Kim, C., Burger, D., and Keckler, S. W. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ASPLOS-X. 211–222.

Kim, S., Chandra, D., and Solihin, Y. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. 111–122.

Kim, Y., Han, D., Mutlu, O., and Harchol-Balter, M. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. 1 –12.

Klues, K., Rhoden, B., Waterman, A., Zhu, D., and Brewer, E. 2010. Processes and resource management in a scalable many-core OS. In *Poster session at 2nd USENIX Workshop on Hot Topics in Parallelism*.

Knauerhase, R., Brett, P., Hohlt, B., Li, T., and Hahn, S. 2008. Using OS observations to improve performance in multicore systems. *IEEE Micro 28,* 3, 54–66.

Knobe. 2009. Ease of use with concurrent collections (CnC). In *HotPar '09*.

Kondo, M., Sasaki, H., and Nakamura, H. 2007. Improving fairness, throughput and energy-efficiency on a chip multiprocessor through DVFS. *SIGARCH Comput. Archit. News 35,* 1, 31–38.

Kotera, I., Egawa, R., Takizawa, H., and Kobayashi, H. 2007. A power-aware shared cache mechanism based on locality assessment of memory reference for CMPs. In *MEDEA '07: Proceedings of the 2007 workshop on MEmory performance*. 113–120.

Kotera, I., Egawa, R., Takizawa, H., and Kobayashi, H. 2008. Modeling of cache access behavior based on zipf's law. In *MEDEA '08: Proceedings of the 9th workshop on MEmory performance*. 9–15.

Koukis, E. and Koziris, N. 2005. Memory bandwidth aware scheduling for SMP cluster nodes. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. 187–196.

Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. 2007. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. 211–222.

Kumar, K., Vengerov, D., Fedorova, A., and Kalogeraki, V. 2011. FACT: a framework for adaptive contention-aware thread migrations. In *ACM International Conference on Computing Frontiers (CF'11)*.

Lee, R., Ding, X., Chen, F., Lu, Q., and Zhang, X. 2009. Mcc-db: minimizing cache conflicts in multi-core processors for databases. *Proc. VLDB Endow. 2,* 1, 373–384.

Leonard. 2007. Dragged Kicking and Screaming: Source Multicore. *Game Developers Conference*.

Li, T., Baumberger, D., and Koufaty et al., D. A. 2007. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proc. of SC '07*.

Liang, Y. and Mitra, T. 2008a. Cache modeling in probabilistic execution time analysis. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*. 319–324.

Liang, Y. and Mitra, T. 2008b. Static analysis for fast and accurate design space exploration of caches. In *CODES+ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*. 103–108.

Liedtke, J., Haertig, H., and Hohmuth, M. 1997. OS-controlled cache predictability for real-time systems. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*. 213.

Lin, J., Lu, Q., Ding, X., Zhang, Z., and Zhang, X. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems.

Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X., and Sadayappan, P. 2009. Enabling software management for multicore caches with a lightweight hardware support. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–12.

Liu, C., Sivasubramaniam, A., and Kandemir, M. 2004. Organizing the last line of defense before hitting the memory wall for CMPs. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*. 176.

Loh, G. H. 2008. 3d-stacked memory architectures for multi-core processors. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 453–464.

Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 78–117.

McGregor, R. L., Antonopoulos, C. D., and Nikolopoulos, D. S. 2005. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*. 28.1.

Merkel, A., Stoess, J., and Bellosa, F. 2010. Resource-conscious scheduling for energy efficiency on multicore processors. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*. 153–166.

Moreto, M., Cazorla, F. J., Ramirez, A., Sakellariou, R., and Valero, M. 2009. Flexdcp: a QoS framework for CMP architectures. *SIGOPS Oper. Syst. Rev. 43,* 2, 86–96.

Moscibroda, T. and Mutlu, O. 2007. Memory performance attacks: denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. 18:1–18:18.

Mutlu, O. and Moscibroda, T. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 146–160.

Mutlu, O. and Moscibroda, T. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. 63–74.

Nesbit, K. J., Aggarwal, N., Laudon, J., and Smith, J. E. 2006. Fair queuing memory systems. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 208–222.

Nesbit, K. J., Laudon, J., and Smith, J. E. 2007. Virtual private caches. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. 57–68.

Pesterev, A., Zeldovich, N., and Morris, R. T. 2010. Locating cache performance bottlenecks using data profiling. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*. 335–348.

Peter, S., Schupbach, A., Barham, P., Baumann, A., Isaacs, R., Harris, T., and Roscoe, T. 2010. Design principles for end-to-end multicore schedulers. In *2nd USENIX Workshop on Hot Topics in Parallelism*.

Qureshi, M. K., Lynch, D. N., Mutlu, O., and Patt, Y. N. 2006. A case for MLP-aware cache replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 167–178.

Qureshi, M. K. and Patt, Y. N. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 423–432.

Rafique, N., Lim, W.-T., and Thottethodi, M. 2006. Architectural support for operating system-driven CMP cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. 2–12.

Reddy, R. and Petrov, P. 2007. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. 198–207.

Reinders, J. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly. Asplos'10, EuroPar'09.

Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D. 2000. Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. 128–138.

Shi, X., Su, F., Peir, J.-k., Xia, Y., and Yang, Z. 2007. CMP cache performance projection: accessibility vs. capacity. *SIGARCH Comput. Archit. News 35,* 1, 13–20.

Snavely, A., Tullsen, D. M., and Voelker, G. 2002. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 66–76.

Soares, L., Tam, D., and Stumm, M. 2008. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. 258–269.

Srikantaiah, S., Das, R., Mishra, A. K., Das, C. R., and Kandemir, M. 2009. A case for integrated processor-cache partitioning in chip multiprocessors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–12.

Srikantaiah, S., Kandemir, M., and Irwin, M. J. 2008. Adaptive set pinning: managing shared caches in chip multiprocessors. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. 135–144.

Stone, H. S., Turek, J., and Wolf, J. L. 1992. Optimal partitioning of cache memory. *IEEE Trans. Comput. 41,* 9, 1054–1068.

Suh, G. E., Devadas, S., and Rudolph, L. 2002. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. 117.

Suh, G. E., Rudolph, L., and Devadas, S. 2004. Dynamic partitioning of shared cache memory. *J. Supercomput. 28,* 1, 7–26.

Tam, D., Azimi, R., and Stumm, M. 2007. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys '07)*.

Tam, D. K., Azimi, R., Soares, L. B., and Stumm, M. 2009. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. 121–132.

Thekkath, R. and Eggers, S. J. 1994. Impact of Sharing-based Thread Placement on Multithreaded Architectures. In *ISCA '94: Proceedings of the 21st annual international symposium on Computer architecture*. 176–186.

Tian, K., Jiang, Y., and Shen, X. 2009. 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*. 41–50.

Viana, P., Gordon-Ross, A., Barros, E., and Vahid, F. 2008. A table-based method for single-pass cache optimization. In *GLSVLSI '08: Proceedings of the 18th ACM Great Lakes symposium on VLSI*. 71–76.

Wang, S. and Wang, L. 2006. Thread-associative memory for multicore and multithreaded computing. In *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design*. 139–142.

Weinberg, J. and Snavely, A. E. 2008. Accurate memory signatures and synthetic address traces for HPC applications. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. 36–45.

Xie, Y. and Loh, G. 2008. Dynamic Classification of Program Memory Behaviors in CMPs. In *Proc. of CMP-MSI, held in conjunction with ISCA-35*.

Xie, Y. and Loh, G. H. 2009. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. 174–183.

Zhang, E. Z., Jiang, Y., and Shen, X. 2010. Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs? In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 203–212.

Zhang, X., Dwarkadas, S., and Shen, K. 2009. Towards practical page coloring-based multicore cache management. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*. 89–102.

Zhao, L., Iyer, R., Illikkal, R., Moses, J., Makineni, S., and Newell, D. 2007. Cachescouts: Fine-grain monitoring of shared caches in CMP platforms. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. 339–352.

Zhong, Y., Shen, X., and Ding, C. 2009. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst. 31,* 6, 1–39.

Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. 2004. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. 177–188.

Zhuravlev, S., Blagodurov, S., and Fedorova, A. 2010. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. 129–142.

...