Design Principles for End-to-End Multicore Schedulers

Simon Peter[⋆], Adrian Schüpbach[⋆], Paul Barham[†], Andrew Baumann[⋆],
Rebecca Isaacs[†], Tim Harris[†], Timothy Roscoe[⋆]
[⋆]*Systems Group, ETH Zurich* [†] *Microsoft Research, Cambridge*

Abstract

As personal computing devices become increasingly parallel multiprocessors, the requirements for operating system schedulers change considerably. Future general-purpose machines will need to handle a dynamic, bursty, and interactive mix of parallel programs sharing a heterogeneous multicore machine. We argue that a key challenge for such machines is rethinking scheduling as an end-to-end problem integrating components from the hardware and kernel up to the programming language runtimes and applications themselves.

We present several design principles for future OS schedulers, and discuss the implications of each for OS and runtime interfaces and structure. We illustrate the implementation challenges that result by describing the concrete choices we have made in the Barrelfish multikernel. This allows us to present one coherent scheduling design for an entire multicore machine, while at the same time drawing conclusions we think are applicable to the design of any general-purpose multicore OS.

1 Introduction

We argue that the key challenge of effective OS scheduling for tomorrow's multicore computers is not well addressed by existing work.

The problem is recent: in the past, highly-parallel machines were generally the domain of high-performance computing. Applications had long run times, and generally either had the machine to themselves or ran in static partitions; the OS was often more of a nuisance than an essential part of the system. In contrast, we believe that future general-purpose machines will need to handle a dynamic mix of parallel programs with interactive and/or real-time response times.

Much excellent work has examined scheduling algorithms for multicores, but this has typically been in the context of conventional models of OS structure that themselves may be unable to handle machines with hundreds of heterogeneous cores.

On the other hand, work which has proposed new OS architectures has appealed to high-level paradigms for scheduling (such as co-scheduling applications on partitions of a machine), but with little attention so far to the feasible algorithms, or the implementation problems to which such architectures lead.

In this paper we step back and take a high-level, principled view of scheduling for multicore. Rather than looking at specific algorithms, we are interested here in the implications of OS interfaces and structure (and, by extension, the design of concurrent language runtimes) for the kinds of scheduling that are possible in a multicore OS, and the implementation challenges that result.

We distill our current thinking on multicore OS scheduling into a set of design principles. While some of these are not novel in themselves, the contribution of this paper is to describe the implications of these principles for the design of any OS which aims to effectively schedule a dynamic, bursty mix of parallel applications on a heterogeneous multicore machine.

We illustrate the implications using the implementation choices we have made in the Barrelfish OS [4], including several novel features: *phase-locked scheduling*, the *system knowledge base*, *scheduling manifests*, and *dispatcher groups*. While this is one point in the design space, it allows us to present a coherent scheduling design for the whole OS. Moreover, other proposals for multicore operating systems sufficiently resemble the multikernel model for the ideas to be widely applicable.

2 Background

We start by observing that the trend towards multicore systems means that commodity hardware will become highly parallel, and mainstream applications will increasingly exploit this hardware. However, most work on parallel computing to date has been in the field of high-performance computing (HPC). Our argument is motivated by three aspects of parallel computing which have either been absent, or addressed very differently, in the techniques employed in HPC and related fields.

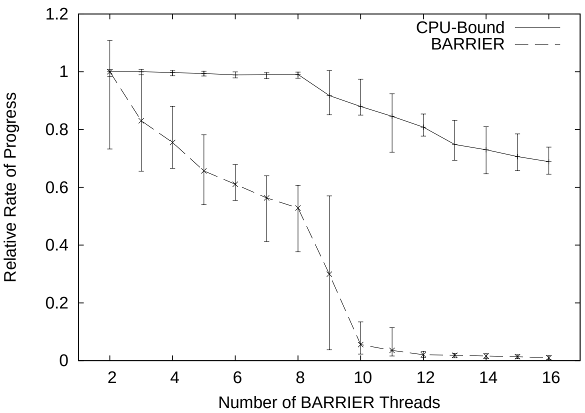

Figure 1: *Relative progress for 2 concurrent OpenMP jobs on a 16-core system. We show mean, minimum, and maximum observed over 20 runs of 15 seconds each.*

System diversity The systems on which mainstream parallel applications will run are increasingly diverse but performance is often highly sensitive to hardware characteristics like cache structure [30]. Manual tuning for a given machine is no longer an option in the market.

The HPC community has long used *autotuners* like ATLAS [28] to effectively specialize code for a specific hardware platform. However, their usefulness in a desktop scenario is limited to the subset of applications amenable to offline analysis [20].

Online adaptation is made easier by models like OpenMP [18], Grand Central Dispatch [2], ConcRT [16] and MPI [15] which make the communication and synchronization aspects of a program explicit. Such runtimes can improve performance by e.g. dynamically choosing the best thread count for a parallel code section, or accounting for hardware characteristics like whether two cores are on the same package.

Such heuristics can work well for batch-oriented HPC jobs on a small range of machines, but may not work across all architectures, and may not be suited for more complex interactive, multi-phase applications which mix parallel and sequential components. They may not even reflect all program requirements, such as a need for some threads to be scheduled simultaneously [8, 19].

Multiple applications HPC workloads have generally enjoyed exclusive use of hardware, or a static partition thereof. However, in a general-purpose system multiple simultaneous parallel programs can interfere significantly.

Figure 1 shows two OpenMP applications interfering on a 16-core Linux computer. A CPU-bound program competes with a synchronization-intensive program, which uses a BARRIER directive in a tight loop enclosed in a PARALLEL directive, causing all threads to wait on the barrier before the next iteration. We measure the progress of each by the number of loop iterations executed, and vary the total number of threads demanded by the BARRIER program from 2 to 16. The CPU-bound application always uses 8 threads.

When there are fewer *total* threads than cores, BARRIER's progress depends both on the number of threads and how the OS places them on cores: a barrier between threads on the same package costs about half as much as one between different packages. This accounts for the high performance variance for BARRIER with low thread counts, even though enough cores are available to schedule all threads simultaneously. When the applications contend for cores their performance degrades unequally: the CPU-bound process slows down linearly, but BARRIER's progress rate collapses since preemption of any thread can cause synchronization to take an order of magnitude longer.

This simple study shows the pitfalls of scheduling a mix of workloads on multicore systems, a problem which to date has not been well studied in the literature. Smart runtimes such as McRT [23] cannot solve this problem, since it is one of lack of coordination *between* runtimes.

Interactive workloads Desktop and other interactive workloads impose real-time requirements on scheduling not usually present in HPC settings. Applications often fall into one of three categories:

Firstly, there are long-running, background applications which are *elastic*: they are not sensitive to their precise resource allocation, but can make effective use of additional resources if available. One example is a background process indexing a photo library and using vision processing to identify common subjects.

In contrast, some background applications are *quality-of-service sensitive*. For example, managing the display using a hybrid CPU-GPU system should take precedence over an elastic application using GPU-like cores.

Thirdly, we must handle bursty, interactive, latency-sensitive applications such as a web browser, which may consume little CPU time while idle but must receive resources promptly when it receives user input.

Moreover, multicore programs *internally* may be more diverse than most traditional HPC applications. The various parts of a complex application may be parallelized differently – a parallel web browser [12] might use data-parallel lexing to parse a page, while using optimistic concurrency to speculatively parallelize script execution.

Traditional work on scheduling has not emphasized these kinds of workloads, where the resource demands may vary greatly over interaction-timescales ($\approx$ 10ms).

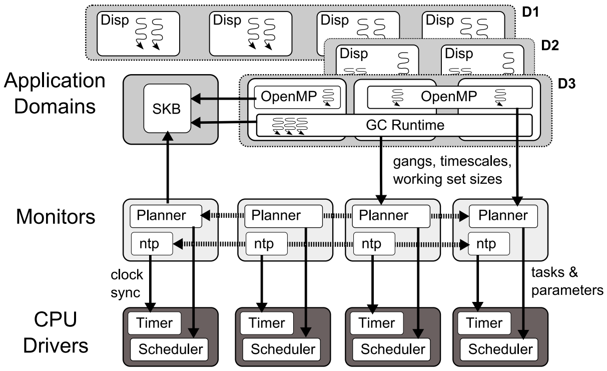

Figure 2: The Barrelfish scheduling architecture

3 Design principles

In this section we present five scheduler design principles that are important for supporting the mix of workloads we expect to find on a general-purpose multicore OS. We don't claim that the set is complete, nor that we have even necessarily picked the right way of addressing the issues at hand. However, we believe that the inevitable parallelization of software brought about by multicore raises some new and challenging issues for OS schedulers of the future, which we attempt to highlight here.

The principles are independent of particular scheduling algorithms, policies or performance metrics (throughput, energy efficiency, etc.), and are applicable to *any* viable approach which aims at addressing all the layers of the scheduling stack. Nevertheless, to explore the concrete implications of each point, we describe our ongoing implementation of the scheduling architecture for the Barrelfish multikernel OS [4], shown in Figure 2.

Barrelfish is structured as a set of low-level *CPU drivers*, each of which manages a processor core in isolation. Across the cores, a distributed system of cooperating processes known as *monitors* communicate exclusively via message passing. Barrelfish uses scheduler activations [1], and so each application runs a set of *dispatchers*, one per core, which collectively implement an application-specific user-level thread scheduler.

3.1 Time-multiplexing cores is still needed

Hardware resources will continue to be time-multiplexed, rather than using spatial partitioning on its own. We give three reasons:

First, unlike many HPC and server systems, machines will not be dedicated to individual applications. A desktop computer may run a parallel web browser alongside an application that uses a managed runtime system (such as the JVM or CLR). If parallel programming models are successful, then any of these applications could potentially exploit all of the resources of the machine.

Second, even if a machine contains a large number of cores in total, these may vary greatly in capabilities. Hill and Marty's analysis [10] suggests that a small number of "big" cores is useful to allow sequential phases of an application to execute as quickly as possible (and reduce the Amdahl's-law impact of these phases). Access to these cores will need to be time-multiplexed

Third, the burstiness of interactive workloads means that the ability to use processing resources will vary according to the user's behavior. Time-multiplexing gives a means of providing real-time QoS to these applications without needlessly limiting system-wide utilization.

Implications The need to time-multiplex cores means that resource management cannot solely operate by partitioning the resources of a machine at the point when a new application starts. The scheduling regime should be sufficiently nimble to react to rapidly changing resource demands by applications. At the same time it must reconcile short-term demands, such as by interactive applications, with coarser-grained requirements, for example a virtual machine monitor needing to gang-schedule the virtual CPUs of a multicore VM.

Implementation In Barrelfish we support different types of workload at the lowest level of CPU allocation. The scheduler in the CPU driver, see Figure 2, is based on RBED [5], a rate-based, earliest deadline first scheduling algorithm, which supports applications with and without deadlines on the same core, while remaining flexible in distributing the slack generated naturally by this scheduling technique. For example, similar to a technique known as "preemption control" in the Solaris scheduler [14], a best-effort dispatcher with a thread in a critical section may request to receive more slack in order to finish the section as quickly as possible. In combination with techniques for scheduling at different timescales that are discussed in the following section, this gives the ability to time-multiplex CPUs while still accommodating the time-sensitive requirements of some applications.

3.2 Schedule at multiple timescales

OS scheduling for multicore will involve a combination of techniques at different time granularities, much as in grid and cluster scheduling [22]. The multikernel model [4], which reflects trends in other proposed multicore operating systems [13, 17, 27], calls for designs that eschew globally-shared data in favor of decentralized communication. Due to the distributed nature of such systems, we argue that scheduling will involve:

- *long-term* placement of applications onto cores, taking into account application requirements, system load, and hardware details – this is where global optimizations and task migration decisions occur;

- *medium-term* resource reallocation, in response to unpredictable application demands, subject to long-term limits;

- *short-term* per-core (or hardware thread) scheduling, including gang-scheduled, real-time and best-effort tasks.

In contrast to prior HPC systems, in a general-purpose OS gang scheduling will need to occur over timescales typical of interactive timeslices (on the order of milliseconds). This means thread dispatch times must be much more accurately synchronized, and also that the overhead of this synchronization must be much smaller.

Implementation The scheduler inside each CPU driver on Barrelfish carries out the short-term allocation of timeslices to threads. We introduce the concept of *phase-locked schedulers* to support fine-grained gang scheduling: core-local clocks are synchronized out-of-band and schedules coordinated, so that CPU drivers locally dispatch tasks at deterministic times to ensure that gangs are co-scheduled without the need for expensive inter-core communication on every dispatch; we are extending our RBED scheduler in this manner.

Long- and medium-term scheduling decisions are made by the planner as shown in Figure 2, which is implemented as a distributed application inside user-space monitors. Planners on each node cooperate to compute coarse-grained temporal and spatial allocations, while each individual planner updates local scheduling parameters appropriately in response to application demands.

3.3 Reason online about the hardware

New processor and system architectures are appearing all the time [11]: portability as hardware evolves is as critical as portability across different processor architectures. Both OS and applications must be able to adapt well to diverse hardware environments. This requires reasoning about complex hardware, beyond what can be achieved by offline autotuners or careful platform-specific coding.

The performance of parallel software is closely tied to the structure of the hardware, and different hardware favors drastically different algorithms (for example, the performance of Dice and Shavit locks [7] depends critically on a shared cache, as opposed to other options [9, 25]). However, the appropriate choice at runtime is hard to encode in a program.

Implications Adapting scheduling policies to diverse hardware, whether across applications or among threads in a single program, requires (1) extensive, detailed information about the hardware in a usable form, and (2) a means to reason about it online in the scheduler.

Limited versions of such functionality exist, e.g. `/proc` and `/sys` on Linux, and the CPUID instruction on x86 processors. However, these APIs are ad-hoc, making it complex and non-portable to process their contents. Although performance models such as Roofline [29] and LogP [6] help by capturing some performance characteristics of available hardware, it is now time to explicitly address the broader problem.

Implementation Barrelfish combines a rich representation of the hardware in a subset of first-order logic, and a powerful reasoning engine, in a single *system knowledge base* (SKB) service [24]. The SKB is populated by hardware discovery information (e.g. cache parameters, interconnect topology), online measurement (including a detailed set of memory and cache latencies obtained at boot) and pre-asserted facts that cannot be inferred. It uses the ECL^{i}PS^{e} constraint logic programming system [3], and the OS can issue complex queries as optimization programs. While it is no magic bullet, the SKB makes it easy to express and prototype longer-timescale scheduling – a complex spatio-temporal thread scheduling algorithm involving memory and cache-affinity can be expressed in about 10 lines of Prolog which (with care) execute in less than a millisecond, and are independent of the hardware configuration.

Applications can also query the SKB. The common, expressive representation of hardware and the reasoning engine further allow the OS to present to applications raw resources described in detail (for example, specific core IDs). This facilitates an Exokernel-like approach where intra-application resource management (e.g. migrating threads to accelerators or cores with less cache contention) is handled inside the application and its libraries, rather than in the OS proper.

3.4 Reason online about each application

In addition to the need for applications to exploit the structure of the hardware on which they are running, the OS should exploit knowledge of the structure of the applications which it is scheduling. For example, gang scheduling can eliminate the problems shown in Section 2 by avoiding pre-emption of synchronized threads. However, simple gang scheduling of all threads within applications is overly restrictive. For instance, OpenMP typically only benefits from gang scheduling threads within each team. Similarly, threads performing unrelated operations would favor throughput (allocation of as much time to all threads as possible) over contemporaneous execution. Finally, a single application may consist of different phases of computation, with changing scheduling and resource requirements over its lifetime. The optimal allocation of hardware cores and memory regions thus changes over time.

Implications An application should expose as much information about its current workload and resource requirements as possible to allow the OS to effectively allocate resources. For example, MapReduce applications follow fixed data-flow phases, and it is possible to determine this information at compile-time for programming paradigms like OpenMP [26].

Implementation At startup, or during execution, Barrelfish applications may present a *scheduling manifest* to the planner, containing a specification of predicted long-term resource requirements, expressed as constrained cost-functions in an ECL^{i}PS^{e} program. A manifest may make use of any information in the SKB including hardware details as well as application properties, such as data proximity and working set size bounds. The SKB includes common functionality to express, for example, the placement of synchronization-intensive threads on closely-coupled cores. We expect that much of the information in manifests could be inferred by compile-time analyses or provided by language runtimes.

The planner uses the application's manifest along with knowledge of current hardware utilization to determine a suitable set of hardware resources for the application, which may then create dispatchers and negotiate appropriate scheduler parameters on those cores (as described in the following section). In general, an application is free to create a dispatcher on any core, however only by negotiating with the planner will it receive more than best-effort resources.

3.5 Application and OS must communicate

The allocation of resources to applications requires renegotiation while applications are running. This can occur when a new application starts, but also as its ability to use resources changes (in an extreme example, when a sequential application starts a parallel garbage collection phase), and in response to user input or changes in the underlying hardware (such as reducing the number of active cores to remain within a power budget).

Hints from the application to the OS can be used to improve overall scheduling efficiency, but should not adversely impact other applications, violating the OS scheduler's fairness conditions.

Implications Efficient operation requires two-way information flow between applications and OS. First, applications should indicate their ability to use or relinquish resources. For instance, an OpenMP runtime would indicate if it could profitably expand the team of threads it is using, or if it could contract the team. Secondly, the OS should signal an application when new resources are allocated to it, and when existing resources are preempted. This allows the application's runtime to respond appropriately; for instance, if the number of cores was reduced, then a work-stealing system would re-distribute work items from the queue of the core being removed.

De-allocation is co-operative in the sense that an application receives a de-allocation request and is expected to relinquish use of the resources in question. If this is not done promptly then the OS virtualizes the resource to preserve correctness at the expense of performance.

Implementation Applications express short- and medium-term scheduling requirements to the OS by placing dispatchers into one or more *dispatcher groups* and then negotiating with the planner how each group is scheduled. Dispatcher groups extend the notion of RTIDs [21], describing requirements such as real-time, gang scheduling and load-balancing parameters.

Membership of dispatcher groups varies dynamically with workload. For instance, a managed runtime using parallel stop-the-world garbage collection would merge *all* its dispatchers into one group during collection, and then divide them into several groups according to the application's work once garbage collection completes.

Scheduling of dispatchers is made explicit via scheduler activations [1]. De-scheduling is made explicit by the scheduler incrementing a per-dispatcher deallocation count. This allows a language runtime to determine whether a given dispatcher is running at any instant and, for example, allows active threads to steal work items from a pre-empted thread via a lock-free queue.

4 Summary

General-purpose multicore computing faces a different set of challenges from traditional parallel programming for HPC, and exploiting existing work in scheduling in this new context requires a principled, end-to-end approach which considers all layers of the software stack.

We have outlined a set of such principles which we feel apply to any viable scheduling architecture for multicores. We have also described the consequences and innovations (phase-locked scheduling, the system knowledge base, scheduling manifests, dispatcher groups) that these lead to in the Barrelfish multikernel.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, Oct. 1991.

[2] Apple. *Grand Central Dispatch Technology Brief*, 2009.

[3] K. R. Apt and M. G. Wallace. *Constraint Logic Programming using* ECL^iPS^e. Cambridge University Press, 2007.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009.

[5] S. A. Brandt, S. A. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proc. of the 24th IEEE Real-Time Systems Symposium*, 2003.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Proc. of the 4th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1993.

[7] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *Proc. of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2009.

[8] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[9] B. He, W. N. Scherer III, and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *Proc. of the 12th International Conference on High Performance Computing*, pages 7–18, 2005.

[10] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, July 2008.

[11] Intel Corporation. Single-chip cloud computer. http://techresearch.intel.com/articles/Tera-Scale/1826.htm, December 2009.

[12] C. G. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodik. Parallelizing the web browser. In *Proc. of the 1st USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[13] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. In *Proc. of the 1st USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[14] J. Mauro. *The Solaris Process Model: Managing Thread Execution and Wait Times in the System Clock Handler*, 2000. http://developers.sun.com/solaris/articles/THREADexec.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, September 2009.

[16] Microsoft. *C++ Concurrency Runtime*, 2010. http://msdn.microsoft.com/en-us/library/dd504870.aspx.

[17] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[18] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 2008. Version 3.0.

[19] J. Ousterhout. Scheduling techniques for concurrent systems. In *IEEE Distributed Computer Systems*, 1982.

[20] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems – an experience report. In *Proc. of the 1st International Workshop on Multicore Software Engineering*, May 2008.

[21] M. Rajagopalan, B. T. Lewis, and T. A. Anderson. Thread scheduling for multi-core platforms. In *Proc. of the 11th USENIX Workshop on Hot Topics in Operating Systems*, May 2007.

[22] R. Raman, M. Livny, and M. H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proc. of the 7th IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[23] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Peterson, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large scale CMP environment. In *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Mar. 2007.

[24] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proc. of the 1st Workshop on Managed Multi-Core Systems*, 2008.

[25] M. L. Scott and W. N. Scherer III. Scalable queue-based spin locks with timeout. In *Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 44–52, 2001.

[26] Z. Wang and M. F. P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[27] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[28] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[29] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Comm. of the ACM*, 52(4):65–76, 2009.

[30] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jan. 2010.