Hardware Execution Throttling for Multi-core Resource Management *

Xiao Zhang Sandhya Dwarkadas Kai Shen
Department of Computer Science, University of Rochester
{xiao, sandhya, kshen}@cs.rochester.edu

Abstract

Modern processors provide mechanisms (such as duty-cycle modulation and cache prefetcher adjustment) to control the execution speed or resource usage efficiency of an application. Although these mechanisms were originally designed for other purposes, we argue in this paper that they can be an effective tool to support fair use of shared on-chip resources on multi-cores. Compared to existing approaches to achieve fairness (such as page coloring and CPU scheduling quantum adjustment), the execution throttling mechanisms have the advantage of providing fine-grained control with little software system change or undesirable side effect. Additionally, although execution throttling slows down some of the running applications, it does not yield any loss of overall system efficiency as long as the bottleneck resources are fully utilized. We conducted experiments with several sequential and server benchmarks. Results indicate high fairness with almost no efficiency degradation achieved by a hybrid of two execution throttling mechanisms.

1 Introduction

Modern multi-core processors may suffer from poor fairness with respect to utilizing shared on-chip resources (including the last-level on-chip cache space and the memory bandwidth). In particular, recent research efforts have shown that uncontrolled on-chip resource sharing can lead to large performance variations among co-running applications [5, 17]. Such poor performance isolation makes an application's performance hard to predict and consequently it hurts the system's ability to provide quality-of-service support. Even worse, malicious applications can take advantage of such obliviousness to on-chip resource sharing to launch denial-of-service attacks and starve other applications [10].

Much research has tried to tackle the issue of fair resource utilization on multi-core processors. Some require significant new hardware mechanisms that are not available on commodity platforms [1, 3, 14, 17]. Without extra hardware support, the operating system must resort to software techniques such as page coloring to achieve cache partitioning [4, 9, 13, 15, 16] and CPU scheduling quantum adjustment to achieve fair resource utilization [5]. However, page coloring requires significant changes in the operating system memory management, places artificial constraints on system memory allocation policies, and incurs expensive re-coloring (page copying) costs in dynamic execution environments. CPU scheduling quantum adjustment suffers from its inability to provide fine-grained quality of service guarantees.

In this paper, we argue that hardware execution throttling can efficiently manage on-chip shared resources with much less complexity and overhead than existing alternatives, while providing the necessary granularity of quality of service. Specifically, we investigate the use of existing hardware mechanisms to control the cache/bandwidth consumption of a multi-core processor. Commodity processors are deployed with mechanisms (*e.g.*, duty cycle modulation and dynamic voltage and frequency scaling) to artificially slow down execution speed for power/thermal management [7]. By throttling down the execution speed of some of the cores, we can control an application's relative resource utilization to achieve desired fairness or other quality-of-service objectives. In addition to direct throttling of CPU speed, we also explore the existing mechanism of controlling L1 and L2 cache hardware prefetchers. Different cache prefetching configurations also allow us to manage an application's relative utilization of the shared memory bandwidth and cache space.

2 Multi-core Resource Management Mechanisms

2.1 Hardware Execution Throttling

One mechanism to throttle a CPU's execution speed available in today's multi-core platforms is dynamic voltage and frequency scaling. However, on some multi-core platforms, sibling cores often need to operate at the same frequency [11]. Intel provides another mechanism to throttle per-core execution speed, namely, *duty-cycle modulation* [7]. Specifically, the operating system can specify a portion (*e.g.*, multiplier of 1/8) of regular CPU cycles as duty cycles by writing to the logical processor's IA32_CLOCK_MODULATION register. The processor is effectively halted during non-duty cycles. Duty-cycle

*This work was supported in part by the NSF grants CNS-0411127, CAREER Award CCF-0448413, CNS-0509270, CNS-0615045, CNS-0615139, CCF-0621472, CCF-0702505, and CNS-0834451; by NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01; and by several IBM Faculty Partnership Awards.

Prefetchers	Description
L1 IP	Keeps track of instruction pointer and looks for sequential load history.
L1 DCU	When detecting multiple loads from the same line within a time limit, prefetches the next line.
L2 Adjacent Line	Prefetches the adjacent line of required data.
L2 Stream	Looks at streams of data for regular patterns.

Table 1: Brief description of four L1/L2 cache prefetchers on Intel Core 2 Duo processors [7].

modulation was originally designed for thermal management and was also used to simulate an asymmetric CMP in recent work [2].

Execution throttling is not work-conserving since it leaves resources partially idle while there are still active tasks. Consequently, there is potential cause for concern about lost efficiency in the pursuit of fairness. We argue that careful execution throttling only affects the relative resource use among co-running applications. It should not degrade the overall system efficiency as long as the bottleneck resource (shared cache space or memory bandwidth) is fully utilized.

Today's commodity processors often perform *hardware prefetching*, which helps hide memory latency by taking advantage of bandwidth not being used by on-demand misses. However, in a multi-core environment, the result might be contention with the on-demand misses of concurrently executing threads. The hardware prefetchers are usually configurable. For example, on Intel Core 2 Duo processors, there are two L1 cache prefetchers (DCU and IP prefetchers) and two L2 cache prefetchers (adjacent line and stream prefetchers) [7]. Table 1 briefly describes the prefetchers on our test platform. Each can be selectively turned on/off, providing partial control over a thread's bandwidth utilization.

Both duty-cycle modulation and prefetcher adjustment can be used to throttle an application's execution speed. The former directly controls the number of accesses to the cache (the execution speed), thereby affecting cache pressure and indirectly the bandwidth usage, while the latter directly controls bandwidth usage, thereby affecting cache pressure and indirectly affecting execution speed. Adjusting the duty cycle alone might not result in sufficient bandwidth reduction if the prefetching is aggressive, while adjusting the prefetching alone might not reduce cache pressure sufficiently. Both mechanisms can be combined to arrive at fair resource allocation.

On our platform, configuring the duty cycle takes $265 + 350$ (read plus write register) cycles; configuring the prefetchers takes $298 + 2065$ (read plus write register) cycles. The control registers also specify other features in addition to our control targets, so we need to read their values before writing. The longer time for a new prefetching configuration to take effect is possibly due to clearing obsolete prefetch requests in queues. Roughly speaking, the costs of configuring duty cycle modulation and cache prefetcher are 0.2 and 0.8 microseconds respectively on our 3.0 GHz machine.

Enabling these mechanisms requires very little operating system software modification. Our changes to the Linux kernel source are $\sim$40 lines of code in a single file.

2.2 Alternative Mechanisms

Cache Partitioning Page coloring, a technique originally proposed for cache conflict mitigation [8, 12], is a software technique that manipulates mapping between memory and cache. Memory pages that are mapped to the same cache blocks are labeled to be in the same color. By manipulating the allocation of colors to applications, the operating system can partition a cache at page granularity (strictly speaking, at a granularity of $PageSize$ times $CacheAssociativity$). The maximum number of colors that a platform can support is determined by $\frac{CacheSize}{PageSize \times CacheAssociativity}$.

Page coloring has recently been used to manage cache allocation [4, 9, 13, 15] by isolating cache space usage among applications. However, page coloring has a number of important drawbacks [16]. First, during dynamic executions in multi-programmed environments, the resource manager may decide to change an application' cache share due to a priority change or a change in the set of simultaneously executing processes. This would require re-coloring of a potentially large number of memory pages with each re-coloring typically requiring an expensive page copy. As a quantitative reference, copying a single page costs around 3 microseconds on our platform, which is already much more expensive that the configuration (or re-configuration) of hardware execution throttling mentioned earlier.

The second drawback is that page coloring enforces strict memory to cache mapping and introduces artificial memory allocation constraints. For example, an application allocated one eighth of all cache colors is also entitled to only one eighth of the total memory space. This artificial memory allocation constraint may force an application to run out of its entitled memory space while many free pages are still available in other colors.

Finally, compared to hardware execution throttling, page coloring requires more significant changes in the operating system memory management code. Our incomplete implementation of page coloring (without full support for page re-coloring) involves more than 700 lines of Linux source code changes in 10 files.

In addition to software-based cache management mechanisms, several hardware-level mechanisms have also been proposed [1, 14, 17]. They generally require adding new hardware counters/tags to monitor fine-grained cache usage, and modify the cache replacement policy based on applications' resource entitlements. It is also possible to implement associativity-based cache partitioning (called column caching in [3]), which is a tradeoff between control flexibility and deployment overhead. While such hardware mechanisms could be beneficial, we focus here on mechanisms available in today's commodity platforms.

CPU Scheduling Quantum Adjustment Fedorova *et al.* proposed a software method to maintain fair resource usage on multi-cores [5]. They advocate adjusting the CPU scheduling time quantum to increase or decrease an application's relative CPU share. By compensating/penalizing applications under/over fair cache usage, the system tries to maintain equal cache miss rates across all applications (which is considered fair). To derive the fair cache miss rate, they profile an application's behavior with several different co-runners.

The key drawback of CPU scheduling quantum adjustment is that it only achieves fairness at granularities comparable to the scheduling time quantum. This would lead to unstable performance of fine-grained tasks (such as individual requests in a server system).

3 Evaluation and Results Analysis

We enabled the duty-cycle modulation and cache prefetcher adjustment mechanisms by modifying the Linux 2.6.18 kernel. Our experiments were conducted on an Intel Xeon 5160 3.0 GHz "Woodcrest" dual-core platform. The two cores share a single 4 MB L2 cache (16-way set-associative, 64-byte cache line, 14 cycle latency, writeback).

Our evaluation benchmarks include three programs from SPECCPU2000: swim, mcf, and equake. We also employ two server-style benchmarks (SPECjbb2005 and SPECweb99) in our evaluation. SPECjbb is configured with four warehouses and a 500 MB heap size. SPECweb is hosted on the Apache web server 1.3.33. When running alone, swim, mcf, and equake take 136.1, 46.1, and 67.5 seconds respectively to complete. We bind each server application to a single core to get its baseline performance. SPECjbb delivers a throughput of 17794.4 operations/second and SPECweb delivers a throughput of 361.5 web requests/second.

Optimization Goal and Policy Settings We measure several approaches' ability to achieve fairness and, in addition, evaluate their efficiency. There are several possible definitions of fair use of shared resources [6]. The particular choice of fairness measure should not affect the main purpose of our evaluation. In our evaluation, we use *communist fairness*, or equal performance degradation compared to a standalone run for the application. Based on this fairness goal, we define an *unfairness factor* metric as the coefficient of variation (standard deviation divided by the mean) of all applications' performance normalized to that of their individual standalone run. We also define an *overall system efficiency* metric as the geometric mean of all applications' normalized performance.

We consider two execution throttling approaches. One is based on the per-core duty cycle modulation. Another is a hybrid approach that employs both duty cycle modulation and cache prefetcher adjustment. We implement two additional approaches in the Linux kernel for the purpose of comparison: an ideal page coloring approach (one that uses a statically defined cache partition point and incurs no page recoloring cost) and scheduling quantum adjustment using an idle process to control the amount of CPU time allocated to the application process. As a base for comparison, we also consider *default sharing*—running two applications on a dual-core processor under the default hardware/software resource management.

For each approach other than default sharing, we manually try all possible policy decisions (*i.e.*, page coloring partitioning point, duty cycle modulation ratio, cache prefetcher configuration, and idle process running time) and report the result for the policy decision yielding the best fairness. Since the parameter search space when combining duty cycle modulation and prefetcher configuration is large, we explore it in a genetic fashion. Specifically, we first select default and a few duty cycle modulation settings that achieve reasonably good fairness and then tune their prefetchers to find a best configuration. In most cases, duty-cycle modulation and duty-cycle & prefetch reach the same duty-cycle ratio except for {swim, SPECjbb}. In this case, setting swim's duty-cycle to 5/8 has a similar throttling effect to disabling its L2 stream prefetcher.

Figure 1 illustrates the page coloring-based cache partition settings yielding the best fairness. Table 2 lists the best-fairness policy settings for the hybrid hardware throttling (duty cycle modulation and cache prefetcher configuration) and scheduling quantum adjustment respectively. All cache prefetchers on our platform are per-core configurable except the L2 adjacent line prefetcher.

Best Fairness Figure 2 shows the fairness results (in terms of the unfairness factor) when running each possible application pair on the two cores (running two instances of the same application shows an unfairness factor close to 0 in all cases, so we do not present these results in the figure). On average, the unfairness factor is 0.191, 0.028, 0.025, 0.027, and 0.017 for default sharing, page coloring, scheduling quantum adjustment, duty-cycle modulation, and duty-cycle & prefetch, respectively. Default sharing demonstrates a higher unfairness factor in several cases. The level of unfairness is a function of the properties of the co-running applications. If their cache and bandwidth usage requirements are similar, the unfairness factor is low. If the requirements are

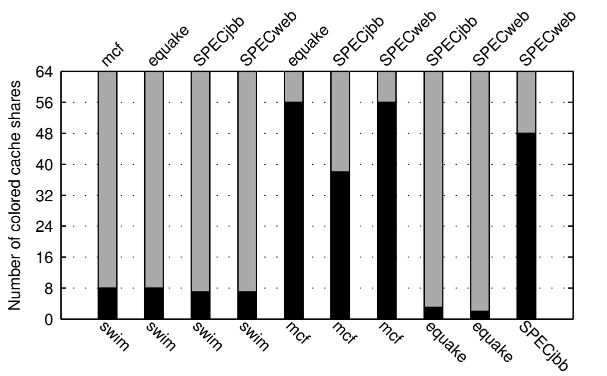

Figure 1: Cache partition settings under page coloring to achieve best fairness. Our experimental platform supports at most 64 colors and therefore the shared 4 MB L2 cache is divided into 64 shares. Results are shown for all possible application pairs from our five benchmarks (pairing an application with itself results in the cache being partitioned in half).

significantly different, and if the sum of the requirements exceeds the available resource, the unfairness factor is high due to uncontrolled usage.

Ideal page coloring-based cache partitioning also shows some variation in the unfairness factor across benchmark pairs. In particular, {swim, SPECweb} shows a comparatively higher unfairness factor due to two competing effects. Under page coloring, if swim was entitled to a very small portion of the cache space, its mapped memory pages might be less than its required memory footprint, resulting in thrashing (page swapping to disk). If swim's cache usage is not curtailed, SPECweb's normalized performance is significantly affected. These competing constraints result in page coloring not achieving good fairness (overall efficiency is also lower than with default sharing) in this case.

While both page coloring and execution throttling achieve better fairness than default sharing, the combination of duty cycle modulation and prefetching control achieves a uniformly low unfairness factor below 0.03. This uniform fairness is achieved without the additional (not accounted for; we elaborate further later in this section) overheads of page coloring. One can extend these fairness goals to additional management objectives like proportional resource allocation.

The scheduling quantum adjustment obtains similar low unfairness factor to hardware throttling. However, these results are calculated based on coarse-grained performance measurement (*i.e.*, at the scale of whole application execution). When examined at finer granularity, performance fluctuates (see Figure 4; we elaborate further later in this section), suggesting unstable fairness.

	Hardware throttling		Scheduling
Co-running applications	Duty-cycle modulation	Non-default cache prefetcher setup	quantum adjustment
swim	5/8	Default	100/30
mcf	Default	Default	NA
swim	7/8	Default	100/20
equake	Default	Enable L1 DCU	NA
swim	Default	Disable L2 stream	100/40
SPECjbb	Default	Default	NA
swim	6/8	Default	100/30
SPECweb	Default	Default	NA
mcf	Default	Disable L2 adjacent line	NA
equake	6/8	Disable L2 adjacent line	100/25
mcf	Default	Default	NA
SPECjbb	Default	Default	NA
mcf	Default	Disable L2 adj. & stream	100/5
SPECweb	Default	Disable L2 adjacent line	NA
equake	6/8	Enable L1 DCU	100/30
SPECjbb	Default	Enable L1 DCU	NA
equake	7/8	Default	100/30
SPECweb	Default	Default	NA
SPECjbb	Default	Disable L2 stream	NA
SPECweb	Default	Default	NA

Table 2: Configurations of hardware throttling and scheduling quantum adjustment to achieve best fairness. The duty-cycle modulation must be a multiplier of 1/8 on our platform. The default hardware throttling configuration is full execution speed (or 8/8), plus enabled L1 IP, disabled L1 DCU, enabled L2 adjacent line, and enabled L2 stream prefetchers. The scheduling quantum adjustment adds an idle process to squeeze one's CPU share. For example, "100/30" means every round, the application and idle process alternate, running for 100 and 30 milliseconds, respectively. "NA" implies no idle process was used. Results are shown for all possible application pairs from our five benchmarks (pairing an application with itself results in the use of default configurations).

Efficiency At Best Fairness Figure 3 shows evaluation results on the overall system efficiency (when the best fairness is achieved under each approach). Here we also include the efficiency results of running two identical applications. Note that fairness can be trivially achieved for these cases by all mechanisms (*i.e.*, equal cache partitioning under page coloring, equal setups on both cores for execution throttling and prefetching, no scheduling quantum adjustment). However, as the results demonstrate, system efficiency for some of the approaches varies. Note that for the prefetcher adjustment, we may choose a (identical) non-default prefetcher setting for efficiency gain. Specifically, we do so in two instances: for {mcf, mcf}, we enable the L1 DCU prefetcher and disable the L2 adjacent line prefetcher; for {SPECjbb, SPECjbb}, we enable the L1 DCU prefetcher and disable the L2 stream prefetcher. On average, the efficiency of all approaches is similar (roughly 0.65). Specific cases where significant differences occur are discussed below.

For {mcf, equake} and {equake, SPECjbb}, equake aggressively accesses the L2 cache and makes its co-runner suffer intensive cache conflicts. Our miss ratio profile shows that equake is not cache space sensitive, demonstrating only a 2% increase in cache miss ratio when varying the L2 cache space from 4 MB to 512 KB.

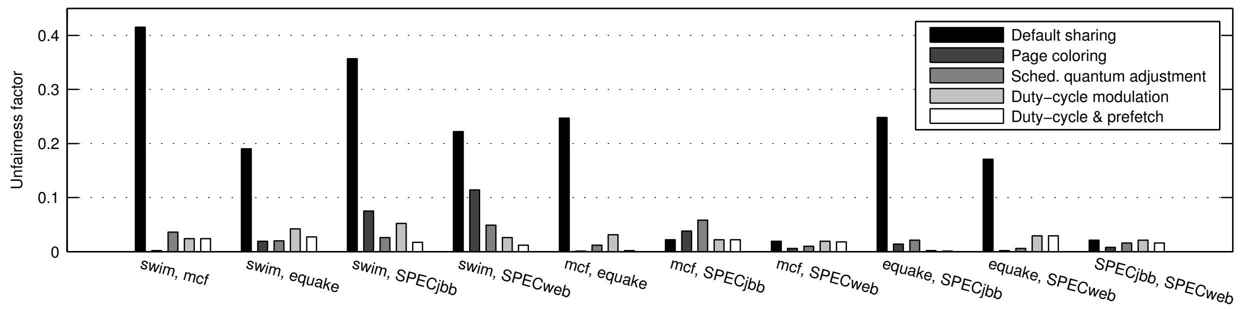

Figure 2: Comparison of the unfairness factor (the lower the better).

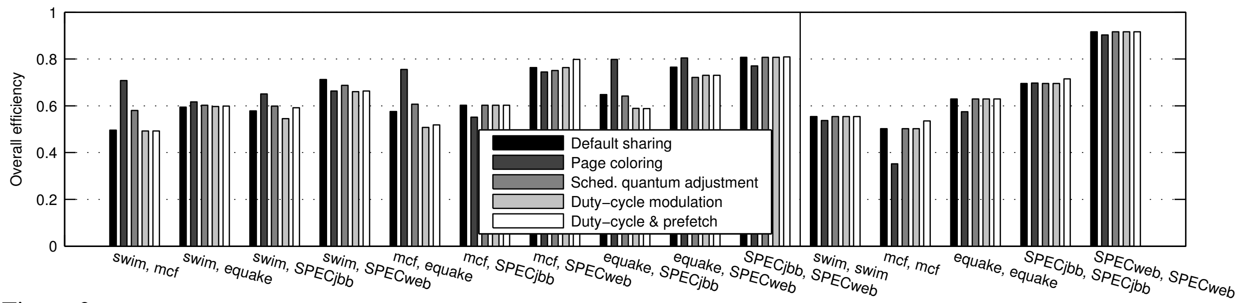

Figure 3: Comparison of the overall system efficiency when each approach is optimized for best fairness. We also include results of running two identical applications.

By constraining equake to a small portion of the L2 cache, page coloring can effectively prevent pollution of the co-runner's cached data without hurting equake's performance. Hardware throttling approaches do not fundamentally solve inter-application cache conflicts and need to slow down equake's execution dramatically to achieve "fair" cache sharing. In these cases, hardware throttling has roughly 10% efficiency degradation while page coloring improves efficiency by 23$\sim$30% relative to default sharing. The scheduling quantum adjustment also achieves better efficiency than hardware throttling in these two cases. This is because equake is less vulnerable to inter-application cache conflicts than the other application. By introducing an idle process to reduce equake's co-running time with the other application, it greatly mitigates the negative cache conflict impact on the other application and therefore boosts overall efficiency. Similar analysis also applies to {swim, mcf}.

For {mcf, mcf}, page coloring shows about 30% degraded efficiency relative to default sharing. mcf is a cache-space-sensitive application. Under page coloring, each instance of mcf gets half the cache space. When it runs alone, mcf has a 17% cache miss ratio when given 4 MB L2 cache and that number increases to 35% with a 2 MB L2 cache. Under sharing, two mcfs' data accesses are well interleaved such that each gets better performance than that using a 2 MB cache. Since the two instances are equally aggressive in requesting cache resources with default sharing, the unfairness factor remains low. By tuning the prefetching, hardware throttling can improve efficiency by 6% over the default.

Costs of Dynamic Page Re-Coloring The high efficiency of page coloring is obtained assuming a somewhat ideal page coloring mechanism, meaning that the cache partition point is statically determined and no page recoloring is needed. In reality, a dynamic environment would involve adjusting cache colors and partition points based on changes in the execution environment. Without extra hardware support, re-coloring a page means copying a memory page and it usually takes several micro-seconds on typical commodity platforms (3 microseconds on our test platform). Assuming an application must re-color half of its working set every scheduling quantum (default 100 milliseconds in Linux), our five benchmarks would incur 18–180% slowdown due to page re-coloring (the smallest working set is around 50 MB (equake) and the largest around 500 MB+ (SPECjbb)). This would more than negate any efficiency gain by page coloring.

Instability of Scheduling Quantum Adjustment While scheduling quantum adjustment achieves fairness at coarse granularities comparable to the scheduling quantum size, it may cause fluctuating performance for fine-grained tasks such as individual requests in a server system. As a demonstration, we run SPECjbb and swim on a dual-core chip. Consider a hypothetical resource management scenario where we need to

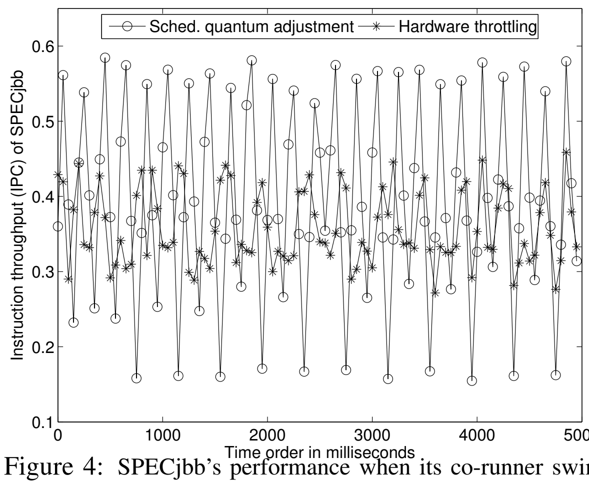

Figure 4: SPECjbb's performance when its co-runner swim is regulated using two different approaches: scheduling quantum adjustment (default 100-millisecond quantum) and hardware throttling. Each point in the plot represents performance measured over a 50-millisecond window.

slow down swim by a factor of two. We compare two approaches—the first adds an equal-priority idle process on swim's core; the second throttles the duty cycle at swim's core to half the full speed. Figure 4 illustrates SPECjbb's performance over time under these two approaches. For scheduling quantum adjustment, SPECjbb's performance fluctuates dramatically because it highly depends on whether its co-runner is the idle process or swim. In comparison, hardware throttling leads to more stable performance behaviors due to its fine-grained execution speed regulation.

4 Conclusion

This paper investigates the use of hardware-assisted execution throttling (duty cycle modulation combined with L1/L2 cache prefetcher configuration) for regulating fairness in modern multi-core processors. We compare against page coloring-based cache partitioning and scheduling time quantum adjustment. Our results demonstrate that simple hardware-assisted techniques to throttle an application's execution speed can achieve high fairness at fine granularity without the drawbacks of page re-coloring costs.

In this work, we have focused on demonstrating the relative benefits of the various resource control mechanisms. Built on a good mechanism, it may still be challenging to identify the best control policy during online execution and exhaustive search of all possible control policies may be very expensive. In such cases, our hardware execution throttling approaches are far more appealing than page coloring due to our substantially cheaper re-configuration costs. Nevertheless, more efficient techniques to identify the best control policy are desirable. In future work, we plan to explore feedback-driven policy control via continuous tracking of low-level performance counters such as cache miss ratio and instructions per cycle executed, in addition to application-level metrics of execution progress.

References

[1] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *15th Int'l Symp. on High-Performance Computer Architecture*, Raleigh, NC, Feb. 2009.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Int'l Symp. on Computer Architecture*, pages 506–517, 2005.

[3] D. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching*. PhD thesis, MIT, 1999.

[4] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *39th Int'l Symp. on Microarchitecture*, pages 455–468, Orlando, FL, Dec. 2006.

[5] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *16th Int'l Conf. on Parallel Architecture and Compilation Techniques*, pages 25–36, Brasov, Romania, Sept. 2007.

[6] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *15th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 13–22, Seattle, WA, 2006.

[7] IA-32 Intel architecture software developer's manual, 2008.

[8] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. on Computer Systems*, 10(4):338–359, Nov. 1992.

[9] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Int'l Symp. on High-Performance Computer Architecture*, Salt Lake, UT, Feb. 2008.

[10] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symp.*, pages 257–274, Boston, MA, 2007.

[11] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and thermal management in the Intel Core Duo processor. *Intel Technology Journal*, 10(2):109–122, 2006.

[12] T. Romer, D. Lee, B. Bershad, and J. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *First USENIX Symp. on Operating Systems Design and Implementation*, pages 255–266, Monterey, CA, Nov. 1994.

[13] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *41st Int'l Symp. on Microarchitecture*, Lake Como, ITALY, Nov. 2008.

[14] G. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *IASTED Int'l Conf. on Parallel and Distributed Computing and Systems*, Anaheim, CA, Aug. 2001.

[15] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, San Diego, CA, June 2007.

[16] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *4th European Conf. on Computer systems*, Nuremberg, Germany, Apr. 2009.

[17] L. Zhao, R. Iyer, R. Illikkal, J. Moses, D. Newell, and S. Makineni. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *16th Int'l Conf. on Parallel Architecture and Compilation Techniques*, pages 339–352, Brasov, Romania, Sept. 2007.