Cooperative Cache Partitioning for Chip Multiprocessors

Jichuan Chang and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison

ABSTRACT

This paper presents **Cooperative Cache Partitioning** (CCP) to allocate cache resources among threads concurrently running on CMPs. Unlike cache partitioning schemes that use a single spatial partition repeatedly throughout a stable program phase, CCP resolves cache contention with multiple time-sharing partitions. Time-sharing cache resources among partitions allows each thrashing thread to speed up dramatically in at least one partition by unfairly shrinking other threads' capacity allocations, while improving fairness by giving different partitions equal chance to execute. Quality-of-Service (QoS) is guaranteed over the long term by orchestrating the shrink and expansion of each thread's capacity across partitions to bound the average slowdown. Time-sharing based cache partitioning is further integrated with CMP cooperative caching [6] to exploit the benefits of LRU-based latency optimizations, which leads to a simplified partitioning algorithm and better performance for workloads that do not benefit from cache partitioning.

We evaluate the effectiveness of CCP by simulating a 4-core CMP running all combinations of 7 representative SPEC2000 benchmarks. For workloads that can benefit from cache partitioning, CCP achieves up to 60%, and on average 12%, better performance than the exhaustive search of optimal static partitions. Overall, CCP provides the best results on almost all evaluation metrics for different cache sizes.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles – Cache memory, C.4 [Performance of Systems] – Design studies

General Terms: Algorithm, Design, Management, Performance

Keywords: Multiple time-sharing partitions, cooperative cache partitioning, CMP, fairness, QoS

1. INTRODUCTION

With an increasing number of processor cores being integrated onto a single chip, chip multiprocessors (CMPs) require efficient organization and management of the on-chip cache resources to provide fast data accesses for concurrently running threads. To efficiently use the aggregate cache capacity, most CMP proposals share cache resources between threads via either a logically shared cache [14, 1, 39, 43] or private caches augmented with capacity sharing policies [35, 8, 6, 2]. However, under high capacity pressure, conventional LRU-based sharing policies cause destructive interference between threads, leading to thrashing [9], unfairness [20] and lack of Quality-of-Service (QoS) [18] (e.g., no guarantee in providing certain baseline performance). Interference isolation for fairness and QoS is important for CMPs as they are used in consolidated servers, shared computing clusters, embedded systems, and other platforms, where meeting these requirements is as important as improving overall throughput. Without hardware solutions, their remedy can complicate the task of operating systems and server administration. In contrast to unconstrained sharing, a private cache design avoids inter-thread interference via design-time partition of the aggregate capacity between processor cores. This design is simple, fair and guarantees QoS, but often incurs many more expensive off-chip misses for thread mixes with non-uniform caching requirements.

To match the perceived requirements of different threads, CMP cache partitioning schemes orchestrate cache allocation with more flexible, usually heterogeneous partitions [38, 20, 42, 18, 21, 17, 27, 29, 11, 10]. Despite their differences in metrics, mechanisms, and policies, prior partitioning schemes have two common characteristics: (1) they use a single spatial partition repeatedly throughout a stable program phase; and (2) compared with LRU-based capacity sharing, they control the partitioning overhead via coarse-grained cache management: allocating large capacity units for long epochs (e.g., 64KB chunks for 5M-cycle epochs [29]). Consequently, prior proposals share two limitations. (1) **Limited functionality.** None of the previous proposals addresses all CMP caching requirements, including thrashing avoidance, fairness improvement, QoS guarantee and priority support, partially due to the difficulty of satisfying multiple, often conflicting, goals in a single cache partition. (2) **Limited scope of application.** Cache partitioning can only outperform LRU-based latency-reducing schemes for some multiprogrammed workloads. An attempt to solely use cache partitioning can cause sub-optimal performance for workloads that do not experience destructive inter-thread interference.

Our proposed solution, **Cooperative Cache Partitioning** (CCP), has two aspects to address the two limitations, respectively. First, CCP introduces Multiple Time-sharing Partitions (MTP) to improve throughput and fairness while maintaining QoS. Specifically, each MTP partition improves at least one thrashing thread's throughput by temporarily shrinking the capacity of other threads. By time-sharing cache resources among multiple unfair partitions that favor different threads, the problems of fairness improvement and priority support are translated into well-studied time-sharing resource management problems. Fairness can thus be improved by giving different threads equal opportunity to speed up, while priority can be supported by allocating different percentages of time slices to different partitions. The MTP partitioning algorithm further guarantees QoS over the long term by using partitions that, on average, can bound each thread's slowdown against the equal partitioning baseline. Comparing with the best single spatial partition based scheme without QoS constraints, MTP achieves up to 60%, and on average 12%, better performance for workloads that can benefit from cache partitioning.

Depending on whether destructive inter-thread interference ex-

ICS'07, June 18-20, 2007, Seattle, WA USA.

ists or not, a workload can either prefer cache partitioning or LRU-based sharing. In order to combine their strengths, the second aspect of CCP is to integrate MTP with CMP Cooperative Caching (CC) [6], an LRU-based caching optimization. The complementary advantages of MTP and CC are combined by dividing the total execution epochs into those controlled by either MTP or CC, according to the fraction of threads that can benefit from each of them. The integrated scheme is shown to achieve robust performance for both workloads with and without destructive inter-thread interference. Furthermore, having CC as the default policy can simplify the MTP partitioning algorithm by focusing only on threads with large speedup potentials, leading to a heuristic-based algorithm that can be practically implemented.

The rest of the paper is organized as follows. Section 2 defines the metrics for CMP cache partitioning, overviews cache partitioning proposals, and classifies benchmarks based on their speedup characteristics. We detail in Sections 3 and 4 the two aspects of our approach: MTP and its integration with CC. Our evaluation methodology and results are presented in Section 5. Related work is discussed in Section 6 and we conclude in Section 7.

2. BACKGROUND AND DEFINITIONS

To simplify later discussion, this section provides background information on CMP cache partitioning and workload characteristics, as well as defines metrics to evaluate different caching policies.

2.1 CMP Multiprogramming Metrics

To compare the effectiveness of CMP caching schemes for multiprogramming, we first need to find proper metrics to summarize the overall performance, fairness and QoS results for a thread schedule. A multiprogrammed workload's throughput can be simply measured as the sum of per-thread throughput (i.e., IPC for our workloads), but quantifying QoS and fairness can be hard, and requires an understanding of these notions in the context of CMP caching.

Our notions of performance, fairness and QoS are based on two principles: (1) proportional-share resource allocation and (2) Pareto efficiency. The first principle states that QoS and fairness is achieved when the shared resource is divided among sharers in proportion to their priorities or weights [41, 4, 40, 34][1]. Using proportional-share allocation to maintain the baseline fairness, the second principle further improve performance (efficiency) by allowing disproportional sharing if it helps some sharers without hurting the others. These principles have been used to define min-max fairness [3], which has wide applications in computer networks and scheduling policies (e.g., Generalized Processor Sharing [26]).

2.1.1 QoS Metric

QoS is the ability to provide a thread with guaranteed baseline performance (corresponding to a specific resource partition) regardless of the load placed on the shared resource from other co-scheduled threads [40]. We use **equal-share cache allocation** to define the performance bottom line for QoS, which corresponds to the special case of proportional-sharing when all threads have the same priority. Notice that equal-priority has been implicitly assumed by previous fair caching proposals [20, 42, 16], while our MTP scheme can also support threads with different priority levels (refer to Section 3.3). This baseline can be implemented either by uniform-sized private caches or an equal partitioning of shared cache capacity between on-chip cores, and it guarantees QoS because all threads get the same capacity and thus can achieve the same performance across different schedules. The equal-share allocation baseline also provides intuitive QoS results to multiprocessor users because it corresponds to traditional multiprocessors with private caches. For similar reasons, Yeh and Reinman [42] use this baseline implemented by private caches. Here we use the even partitioning of a shared cache as our baseline because most existing cache partitioning schemes assume a shared cache.

The QoS metric is thus defined as the sum of per-thread slowdowns (as negative percentages) over this baseline. Same as [42, 25], we claim a caching scheme can guarantee QoS if this measurement is bounded within a user-defined threshold (e.g., -5%). Other ways of measuring QoS exist (e.g., reporting the maximum slowdown or the number of threads that violate QoS), but we use the total slowdown because it captures the behavior of the entire workload and thus is a more stringent criteria.

$$QoS(scheme) = \sum_{i=1}^{\#app} min(0,\ \frac{IPC_i(scheme)}{IPC_i(base)} - 1)$$

2.1.2 Fair Speedup Metric

According to the principle of Pareto efficiency, CMP caching schemes should further improve performance while maintaining fairness, if uneven resource allocation can speed up some threads over the equal-share allocation baseline without hurting others. Now we consider how to measure the scale of performance improvement for co-scheduled threads.

Summarizing the overall performance of multiple benchmarks (co-scheduled threads in our context) has been an extensively discussed topic [32, 19]. We adopt prior wisdom and define the **Fair Speedup (FS)** metric to quantify the overall performance of co-scheduled threads. FS is calculated as the harmonic mean of per-thread speedups over the equal-share allocation baseline.

$$FS(scheme) = \#app / \sum_{i=1}^{\#app} \frac{IPC_i(base)}{IPC_i(scheme)}$$

Using harmonic mean of speedups, FS measures the execution time reduction against a baseline cache configuration that resembles traditional multiprocessors (so higher FS is better). FS is also a fair metric because using the harmonic mean (instead of the sum as used by [42]) rewards uniform speedups and penalizes slowdowns[2], which corresponds to the principle of Pareto efficiency.

The notion of fair speedup is similar to the fair slowdown metrics proposed by Kim et al. [20], which is measured against a single-thread execution baseline where one thread has exclusive use of all cache resources. Such a baseline is borrowed from SMT processors [33], where it corresponds to the single-thread execution mode that allocates all execution and cache resources to one thread. However, single-thread execution in a CMP will waste the majority of execution resources. Instead, we choose to use the equal-share allocation baseline because it has better resource utilization by supporting multiple concurrently running threads and performs similarly as in traditional multiprocessors. For the same reason, two other SMT performance metrics using a single-thread execution baseline—weighted speedup [33] (or WS, which is the sum of speedups) and harmonic mean of speedups [22]—are not used.

2.1.3 Metrics Comparison

The choice of evaluation metrics has a significant impact on CMP caching policies. Below we use two examples to demonstrate the differences between caching schemes that optimize for different metrics.

[1]Contention in other shared resources, especially the memory system, can also cause destructive interference. Here we focus on the impact of destructive interference occurring in the last-level CMP caches, assuming a fair memory system as proposed in [25].

[2]According to the power-mean inequality, the harmonic mean of a vector is maximized when all elements have the same value.

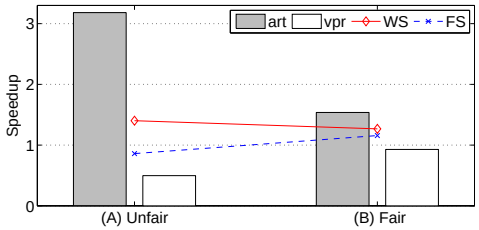

Figure 1: FS vs. WS for Two Example Schemes

Figure 1 shows the per-thread speedups of benchmarks `art` and `vpr` using two partitioning schemes (2 threads sharing a 2MB L2 cache). Scheme (A) maximizes weighted speedup (WS) by tripling the performance of `art`, however, its fair speedup (FS) measurement is worse than the baseline (FS = 1) due to unfair per-thread speedups. On the other hand, the fair scheme (B) optimizes FS at the cost of a lowered WS result because the low-speedup thread is given a more fair cache allocation. This example shows that (1) FS optimization has the side effect of avoiding unfair caching, (2) a scheme that optimizes FS may hurt the WS or IPC results, and vice versa, due to different optimization tradeoffs.

Metrics	Scheme A	<=>	Scheme B
Speedups	0.76/0.76/3.18/3.18		1.97/1.97/1.97/1.97
IPC	*0.52*	==	*0.52*
WS	*2.42*	==	*2.42*
QoS	**-52%**	<	**0%**
FS	**1.28**	<	**2.00**

Table 1: Performance Comparison Using Different Metrics

Table 1 compares the performance of two partitioning schemes for workload `art-art-art-art` (a co-schedule of 4 copies of `art`). Scheme A optimizes WS and throughput without considering its implications on fairness and QoS, while Scheme B aims to simultaneously optimize FS and maintain QoS. If only comparing throughput and WS results, the two schemes have the same performance (shown as *italic* in Table 1). However, our QoS and FS metrics (marked as **bold** in Table 1) reveal that Scheme A cannot guarantee QoS while Scheme B can, and Scheme B achieves better fairness and execution time than A. This example shows that our QoS and FS metrics are able to distinguish whether a scheme can maintain QoS and fairness, but the WS and IPC metrics cannot.

To summarize, using QoS and FS metrics together, we can measure a caching scheme's effectiveness in improving performance, fairness and QoS. We will report results using the FS and QoS metrics in this paper, but also provide WS and IPC results in the evaluation section for comparison.

2.2 Cache Partitioning Background

CMP cache partitioning schemes generally work in repetitive cycles, each consisting of three steps: (1) measurement, (2) partitioning, and (3) enforcement. The first step is to measure and estimate each thread's performance (in terms of miss rate or IPC) for candidate cache partitions. Then this information is used to determine the next cache partition to reach a given optimization goal. The new partition will be enforced in the next execution epoch, while new measurement will be gathered and used in the subsequent cycles. Measurement information can be gathered via profiling [20, 16], LRU stack hit position counting [42], monitoring [37], or dynamic set sampling [29]. This step can incur space or execution time overhead, while inaccurate information can lead to sub-optimal partitioning decisions.

Most prior proposals use coarse-grained **way partitioning** [7, 38] as the basic mechanism to enforce a cache partition (with the exceptions of STATSHARE [27] and cache-level-quota enforcement [31]). Assuming a set-associative cache, way partitioning allocates cache resources in units of cache ways (each way having the same number of cache sets). This mechanism can be implemented with a modified cache replacement policy to ensure that the number of blocks used by a thread at a cache set level does not exceed its way quota.

Cache partitioning proposals differ mainly in their optimization goals and partitioning policies, which are compared in Table 2. To avoid exhaustive search, cache partitioning algorithms use heuristics to prioritize capacity allocation according to the miss rate and speedup characteristics of co-scheduled threads. Unlike prior partitioning algorithms that select the best single spatial partition (SSP) for a given epoch, the proposed MTP scheme selects multiple partitions and allows them to be enforced in a time-sharing manner, possibly across multiple epochs within a stable program phase.

Because cache partitioning schemes are often coarse-grained, they are amenable to not only online simulation, but also offline analysis [21, 20, 16]. To do offline analysis, we first gather performance profiles for all possible (benchmark, capacity) combinations. Comparing against each benchmark's baseline IPC, we can calculate the per-thread speedups for all (benchmark, capacity) combinations and use them to calculate metrics such as FS, WS and QoS. For a given metric, we construct the candidate cache partition space for each workload and exhaustively search in the partition space for the optimal result. Compared with online simulation, offline analysis is idealized because (1) it uses accurate measurement information and (2) it searches for all possible partitions, which can be too slow to be practically implemented.

Due to its idealized nature, offline analysis can be used to estimate the performance upper bounds for given cache partitioning policies. We will use this approach to demonstrate the advantage of our proposed MTP policy over prior cache partitioning schemes, and avoid the need to compare against realistic implementations of prior proposals. We will also compare the offline analysis results of MTP with online simulation results of LRU-based caching schemes to identify the limitation of cache partitioning schemes. However, our final scheme CCP, which integrates MTP with CC, will be evaluated using a practical implementation, online measurement information, and execution-driven simulation results.

2.3 Benchmark Characteristics

Different cache partitioning schemes should be compared using a wide range of multiprogrammed workloads to evaluate their performance robustness. In this paper, we assume a CMP with 4 single-threaded cores and consider all 210 4-thread multiprogramming combinations (repetition allowed) from 7 SPEC benchmarks[3].

Figure 2 shows the performance of 7 SPEC benchmarks under different cache allocations. The IPC data are gathered using a 4-core CMP with 4MB 16-way total L2 cache. With way partitioning, cache resources are allocated in 256KB chunks. The equal-share allocation baseline (marked as the vertical line) is for each thread to use 1MB cache. At least 1 way is allocated to each thread. With three other cores on-chip, this leaves 13 ways ($13 = 16 - 3 * 1$), or 3.25MB, as the maximum capacity for one thread to have. This figure shows that our selected benchmarks have a wide variety of working set sizes and IPC curve shapes, therefore their combinations are able to generate a wide range of workload behaviors.

Figure 3 breaks the IPC curve of `art` into three distinctive regions as more capacity is allocated: (1) pre-working-set region

[3]With repetition, the number of K combinations selected from N objects is C_K^{N+K-1}. Therefore, selecting 4-thread combinations from 7 benchmarks can generate C_4^{7+4-1}=210 workloads.

	Optimization goals	Policies (threads with allocation priority)
Liu et al. [21]	Maximize throughput	Static partitioning
Suh et al. [38]	Minimize miss rate	Greedy (thread with best marginal miss reduction)
Fair Sharing [20]	Minimize slowdown difference	Greedy (thread with most extra misses)
Fast and Fair [42]	Maximize $\sum speedup$ under QoS	Greedy (thread with best speedup)
CQoS [18]	QoS	Generic framework, open policies
OS-managed [31], STATSHARE [27]	Open	Generic model/mechanism, open policies
Utility-based [29]	Maximize WS	Lookahead (thread with best marginal utility)
MTP	Maximize FS under QoS	Iterative (threads with high speedups)

Table 2: Comparing CMP Cache Partitioning Policies

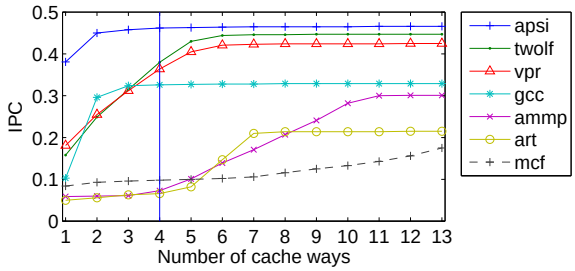

Figure 2: IPC Curves

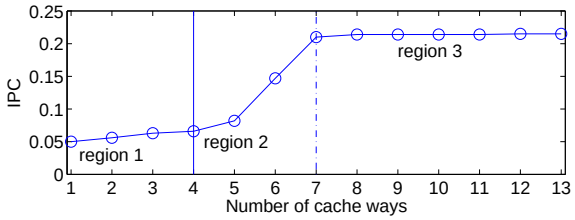

Figure 3: IPC of `art`

(from 1 way to 4 way) represents gradual speedups before the program's working set starts to fit into cache; (2) in-working-set region (from 5 way to 7 way) indicates dramatic throughput increases when the working set can be partly cached; (3) post-working-set region (starting from 7 way) shows saturated performance after the working set is fully cached. Except for benchmarks whose working sets are beyond the capacity of the on-chip cache (i.e., streaming threads), most threads demonstrate IPC curves that consist of regions with distinct slopes, albeit with different cache configurations. According to which region intersects with the equal-share allocation (which is dependent on both thread characteristic and cache configuration), we classify these threads into 3 categories.

- **Supplier threads**: These threads can supply some or all of their equal-share capacity for other threads while still achieving the same level of performance as using all cache resources. They include threads with very small working sets (e.g., `apsi` and `gcc` whose working set sizes are less than 1MB) and streaming threads (e.g., `swim` and `facerec` which are not included in Figure 2).

- **Sensitive threads**: `vpr` and `twolf` are benchmarks whose in-working-set regions are divided by the line of equal-share capacity. The performance of such programs changes significantly over the baseline as cache size varies, therefore judicious cache partitioning is needed when they are co-scheduled with other threads.

- **Thrashing threads**: `art`, `ammp` and `mcf` are benchmarks whose in-working-set regions are beyond their equal-share capacity. These threads usually slow down gradually with reduced capacity, but can speed up dramatically when a certain amount of extra capacity is allocated.

A similar classification can be found in [29], according to the benefit of increased capacity (or utility). Our classification is different by separating thrashing threads from sensitive threads, both called high-utility programs in [29] because they can speed up with more capacity.

3. MULTIPLE TIME-SHARING PARTITIONS

Prior CMP cache partitioning policies use the best single partition to achieve their optimization goals. However, it is an intrinsically hard problem to satisfy multiple goals (e.g., throughput, fairness and QoS) with a single partition when conflicts exist between competing threads. In this section we introduce the notion of Multiple Time-sharing Partitions (MTP) to resolve such conflicts over longer terms. Below we detail the development of MTP as we add support for different caching requirements.

3.1 Thrashing Avoidance

We first discuss when cache partitioning is needed by examining when destructive interference occurs. Starting with the equal-share allocation baseline, if this configuration can satisfy the caching requirements of every co-scheduled thread, then cache partitioning is not needed because little inter-thread interference exists. Cache partitioning is needed only if some threads experience thrashing with their current capacity allocations. These threads will attempt to acquire extra cache resources from each other and from other threads, which leads to performance, fairness and QoS problems.

Thrashing is a classic virtual memory management problem [9], and can be avoided by reducing the multiprogramming level: when the number of competing threads is reduced to a point that their working sets can be cached simultaneously, they can all run much faster. In the context of CMP caching, the number of co-scheduled threads is fixed by the operating system, but cache partitioning can intentionally manage capacity contention by unfairly starving some thrashing threads to make room for other thrashing threads.

Consider partitioning a 4MB 16-way L2 cache between 4 co-scheduled copies of `art`. With the equal-share allocation of a 1MB L2 cache, `art` has a low IPC of 0.066 due to thrashing (over 50 off-chip misses per thousand instructions) as previously shown in Figure 3. As more cache resources are allocated, its throughput increases quickly and reaches a saturating point of 0.215 IPC with 1.75MB capacity. At this point, thrashing can be avoided for 2 threads by unfairly allocating to each of them 1.75MB capacity, and starving the other threads each with a 256KB cache (0.05 IPC). This partition doubles the total throughput ($0.215*2+0.05*2$=0.52, which is two times of $0.066 * 4$=0.264), but is unfair to the threads being starved.

3.2 Fairness Improvement

Cache partitioning between 4 copies of `art` is an example of the throughput-fairness dilemma. When the available cache capacity cannot simultaneously satisfy the working set requirements of multiple large threads, a compromise has to be made within a single

spatial partition. In this example, fair partitions cause thrashing for all threads, while thrashing avoidance requires unfair partitioning. Existing cache partitioning schemes all face this dilemma, but differ in the way they trade off between throughput and fairness.

We resolve this dilemma by learning from a similar example in game theory [12]. Consider two officemates who commute to their workplace: efficiency is doubled when they carpool, but it is unfair because the driver invests more effort and money. Not carpooling is a fair strategy, but is also inefficient. In real life, such games are played daily by the same players who often improve both performance and fairness by "taking turns" to drive when they carpool. We adopt the same cooperative policy to simultaneously improve throughput and fairness with multiple time-sharing partitions (MTP). Instead of using a single partition that is either low-throughput or unfair, multiple unfair but high-throughput partitions are selected and given equal opportunity to be used.

Specifically, individual threads are coordinated to shrink and expand their cache allocations in different cache partitions. Within a partition, the spare capacity collected from shrinking threads is used by expanding threads, and different threads are expanded in different partitions. As a thrashing thread goes through shrinking and expanding partitions, its average throughput can be much better than its baseline throughput. This is because a thrashing thread's baseline performance is already low by definition, and shrinking capacity usually causes insignificant slowdown. However, the thread can achieve dramatic speedup in expanding partitions (when the allocated cache can hold its working set) and, on average, the speedup in one expanding partition is often more than what is needed to compensate the slowdowns in multiple shrinking partitions. Overall, the workload's fair speedup (FS) is improved because all expandable threads get a fair chance to speedup.

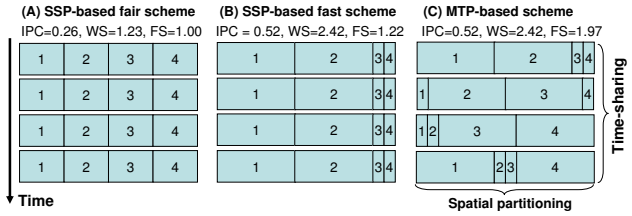

Figure 4: Cache Partitioning Options for `art-art-art-art`

Figure 4 compares three cache partitioning schemes for 4 copies of `art`. Single spatial partition (SSP) based schemes A and B provide the most fair and fast partitions, respectively. Based on MTP, scheme C can both maintain the same level of fairness as scheme A (by equalizing per-thread speedups) and achieve the same high throughput (IPC=0.52) and weighted speedup (WS=2.42) of scheme B. Such improvement is reflected by its high FS result (97% and 61% higher than scheme A and B, respectively), but can be overlooked if we only compare the IPC or WS results.

3.3 Priority Support

MTP extends the option of cache partitioning from the single dimension of space-sharing into two-dimensional time-sharing between spatial partitions. The time-sharing optimization can be applied to any proportional-sharing resource partition baseline, thus supporting priority if the priority levels of co-scheduled threads are reflected in the baseline.

Priority can also be supported through time-sharing. Instead of giving different threads equal opportunity to speedup, different time-sharing priorities can be assigned to different unfair partitions to deliver differentiated levels of performance. Because time-sharing based priority support has been well understood and implemented by operating systems [15, 41], MTP can serve as the cache management primitive to the high-level software by focusing on the determination and enforcement of multiple unfair partitions.

As priority specification and interpretation are usually conducted by end-users and operating systems, we leave the development and evaluation of priority algorithms for future work and assume the co-scheduled threads have the same priority for the rest of the paper.

3.4 QoS Guarantee

QoS can be guaranteed either in a real-time manner, or over the long term to meet different thread's timing requirements. Real-time QoS is needed only by certain programs (e.g., real-time video playback), and is not needed for many other programs. For example, users of SPEC-like programs are most concerned about total execution time, and thus long-term QoS, often measured over many millions of cycles.

To guarantee real-time QoS, fast and fair partitioning [42] reserves for each thread a **guaranteed partition**, which is the minimum amount of cache space required to achieve the same level of performance as using the equal-share cache allocation. Further speedup can be obtained by intelligently partitioning the remaining space. However, because only supplier threads (defined in Section 2.3) can have their guaranteed partitions smaller than the equal-share capacity, the cache partitioning algorithm is often left with a limited amount of space to optimize, resulting in lowered performance compared with schemes under no QoS constraints.

Single spatial partition (SSP) based cache partitioning schemes experience the same problem even for threads that require only long-term QoS. Because the same spatial partition is used repeatedly throughout a stable program phase, these schemes have to guarantee long-term QoS by guaranteeing QoS within every cache partition. In contrast, MTP's cooperative shrink/expand model can be used to guarantee long-term QoS with little loss of performance. To meet the QoS requirement, the MTP partitioning algorithm now uses multiple partitions to maximize FS, under the constraint that each thread's average throughput across multiple partitions is no worse than the equal-share baseline throughput.

To demonstrate MTP's advantage in guaranteeing long-term QoS, Figure 5 compares SSP and MTP based cache partitioning schemes that optimize FS under different QoS requirements: without QoS (SSP_{noQoS} and MTP_{noQoS}), real-time QoS (SSP_{QoS} and MTP_{rtQoS}), and long-term QoS (MTP_{ltQoS}). These results are obtained from an offline analysis to show the performance potential of an ideal MTP_{ltQoS} implementation. In Sections 4 and 5, we will develop and evaluate its practical implementation.

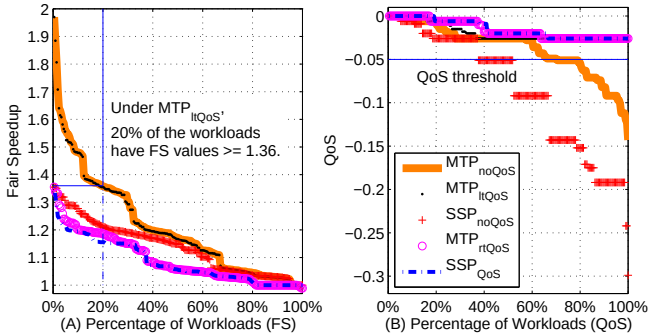

Figure 5: Comparing SSP and MTP Schemes

For each scheme, we plot the percentage of workloads that can achieve various metric values. These curves are essentially Cumulative Distribution Functions (CDF) being transposed, so that a higher curve indicates a better performing scheme. For example in Figure 5(A), each point (X%, Y) on the MTP_{ltQoS} curve indicates

that, X% of workloads have FS measurements equal to or large than ($\geq$) value Y. Notice that for the same type of QoS guarantee, the ideal SSP scheme can never outperform the ideal MTP because single spatial partitioning is a special case in the MTP model, which is also empirically shown in the figure.

Figure 5(A) only shows 4 distinct curves for 5 schemes because MTP_{ltQoS} and MTP_{noQoS} have almost the same FS results. Similarly, the curves of the QoS guaranteeing schemes overlap in Figure 5(B). Figure 5 further shows that (1) SSP_{noQoS} and MTP_{noQoS} can not bound per-thread slowdown within the user-specified threshold (-5% in this paper); (2) SSP_{QoS} and MTP_{rtQoS} are the worst performing policies (their curves overlap for workloads with smaller FS values), indicating that real-time QoS guarantee can restrict performance optimization; and (3) MTP_{ltQoS} can maintain long-term QoS while achieving almost the same performance as the best performing scheme MTP_{noQoS}.

For its performance and QoS benefits, we now use MTP_{ltQoS} as the representative MTP policy in the rest of the paper, and denote it as MTP. MTP can support real-time QoS by simply reserving guaranteed partitions for real-time threads and optimizing the other threads with the remaining capacity.

3.5 Summary

MTP is a high-level cache partitioning policy that extends existing proposals with time-sharing multiple cache partitions. MTP addresses four cache partitioning requirements: (1) thrashing is avoided by unfair allocation within a partition; (2) fairness is improved with fair time-sharing between unfair partitions that each favors a different subset of co-scheduled threads; (3) priority can be supported with either prioritized proportional-share baselines or unfair time-sharing; and (4) different types of QoS can be guaranteed by bounding per-thread slowdown within each partition or across multiple partitions.

With multiple epochs in one iteration, MTP takes longer to adapt to phase/scheduling changes. MTP (and most partitioning schemes) cannot promptly adapt to frequent, irregular changes because their prediction of future execution relies on stable phases. In our infrastructure (Solaris-based full-system simulator), we observe stable thread-scheduling and repetitive phase changes. Our scheme works for such changes by using longer epochs (20M-cycles) to include multiple phases, whose aggregate behavior is stable enough for cache partitioning.

MTP can be implemented in different ways. A hardware-only solution is transparent but less flexible, especially considering priority support. Cooperation between hardware and software allows hardware to collect measurement and enforce partitioning decisions, and software to schedule partitions to meet high-level requirements. This section used offline analysis results to demonstrate the advantages of MTP without considering implementation details. Next we will present a practical, heuristic-based MTP partitioning algorithm by exploiting the benefits of CMP Cooperative Caching (CC) [6].

4. INTEGRATING MTP WITH CC

This section addresses another limitation of existing cache partitioning proposals—inadequacy for workloads that are well supported by LRU-based latency optimizations, where cache partitioning can hurt. We propose a new hybrid scheme, Cooperative Cache Partitioning (CCP), that combines the advantages of MTP and CMP Cooperative Caching (CC) [6]. Below, we motivate the need for the integration by showing the complementary advantages of MTP and CC on different workloads. We then develop a simple online heuristic to select MTP partitions based on the different characteristics of MTP and CC, and extend the CC design to implement the hybrid scheme.

4.1 Motivation

Figure 6 compares the FS and QoS results of MTP with two LRU-based caching schemes: CC and shared cache, using the same aggregate cache size and associativity. We use scatter plots to reveal the correlation between the best performing schemes and workload characteristics. To show the advantages of MTP and CC over shared cache, we normalize the FS values against the better results provided by MTP and CC (i.e., Max[FS(MTP), FS(CC)]). The 210 workloads are also clustered into two groups according to whether MTP outperforms CC (i.e., FS(MTP)>FS(CC)).

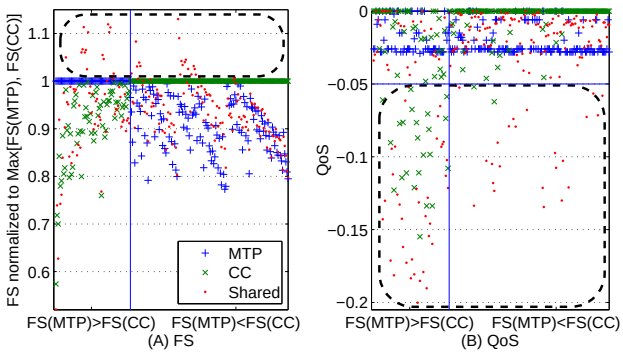

Figure 6: Comparing MTP with CC and Shared Cache. Each point on the X-axis represents a workload, and the corresponding Y-values are the measured results of the three schemes.

Focusing on the regions highlighted by the dotted rectangles, several observations can be made from Figure 6. First, Figure 6(A) shows that only a small number of dots (10%) are above 1 and fewer (3%) are above 1.1, indicating that a shared cache only outperforms both MTP and CC infrequently and insignificantly. Second, MTP only provides better performance than CC for 32% of the workloads, indicating the limited effectiveness of cache partitioning over CC for many workloads. Third, for workloads that benefit less from MTP, CC is almost always the best performing scheme (Figure 6(A)) and can guarantee QoS (Figure 6(B)), showing the complementary strengths of MTP and CC.

These observations imply that we can choose the best performing scheme (in both FS and QoS) for a given workload according to whether FS(MTP)>FS(CC): if MTP provides better FS result than CC, then MTP is very likely to perform the best; otherwise, CC is the best choice. Therefore, a hybrid scheme that integrates MTP and CC can potentially provide the best performance for all workloads by simply choosing the better scheme for any given workload. Below we analyze the reasons for CC and MTP's performance advantages, in order to achieve such an integration.

4.1.1 Advantages of CC

Two major reasons contribute to CC's performance advantage over MTP: (1) latency optimization over shared cache and (2) LRU-based fine-grained cache sharing. The first reason is unique to CC—it is the only private cache based CMP caching proposal that approximates global LRU replacement for multiprogrammed workloads; the second is supported by both CC and a shared cache.

CC reduces the average cache access latency by keeping a thread's data set locally in the processor's private L2 cache. Due to on-chip wire delay, local cache access latencies are much lower than remote access latencies. Comparing with a shared cache where data are distributed evenly across all banks, and a large fraction of L2 accesses are to remote banks, CC has the advantage of servicing

most L2 accesses locally. For threads whose working sets can be mostly satisfied by a private cache, such reduced L2 cache latencies often translate into higher performance.

Similar to a shared cache, CC supports LRU-based capacity sharing by (1) allowing a local cache's victim block to be placed in a randomly picked peer cache (called spill in [6]), and (2) approximating global LRU replacement for multiprogrammed workloads via the combination of local LRU and global spill/reuse history. CC differs from a shared cache in that it allocates capacity according to the local thread's L2 reference stream and the remote threads' L2 miss streams, which are first filtered by their local L2 caches.

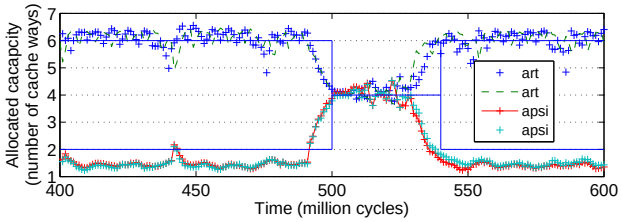

Figure 7: Cache Allocation for `art-art-apsi-apsi`

An LRU-based policy can provide near-optimal cache allocation for many workloads [36]. Figure 7 shows the amount of cache allocated by CC for individual threads in `art-art-apsi-apsi`. CC shows spatial fine-grained sharing by allocating on average 6.5 ways and 1.5 ways of capacity to `art` and `apsi` to better fit their capacity requirements, while way partitioning schemes can only allocate capacity in units of cache ways (the best partition allocates 6 ways to each `art` and 2 ways to each `apsi`). Temporal fine-grained sharing occurs during simulation time 500 to 530 million cycles, when `apsi` enters a phase that needs more capacity. CC adapts to this change swiftly without explicit phase tracking support or cache repartitioning. Due to spatial fine-grained sharing, `art` achieves 34% better speedup than way partitioning. The throughput of `apsi` is also slightly improved due to temporal fine-grained sharing (even though its average cache capacity is lower than way partitioning). This example also shows that CC is capable of managing workloads whose aggregate working set size modestly exceeds the total cache capacity, by dynamically balancing each individual thread's cache allocation.

Programs with highly non-uniform associativity demands across different cache sets [30] can also benefit from fine-grained sharing. For example, although `ammp` has a working set size of 1.5MB (or 6 cache ways), it can further speed up by 2X when the associativity requirements of certain hot sets are satisfied by a 16-way allocation.

4.1.2 Heuristics for MTP to Outperform CC

We now try to discover the characteristics of the workloads that can achieve better performance with MTP than with CC. Such characteristics will then be used to develop a simple heuristic to integrate MTP with CC.

To simplify discussion, we first assume that within one group of MTP partitions, a thread always uses the same capacity C_{expand} in all of its expanding partitions and the same capacity C_{shrink} in all shrinking partitions, thus achieving the same speedup Sp and slowdown Sd repeatedly. Offline analysis results show that this assumption has almost no performance impact on MTP. We further filter out supplier threads by allocating their guaranteed partitions to them, and allocate the remaining space among other threads.

To guarantee QoS and improve performance using MTP, each thread's total speedup has to exceed its total slowdown, and at least one thread should have a much larger total speedup. This can be achieved in two ways. The first way is to have a thrashing thread whose dramatic Sp can compensate the total slowdown in multiple shrinking partitions. The other way for a thread to achieve speedup using MTP, without a large Sp, is to have a modest Sp, but a steep speedup curve and a gradual slowdown curve, so that the speedups accumulated in multiple expanding partitions exceed the slowdown in one shrinking partition. However, achieving a modest Sp along with a steep speedup curve requires only a small amount of extra capacity, in which case CC is likely to achieve the same effect because the LRU policy is better at fine-turning cache allocation to achieve speedups (example shown in Figure 7).

The above analysis suggests that the common case for MTP to outperform CC is to have at least one thrashing thread, determined by whether its speedup Sp in one expanding partition is larger than the total slowdown accumulated in shrinking partitions. Here the Sp and Sd values are dependent on both the thread's IPC curve and the available capacity (which further depends on capacity allocated to co-scheduled threads). The test of a thrashing thread will be used as the partitioning heuristic for MTP.

The common case also explains why CC can guarantee QoS when its FS value is better than MTP (Figure 6(B)). A QoS violation occurs in CC only when a thread's private cache is overly used by blocks replaced from other threads (or spilled blocks [6]). Because CC's private cache has the same capacity as the equal-share baseline used to define thrashing, the aggressive spilling implies the existence of high miss rates and thus thrashing threads. Therefore, for workloads that prefer CC, the spilling should not be too invasive to affect QoS, otherwise thrashing will occur and cause MTP to be preferred.

4.2 Cooperative Cache Partitioning

We now develop Cooperative Cache Partitioning (CCP), a heuristic-based hybrid cache allocation scheme that integrates MTP with CC. CCP consists of three components: (1) a heuristic-based partitioning algorithm to determine the MTP partitions, (2) a weight-based integration policy to decide when to use MTP and CC, and (3) modifications to the baseline CC design to enforce fine-grained cache partitioning decisions.

4.2.1 CCP Partitioning and Weighting Heuristics

Before MTP partitioning, CCP first gathers each thread's L2 cache miss rates under candidate cache allocations, and uses them to estimate the IPC curve. Miss rates are collected in our simulator in dedicated, online sampling epochs where each thread takes turns to use the maximum amount of cache. We use LRU stack hit counters to estimate miss rates under all possible cache associativities to reduce sampling overhead. Although such overhead can be avoided with the recently proposed UMON online sampling mechanism [29], we include it in our evaluation results.

Using IPC estimations, each thread's guaranteed partition (for real-time QoS guarantee) can be calculated. CCP also initializes each thread's C_{expand} to the minimum capacity needed to achieve the highest speedup, and C_{shrink} to the minimum capacity that can ensure long-term QoS when cooperating with C_{expand}. A thread is a supplier thread if its C_{shrink} and guaranteed partition are the same.

The CCP partitioning algorithm (shown in Table 3) then returns a set of MTP partitions that are likely to outperform CC, using the test of a thrashing thread as a simple heuristic. This algorithm has the following three steps: (1) filtering out supplier threads which cannot benefit from cache partitioning; (2) determine MTP partitions that each favors one thrashing thread by starving other thrashing threads with their C_{shrink} capacity; (3) for MTP partitions where one expanding thread can not use all the remaining space, expand other threads to further increase speedup. We will describe

Inputs: capacity C, thread set TS, sample results (IPC[i][c], guaranteed partitions g[i]);
Outputs: expanded[i], MTP partitions MTP[p][i]; /* *Thread i's capacity in partition p* */

/* **Step 1: Filter out supplier threads** */
Identify supplier threads SupplierTS, subtract their g[i] from C;

/* **Step 2: Determine the set of thrashing threads ThrashTS** */
/* init stable = false; ThrashTS=TS-SupplierTS; */
while (ThrashTS is non-empty and !stable)
    stable=true;
    foreach thread i∈ThrashTS
        C_{expand}[i]=i's capacity usage when other threads use their C_{shrink}[j];
        stable &= thrashing_test(i, size(ThrashTS), C_{expand}[i], C_{shrink}[i]);

/* **Step 3: Merge multiple expanding threads** */
/* init p = 0; expanded[i] = false; MTP[p][j]=C_{shrink}[j]; */
foreach thread i∈ThrashTS, p++
    foreach thread j, start from i, in circular order
        MTP[p][j] += minimal remaining capacity for j to achieve its best speedup;
        if (MTP[p][j]≥C_{expand}[j]) expanded[j]=true; /* *Expanded in MTP* */

thrashing_test(i, nump, expand, shrink) /* **Key heuristic** */
    if (IPC[i][expand]-IPC[i][base])≥(nump-1)*(IPC[i][base]-IPC[i][shrink]) **return** true; /* large speedup */
    C_{shrink}[i]=g[i]; C=C-g[i]; remove i from ThrashTS; **return** false;

Table 3: CCP Partitioning Algorithm

steps (2) and (3) in detail because step (1) is rather straightforward.

Step (2) determines the set of thrashing threads by removing threads whose speedups are not large enough to guarantee long-term QoS. Each candidate thread is tested by the function **thrashing_test**, to see whether its speedup in one expanding partition can compensate for the total slowdown accumulated in other (shrinking) partitions. The threads that fail the **thrashing_test** are assigned with their guaranteed partitions and removed from the candidate set, which will reduce the number of candidate partitions, the amount of remaining capacity and possibly remaining candidates' C_{expand} and speedups. Such tests are repeated until one of the two termination conditions is satisfied: (1) the candidate set is empty, or (2) all candidate threads pass the test. This step is guaranteed to terminate because each round of tests either reduces the candidate set size which leads to condition (1) in a finite number of steps, or satisfies condition (2).

After step (2), it is possible that in an MTP partition, the expanding thread does not need all the spare space provided by other shrinking threads. Step (3) merges multiple expandable threads in such a partition to further increase speedup. To be fair, the algorithm expands different sets of threads in different partitions.

This algorithm returns a set of MTP partitions and a vector $expanded$. A thread i benefits from MTP if it is allocated with C_{expand} capacity in at least one partition ($expanded$[i] is true), otherwise it is likely to benefit from CC. This observation leads to the CCP integration heuristic: the execution time is broken into epochs managed by either MTP or CC, weighted by how many threads can benefit from them respectively. For N co-scheduled threads, if M of them can be expanded by MTP partitions, then CCP will use MTP for every M out of N epochs and use CC for other epochs. A special case is when no thread is expanded because step (2) cannot find any MTP partitions, in which case CC should be used throughout the execution.

4.2.2 Extending CC to Enforce Capacity Quota

To enforce MTP partitions, CCP modifies the baseline CC design to monitor each thread's capacity usage and maintain capacity quota by throttling spill-based capacity sharing. For our evaluated 4-core CMP (assuming single-threaded cores), every cache block is augmented with 2 bits to indicate its owner thread ID. Each cache maintains 4 counters to reflect the numbers of blocks used by different threads. By periodically exchanging capacity usage information between caches, CCP can monitor the capacity usage of different threads. With such information, CCP maintains quota by disallowing over-quota threads to spill and disabling spills into under-quota threads.

5. EVALUATION AND RESULTS

We evaluate the effectiveness of different cache allocation schemes using Virtutech Simics-based [23] full-system simulation. The cache and memory simulator is derived from Ruby, which is part of the GEMS toolset [24]. A single-issue, in-order processor model is used, which allows us to simulate all 210 multiprogrammed workloads. We choose this methodology because, under the same simulation time, simulating a wide range of workload combinations allows us to recognize the limitations of different approaches on different workloads, which could have been missed by simulating a few combinations with a more detailed processor model.

Table 4 lists the relevant configuration parameters used in our simulations. The same total capacity and associativity are used for shared cache (both with and without way partitioning) and CC. We use the reference input sets for the selected SPEC benchmarks, except for `art` which uses the train input.[4] All benchmarks are fast forwarded by 800M instructions to bypass program initialization, and simulated for 700M cycles.

We compare the online simulation results of realistic CCP implementation with offline analysis results of ideal cache partitioning policies (e.g., MTP). Because the ideal MTP implementation results were shown to be the performance upper bound of existing cache partitioning schemes in Section 3, we do not compare CCP with realistic implementations of prior partitioning proposals.

We first compare CCP with its two baseline schemes CC and MTP in terms of FS, followed by comparison between CCP with

[4] `Art` with reference input is a streaming (thus supplier) thread. We do not include streaming threads because cache partitioning for them is very simple: their IPCs don't change with L2 cache allocations, so we can simply allocate the minimal capacity (e.g., 1 cache way) to them.

Component	Parameters
Processors	4-core, single-issue, in-order
Block size	128-byte
L1 I/D caches	32KB, 2-way, 2-cycle
L2 caches	sequential tag/data access, 15 cycles total
On-chip interconnect	Point-to-point, 5-cycle per-hop latency
Main Memory	300 cycles total
Coherence	MOSI-based directory protocols
CC	1MB per-core private cache, 4-way
shared cache	1MB per-bank capacity, 4 banks, 16-way

Table 4: Processor and Memory System Parameters

idealized offline analysis results on other metrics—QoS, throughput and weighted speedup. Lastly, we evaluate the performance robustness of CCP by halving the total cache size.

5.1 Effectiveness of CCP

In Section 4, MTP was shown to be better than CC for only a subset of workloads. Since the ideal MTP implementation represents the best cache partitioning results, we now refer to workloads that prefer CC over MTP as workloads where cache partitioning could hurt performance, and the other workloads as workloads that need the help of cache partitioning. Figure 8 compares the performance of CCP (realistic) with MTP (ideal) and CC (realistic) on both classes of workloads. Only FS results are reported because both CCP and MTP can guarantee QoS.

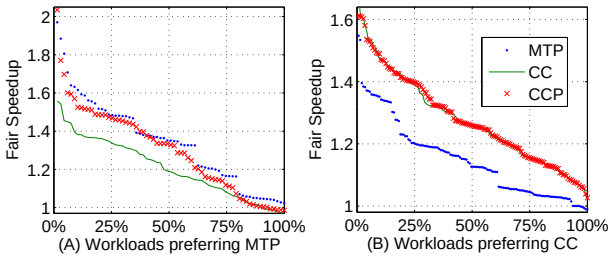

Figure 8: Comparing MTP, CC and CCP's FS Results

Same as in Figure 5, we use transposed CDF curves to show the percentage of workloads that can achieve various levels of performance. Here, a higher curve indicates a better scheme because it achieves better FS measurements across different fractions of the workloads, and the gaps between curves correspond to their performance differences.

Figure 8(A) shows that when cache partitioning is needed, CCP achieves comparable performance as MTP (the gap between CCP and MTP curves is small), and much better FS values than CC (the gap between CCP and CC is much larger). The performance difference between CCP and MTP reflects the difference between our practical partitioning heuristic and a less realistic, offline, exhaustive search of MTP partitions. For workloads where cache partitioning hurts, Figure 8(B) shows that CCP performs slightly better or the same as CC and significantly better than MTP. Together they demonstrate that CCP effectively combines the strengths of both MTP and CC.

5.2 Results of Different Metrics

Besides FS, CMP caching performance can also be evaluated using other metrics. We use transposed CDF plots to compare CCP (realistic) and MTP (ideal) against two single spatial partition (SSP) based schemes IPC_{opt} and WS_{opt}, which optimize offline for throughput and weighted speedup, respectively. Focusing on workloads that need cache partitioning, Figure 9 compares IPC_{opt}, WS_{opt}, MTP and CCP over 4 different metrics: (A) fair speedup, (B) QoS, (C) throughput, and (D) weighted speedup.

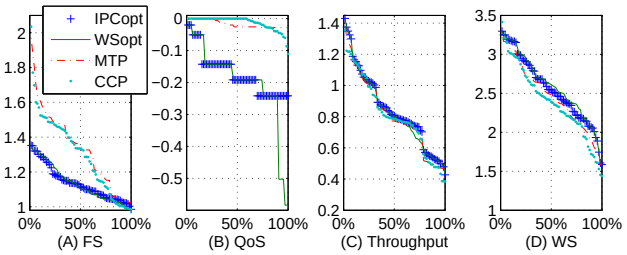

Figure 9: Results of Multiple Metrics for Workloads that Need Cache Partitioning (32% of All Workloads) for 4MB Cache

For the first two metrics (FS and QoS), MTP and CCP are both significantly better than IPC_{opt} and WS_{opt}. This is because the SSP-based schemes, when their goals conflict with fairness and QoS requirements, often optimize by favoring only a subset of threads while sacrificing the performance of other threads. For IPC and WS metrics, both IPC_{opt} and WS_{opt} are better, although the gap between different schemes are much smaller than in Figure 9(A) and Figure 9(B). As illustrated in Section 2.1.3's examples, this is because schemes optimizing for WS, IPC and FS have different tradeoffs between performance and fairness.

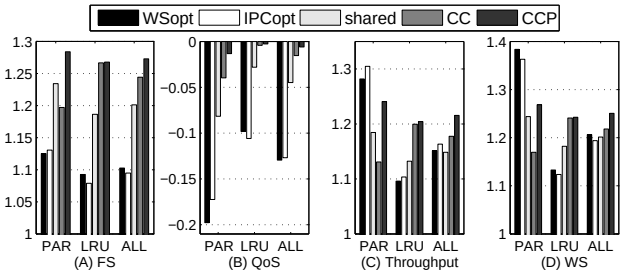

Figure 10: Average Improvement for 4MB L2 Cache

Figure 10 summarizes the average improvement of WS_{opt}, IPC_{opt}, shared cache, CC and CCP over the equal-share baseline for different metrics. The average improvements are calculated as geometric means of per-workload improvements[5]. The results are summarized over three groups of workloads: "PAR" represents workloads that prefer cache partitioning, "LRU" covers other workloads, while "ALL" includes all workload combinations. This figure shows that for workloads preferring cache partitioning (PAR), CCP performs much better than a shared cache and CC, while achieving similar or much better results than the two cache partitioning schemes. Considering workloads that prefer LRU-based sharing (LRU) and all workloads (ALL), CCP provides the best average results on all reported metrics.

5.3 Results for a 2MB L2 Cache

Now we evaluate the robustness of CCP when the total L2 cache capacity is reduced to 2MB. The reduction of cache size not only increases capacity contention between threads, but also causes some benchmarks to switch their categories (e.g., from supplier threads to sensitive threads, or from sensitive threads to thrashing threads) so the performance of CCP can be tested under new scenarios.

Figure 11 uses transposed CDF plots to compare 4 cache partitioning schemes (IPC_{opt}, WS_{opt}, MTP and CCP) on workloads that need cache partitioning. CCP again achieves comparable FS and QoS results as the ideal MTP implementation and outperforms the two SSP-based partitioning schemes. This shows the robustness

[5]QoS results are summarized using the arithmetic mean because the QoS measurements of many workloads are zero, which causes the average results to be the same (zero) with geometric means.

of CCP's heuristic-based partitioning algorithm. In terms of fairness and throughput tradeoff, the weighted speedup results of MTP and CCP are similar to IPC_{opt} and WS_{opt}, while their throughput results are 10% lower. Again, QoS constraint and fair speedup optimization are the two reasons that cause MTP and CCP's lower throughput, while IPC_{opt} and WS_{opt} can achieve better throughput without satisfying such constraints.

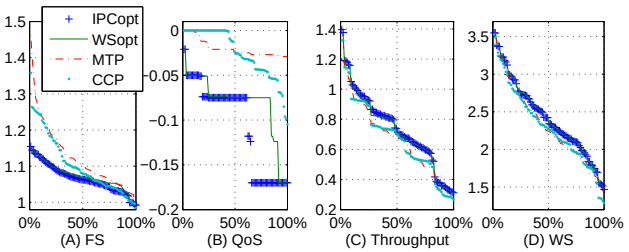

Figure 11: WS_{opt}, MTP and CCP on Workloads that Need Cache Partitioning (40% of All Workloads) with 2MB Cache

Figure 12 compares the average improvements of various schemes over the equal-share baseline. Due to higher cache contention, more workloads now prefer cache partitioning (increased from 32% to 40%). The performance of a shared cache also drops significantly and comes close to the equal-share baseline performance. In contrast, CCP still consistently outperforms other schemes for workloads where cache partitioning can hurt. However, the gap between cache partitioning schemes and CCP (as well as CC) is reduced because, due to capacity pressure, latency optimization contributes less to the overall speedup. Averaged over all 210 workloads, CCP achieves the best results on almost all metrics (except for throughput, where CCP is 1% lower than IPC_{opt}).

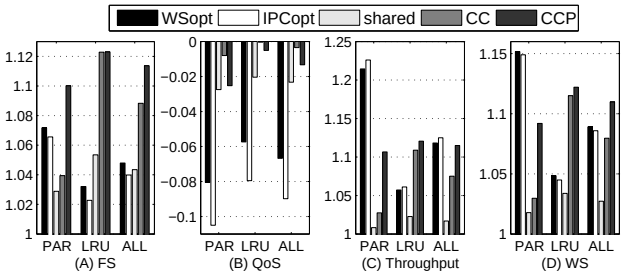

Figure 12: Average Improvement for 2MB L2 Cache

6. RELATED WORK

Cooperative cache partitioning is closely related to CMP cache partitioning research. Stone et al. [36] studied the problem of partitioning cache capacity between different reference streams, and identified LRU as the near-optimal policy for their workloads. Suh et al. [38, 37] first applied way partitioning to shared CMP caches, using in-cache monitoring mechanism and marginal gain based partitioning algorithm to reduce off-chip misses. Recently Qureshi and Patt [29] proposed UMON sampling mechanism to provide more precise measurement and lookahead partitioning to handle workloads with non-convex miss rate curves. The UMON mechanism can be combined with CCP to improve miss rate measurement and adapt to fine-grained program phase changes. Dybdahl et al. [11] extended way partitioning by overbooking cache capacity to account for non-uniform per-set requirements, and evaluated its effectiveness using private L1/L2 caches with a shared L3 cache. Dybdahl and Stenstrom [10] also proposed an adaptive shared/private partitioning scheme to both avoid inter-thread interference and exploit the locality benefits of private caches. Their partitioning algorithm is essentially the same as in [38], but instead of in-cache monitors, "shadow tags" are used to measure the benefit of having one extra cache way.

Kim et al. [20] emphasized the importance of fair CMP caching, and proposed a set of fairness metrics as their goal of optimization. Iyer [18] motivated the importance of QoS guarantee and prioritization with a general QoS framework. Yeh and Reinman [42] focused on throughput improvement with QoS guarantee and exploited the latency advantage of a NUCA design. Their notion of guaranteed partition is adopted in this paper. Rafique et al. [31] and Petoumenos et al. [27] proposed spatially fine-grained partitioning support, which is supported in CCP by throttling cooperative sharing activities. Hsu et al. [16] studied various partitioning metrics and policies, and recognized the difficulties of improving both throughput and fairness. CCP is the only partitioning proposal that simultaneously optimizes throughput, fairness, and QoS for a wide range of workload combinations.

Beside capacity optimization, CMP caching proposals also reduce on-chip access latency. CCP combines the strength of cache partitioning with such fine-grained latency reduction optimizations via the integration of MTP and CC [6]. MTP can also be integrated with other private cache based latency optimizations [35, 2] that do not support global LRU replacement. Planas et al. [28] provided a model to explain cache partitioning speedups over LRU-based fine-grained sharing.

CCP also borrows heavily from virtual memory management research. Verghese et al. [40] recognized the need for performance isolation for SMP-based systems, and proposed mechanisms to provide isolation under heavy load while allowing sharing under light load. CCP aims to support a broader sense of performance isolation that also balances between partitioning and sharing. The integration between MTP and CC resembles WSClock [5], a paging algorithm that integrates working-set based partitioning with global LRU replacement. CCP also uses other well-known operating system techniques such as thrashing avoidance and time-sharing based resource management, but differs from software-only schemes [13] that manage cache sharing solely in the operating system.

7. CONCLUSION

Current cache partitioning schemes have limited functionality and applicability because they can only support a subset of CMP caching requirements, and they can not compete with LRU-based latency-reducing caching schemes (e.g., CC) for many workloads. To answer these challenges, this paper introduces Multiple Time-sharing Partitions (MTP) to simultaneously improve throughput and fairness while guaranteeing QoS. MTP is further integrated with CMP Cooperative Caching (CC) to exploit its latency optimizations. The resulting Cooperative Cache Partitioning (CCP) scheme is evaluated and shown to provide the best overall performance over 210 combinations of 7 representative SPEC2000 benchmarks under different cache sizes. For a 4-core CMP with 4MB L2 cache, CCP on average improves performance (measured by fair speedup) by 12% and throughput by 4.5%, respectively, comparing against the best static partitioning schemes optimizing fair speedup and throughput, respectively, while maintaining long-term QoS.

CCP takes a first step in balancing partitioning-based capacity optimizations and LRU-based latency optimizations for multiprogrammed workloads. Future research is needed to extend CCP to better adapt to phase/scheduling changes, as well as to support large-scale CMPs and CMPs with SMT cores. For environments that prefer higher throughput over execution time reduction, fairness, and QoS guarantee, CCP can also be modified to use a single unfair partition that only optimizes throughput, which is also left as future work.

8. ACKNOWLEDGEMENTS

The authors thank Philip Wells, Koushik Chakraborty, Saisantosh Balakrishnan, Matthew Allen and Vikas Garg for helpful discussion and proofreading. We also thank Jim Smith and Kyle Nesbit for their valuable comments on the selection of baselines and metrics, and the anonymous reviewers for their comments. Special thanks to Philip Wells who first suggested time-sharing based cache partitioning.

This research is supported in part by NSF grants CCR-0311572 and CNS-0551401, funds from the John P. Morgridge Chair in Computer Science, and the University of Wisconsin Graduate School (a WARF Named Professorship). Sohi has a significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the National Science Foundation.

9. REFERENCES

[1] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, 2000.

[2] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO-39)*, 2006.

[3] D. Bertsekas and R. Gallager. *Data Networks (2nd ed.)*. Prentice-Hall, 1992.

[4] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *USENIX 1998 Annual Technical Conference*, 1998.

[5] R. W. Carr and J. L. Hennessy. WSClock - a Simple and Effective Algorithm for Virtual Memory Management. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP-8)*, 1981.

[6] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33th Annual International Symposium on Computer Architecture (ISCA-33)*, 2006.

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

[8] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication and Capacity Allocation in CMPs. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, 2005.

[9] P. J. Denning. Thrashing: Its Causes and Prevention. In *AFIPS 1968 Fall Joint Computer Conference*, volume 33, pages 915–922, 1968.

[10] H. Dybdahl and P. Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA-13)*, 2007.

[11] H. Dybdahl, P. Stenstrom, and L. Natvig. A Cache-Partition Aware Replacement Policy for Chip Multiprocessors. In *ACM 2006 Conference on High Performance Computing (HiPC-13)*, 2006.

[12] R. Fagin and J. H. Williams. A fair carpool scheduling algorithm. *IBM Journal of Research and Development*, 27(2):133–139, 1983.

[13] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance Of Multithreaded Chip Multiprocessors And Implications For Operating System Design. In *USENIX 2005 Annual Technical Conference*, 2005.

[14] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[15] J. L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Transaction of Software Engineering*, 19(8), 1993.

[16] 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 *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (PACT-15)*, 2006.

[17] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of the 19th ACM International Conference on Supercomputing (ICS-19)*, 2005.

[18] R. Iyer. CQoS: a Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proceedings of the 18th ACM International Conference on Supercomputing (ICS-18)*, 2004.

[19] L. K. John. More on Finding a Single number to Indicate Overall Performance of a Benchmark Suite. *SIGARCH Computer Architecture News*, 32(1), 2004.

[20] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT-13)*, 2004.

[21] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA-10)*, 2004.

[22] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2001)*, 2001.

[23] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[24] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[25] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO-39)*, 2006.

[26] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: the Single-node Case. *IEEE Transaction of Networks*, 1(3):344–357, 1993.

[27] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. STATSHARE: A Statistical Model for Managing Cache Sharing via Decay. In *Second Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2006)*, 2006.

[28] M. M. Planas, F. Cazorla, A. Ramirez, and M. Valero. Explaining Dynamic Cache Partitioning Speed Ups. *IEEE Computer Architecture Letters*, 6(1), 2007.

[29] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO-39)*, 2006.

[30] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way Cache: Demand Based Associativity via Global Replacement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, 2005.

[31] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (PACT-15)*, 2006.

[32] J. E. Smith. Characterizing Computer Performance with a Single Number. *Communication of ACM*, 31(10), 1988.

[33] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[34] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2002.

[35] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, 2005.

[36] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transaction of Computers*, 41(9):1054–1068, 1992.

[37] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-aware Scheduling and Partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA-8)*, 2002.

[38] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[39] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy. IBM Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[40] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-memory Multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.

[41] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical Report MIT/LCS/TR-667, Cambridge, MA, USA, 1995.

[42] T. Y. Yeh and G. Reinman. Fast and Fair: Data-stream Quality of Service. In *Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'05)*, 2005.

[43] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled CMPs. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, 2005.