A Comparison of Capacity Management Schemes for Shared CMP Caches

Carole-Jean Wu and Margaret Martonosi
Department of Electrical Engineering
Princeton University
{carolewu, mrm}@princeton.edu

Abstract

In modern chip-multiprocessor (CMP) systems, multiple applications running concurrently typically share the last level on-chip cache. Conventionally, caches use pseudo Least-Recently-Used (LRU) replacement policies, treating all memory references equally, regardless of process behavior or priority. As a result, threads deemed high-priority by the operating system may not receive enough access to cache space, because other memory intensive, but lower-priority, threads are running simultaneously. Consequently, severe performance degradation and unpredictability are experienced. To address these issues, many schemes have been proposed for apportioning cache capacity and bandwidth among multiple requesters.

This work performs a comparative study of two existing mechanisms for capacity management in a shared, last-level cache. The two techniques we compare are static way-partitioning and decay-based management. Our work makes two major contributions. First, we make a comparative study demonstrating potential benefits of each management scheme, in terms of cache utilization and detailed intuition on how each scheme behaves. Second, we give performance results showing the benefits to aggregate throughput and performance isolation. We find that aggregate throughput of the targeted CMP system is improved by 50% using static way-partitioning and by 55% using decay-based management, demonstrating the importance of shared resource management in future CMP cache design.

1 Introduction

It is common to run multiple heterogeneous applications, such as web-server, video-streaming, graphic-intensive, scientific, and data mining workloads, on modern chip-multiprocessor (CMP) systems. Commonly-used LRU replacement policies do not distinguish between processes and their different memory needs. In addition, as the number of concurrent processes increases in CMP systems, the shared cache is highly contested. Thus, high-priority processes may not have enough of the shared cache throughout their execution due to other memory-intensive, but lower-priority, processes running simultaneously. The absence of performance isolation and quality of service can result in performance degradation.

To address these critical issues, various shared resource management techniques have been proposed for shared caches [3, 6, 8, 10–12, 14]. Techniques in both software and hardware have been investigated to distribute memory accesses to the shared cache from all running processes. In this work, we compare two of these mechanisms: static way-partitioned management and decay-based management.

Static way-partitioning has been widely used for cache capacity management because it has low hardware complexity and straightforward management. In static way-partitioned management, the set-associative shared cache is partitioned to various way configurations and available ways are allocated to each process based on its resource requirement and priority. A more detailed discussion is in Section 2.3.

Decay-based management offers a different strategy for cache capacity management. It is a hardware mechanism that interprets process priority set by the operating system and assigns a lifetime to cache lines accordingly, taking into account process priority and memory footprint characteristics. It is a fine-grained technique that adapts to each process' temporal memory reference behavior. A more detailed discussion is in Section 2.4.

The contribution of our work lies in the extensive and detailed study of shared resource management schemes. We offer a comparative study on the effectiveness of these techniques both qualitatively and quantitatively. We use a full-system simulator to *duplicate* each management scheme and evaluate performance effects taking into account operating system influence on the problem. In addition, we *deconstruct* each scheme to demonstrate its potential benefits in terms of cache utilization and offer a detailed intuition on how each scheme behaves. Finally, we show that aggregate throughput of the targeted CMP system is improved by *50%* using way-partitioning and by *55%* using decay-based management demonstrating the importance of shared resource management in future CMP system design.

The structure of this paper is as follows: In Section 2, we give an overview of shared resource management. Then, we discuss two shared resource management mechanisms: static way-partitioned management and decay-based management.

Time	Reference Stream		Outcome	
	P0 Low Priority	P1 High Priority	Pure LRU Replacement	Priority-Based Replacement
	Start with ACEF in the cache		H: Hit	M:Miss
1	B		M	M
2		A	M	M
3	B		H	M
4		C	M	M
5	B		H	M
6		E	M	M
7	B		H	M
8		F	M	M
9	B		H	M
10		A	M	M

Figure 1. An illustrative case study, where LRU Replacement Policy outperforms Priority-Based Replacement Policy.

This is followed by examples of memory reference streams which benefit from the studied management policies. In Sections 3 and 4, we describe our simulation framework, evaluate the shared resource management schemes, and analyze performance qualitatively and quantitatively. Then, in Section 5, we discuss related work on shared resource management. Section 6 discusses further issues and future work. Finally, Section 7 offers our conclusions.

2 Shared Resource Management

2.1 Overview and Goals

Performance isolation is an important goal in shared resource management. While multiple processes have accesses to the shared cache in CMP systems, it is possible that a process, e.g., a memory-bound one, uses the shared cache intensively, and, other processes, e.g., high-priority ones, are left with insufficient cache share throughout execution. This results in performance unpredictability. In order to provide performance predictability, particularly for high-priority processes, we have to prioritize accesses to the shared cache and isolate inter-process interference in the shared cache resource. Consequently, we can achieve *performance isolation* among all processes, and meet performance expectations.

Shared resource management refers to apportioning common resources among multiple requesters. In the case of heterogeneous applications running on CMP systems, multiple requests are made to the shared cache simultaneously. However, current systems cannot explicitly assign shared cache resources effectively based on process priority and characteristics of each process's memory footprints. Consequently, in order to arbitrate memory accesses from all running processes taking into account heterogeneity and process priority, shared resource management is critical.

In Section 2.2, we discuss a simple priority-based replacement policy and discuss why it is insufficient. Then, in Sections 2.3 and 2.4, we discuss static way-partitioned and decay-based management schemes in detail.

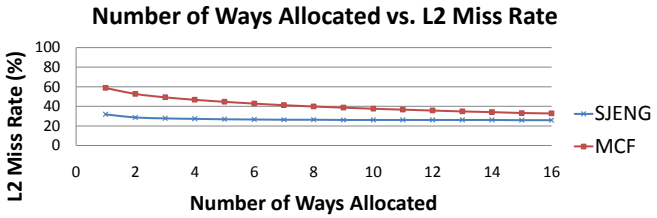

Figure 2. Various way configurations and miss rates: L2 miss rates for `sjeng` and `mcf` improve as number of ways allocated increases.

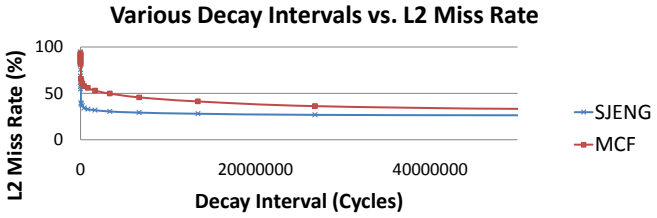

Figure 3. Various decay interval configurations: L2 miss rates for `sjeng` and `mcf` improve as decay intervals increase.

2.2 Why Not Use Priority-Based Replacement?

Priority-based replacement schemes preferentially evict cache lines associated with low-priority processes. This scheme favors cache lines associated with high-priority processes, and intends to provide certain performance expectation for high-priority applications while sacrificing tolerable amounts of performance for low-priority applications. However, this performance tradeoff between high and low-priority applications may not always pay off. In Figure 1, we illustrate a simple example of a memory reference stream, where LRU outperforms priority-based replacement policy.

Suppose that there are two running processes. The higher-priority process issues memory accesses to *A*, *C*, *E*, and *F*, and the lower-priority process issues memory accesses to *B*, as illustrated in Figure 1. All addresses are mapped to the same set of the 4-way set-associative shared cache.

While there are no hits for the higher-priority process in either replacement policy, 4 out of 5 memory references are hits for the low-priority process with the LRU replacement policy, but all 5 memory references are misses for the low-priority process with the priority-based policy. This is because priority-based replacement does not take into account the temporal behavior of memory references. In this case, we should recognize the significance of memory address *B* due its temporal locality. We use this example to illustrate that although priority-based replacement policy guarantees cache resource precedence to high-priority applications, it does not always translate to overall performance gain. As illustrated in the example, unnecessary performance degradation for the low-priority process is experienced.

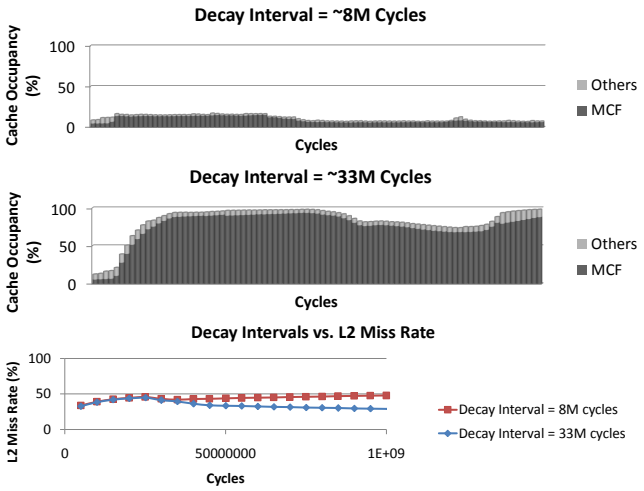

Figure 4. `mcf`**'s cache space utilization and miss rate for decay intervals set to 8M and 33M cycles**: As decay intervals increase from 8M to 33M cycles, `mcf` is allocated with 10 times more cache space and, correspondingly, its miss rate is improved by *20%*.

Step	Reference Stream		Outcome		
	P0 High Priority	P1 Low Priority	Pure LRU Replacement	Decay-Based Replacement	
	Start with ABCD in the cache		H: Hit M:Miss		
1					B decays
2	E		M	M	
3	A		M	H	
4					D decays
5		B	M	M	
6	C		M	H	
7					B decays
8		D	M	M	
9	E		M	H	
10	A		M	H	
11					D decays
12		B	M	M	
13	C		M	H	

Figure 5. An illustrative case study, where Decay-Based Replacement Policy outperforms LRU Replacement Policy.

2.3 Static Way-Partitioned Management

Various versions of static way-partitioning replacement mechanism have been proposed in the past to partition shared caches in CMP systems [3, 6, 8, 11, 12, 14]. Shared set-associative caches are partitioned to various way-configurations and are allocated to multiple processes. For example, given a 4-way set-associative cache and 2 active processes, the operating system can assign 3 ways to one process, and the remaining way to the second process, depending on the priority or cache resource requirements of each process.

Static way-partitioning is beneficial in several ways. First, it is straightforward to partition and allocate the shared cache capacity in the granularity of cache ways. The system can assign more ways of the shared cache to high-priority applications and less ways of the cache to low-priority applications. Static way-partitioned management can also be employed to ensure performance isolation: since each process has been allocated a certain amount of the shared cache for its exclusive use, its memory performance is not impacted by concurrently running processes.

Figure 2 illustrates that both `sjeng`'s and `mcf`'s miss rates are improved as the number of ways allocated to them increases. The number of ways can be assigned to a process based on its process priority accordingly. This demonstrates how performance predictability can be provided by static way-partitioned management technique.

Despite the advantages, static way-partitioning has two major drawbacks. First, in order to achieve the performance isolation discussed previously, it is preferable to have more ways in the set-associative cache than the number of concurrent processes. As the number of concurrent processes increases in today's CMP systems, this imposes a significant constraint to cache capacity management at the granularity of cache ways. The other drawback is inefficient cache space utilization, again due to the coarse granularity in space allocation.

2.4 Decay-Based Management

Decay-based management builds on the cache decay idea [5]. In a decay cache, each cache line is associated with a *decay counter* which is decremented periodically over time. Each time a cache line is referenced, its decay counter is reset to a *decay interval*, T, which is the number of idle cycles this cache line stays active in the cache. When the decay counter reaches 0, it implies that the associated cache line has not been referenced for the past T cycles and its data is unlikely to be referenced again in the near future. This timer signals to turn off the power supply of this cache line to save leakage power.

In shared resource management, however, this idea can be used to control cache space occupancy of multiple processes [9, 10]. When a cache line is not referenced for the past T cycles, it becomes an immediate candidate for replacement regardless of its LRU status. The key observation is that we can use different decay intervals for each process. This allows us to employ some aspects of priority-based replacement while also responding to temporal locality.

As before, the operating system assigns priority to active processes. Then the underlying cache decay hardware interprets process priority accordingly. It gives the decay counters of cache lines associated with compute-bound processes or high-priority processes a longer decay interval, so over time more cache resources are allocated to these processes. Similarly, the hardware may assign a shorter decay interval to cache lines associated with memory-bound processes or low-priority processes, in order to release their cache space more frequently. Consequently, higher-priority data tends to stay in the shared cache for a longer period of time.

Figure 3 illustrates how cache decay works for two applications taken individually. As decay intervals decrease, miss rates increase. This is because cache lines which are not referenced often enough decay from the cache more frequently.

In Figure 4, we demonstrate how decay intervals can be manipulated to control the amount of cache space an active process uses. As the decay interval increases from 8 million cycles to 33 million cycles, about 6 times more cache space is actively used by `mcf`. As a result, its miss rate is decreased by *15%*. Thus cache decay represents a fine-grained dynamic method for adjusting cache resource usage.

In Figure 5, we show how cache decay can be individualized to different applications in a mixed workload. Suppose memory addresses *B* and *D* are referenced by the lower-priority process. After some *T* cycles, *B*'s and *D*'s cache lines will decay and release their cache occupancy for replacement, as illustrated in Figure 5. We observe that with the LRU replacement policy all memory references are missed. In decay-based management, 5 out of 9 memory references are hits. More importantly, these hits are for the high-priority process. This is because cache lines associated with the high-priority process do not decay. In this example, LRU replacement works poorly because it treats all memory references equally. In contrast, decay-based replacement has more hits because it takes into account process priority.

While decay-based management is more complex than static way-partitioning, it may also have advantages. In particular, the shared cache space is utilized more effectively. Data remaining in the cache exhibits two critical characteristics: *high-priority* and *temporal locality*, as illustrated in the example in Figure 5. Although it offers useful fine granularity control, the decay-based management technique brings more hardware overhead.

3 Experimental Setup

3.1 Simulation Framework

We use GEMS [7], a full system simulator to evaluate both way-partitioned and decay-based management. We simulate a 16-core multiprocessor system based on the Sparc architecture running Solaris 10 operating system. Each core has a private 32KB level one (L1) cache and shares a 4MB level two (L2) cache. The L1 cache is 4-way set-associative and has block size of 64B. The shared L2 cache is 16-way set-associative and has block size of 64B. In our model, L1 cache access latency is 3 cycles, L2 cache access latency is 10 cycles, and L2 miss penalty is 400 cycles.

3.2 Workloads

We use the SPEC2006 benchmark suite to evaluate shared resource management mechanisms. First, in order to model a high contention scenario to the shared L2 cache, we simultaneously run multiple instances of `mcf`, a memory-bound application, along with one instance of `libquantum`, a non-memory intensive application. Second, in order to model memory accesses from heterogeneous applications in a multiprogramming environment, we use benchmarks from SPEC2006 CINT Benchmark Suite: `bzip2`, `mcf`, `sjeng`, `astar`, `libquantum`, `gcc`, `xalanc`, `hmmer`, `lbm`, `soplex`, `povray`, `omnetpp`, and `namd`. A brief description of the benchmarks can be found in [1].

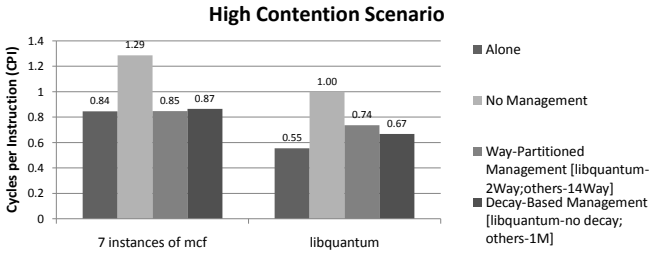

Figure 6. Management Policies and CPI for the high contention scenario.

4 Performance Evaluation

We evaluate both static way-partitioned management and decay-based management policies, and compare these two management policies to a baseline system which does not apply shared resource management for its level-two cache. We evaluate all possible configurations for static way-partitioning and pick the configuration giving the best average performance of all benchmarks. Similarly, in decay-based management we evaluate for several decay intervals ranging from *1000* cycles to *33,000,000* cycles and use the decay interval giving the best average performance of all benchmarks.

4.1 Results for High Contention Scenario

We model a high contention memory access to the shared L2 cache by running one instance of `libquantum` along with seven instances of `mcf` in a multiprogrammed environment. Throughout the simulation, there are eight active processes generating memory requests to the shared cache, in addition to the operating system scheduler process.

Figure 6 illustrates that when one instance of `libquantum` is running along with multiple instances of `mcf`, performance of `libquantum` is degraded by *80%* compared to running alone. Likewise the copies of `mcf` also show a *52%* degradation. This serious performance degradation is caused by `libquantum` and the multiple instances of `mcf` taking turns evicting each other's cache lines out of the shared cache repetitively.

Figure 7 depicts cache space utilization among all running processes. When there is no cache capacity management, all active processes compete for the shared L2 cache. As expected, most of the shared cache space is occupied by the multiple instances of `mcf`, which leaves an insufficient portion to

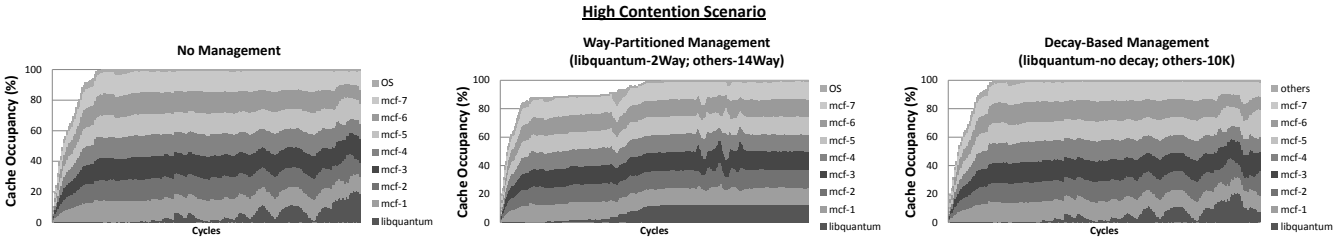

Figure 7. **Cache space utilization for the high contention scenario in baseline, static way-partitioning, and decay-based capacity management schemes.**

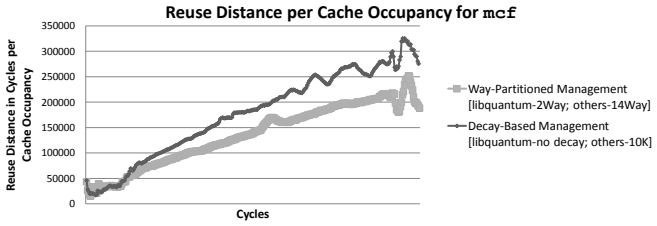

Figure 8. **Reuse-distance per cache space occupancy of** `mcf` **with decay-based and static way-partitioned management techniques.**

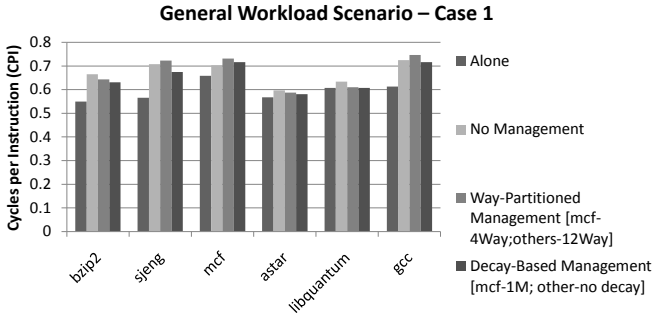

Figure 9. **Management policies and CPI for the general workload scenario – Case 1.**

`libquantum`.

In order to provide performance predictability for `libquantum`, performance isolation has to be enforced. In the static way-partitioned management scheme, `libquantum` is allocated with 2 ways of the shared cache exclusively, and other processes share the remaining 14 ways. In return, the performance of `libquantum` improves by *47%* compared to when no management is applied. More significantly, there is no performance degradation to the multiple instances of `mcf` compared to when the multiple instances are running alone. This is because the interference between `libquantum` and the multiple copies of `mcf` is eliminated completely in a static way-partitioning technique.

We next consider decay-based management. Here, cache lines associated with `libquantum` do not decay. For other processes, the decay interval is set to *10,000* cycles. This configuration is used to retain more of `libquantum`'s data in the shared cache. When decay-based management technique is applied, the performance of `libquantum` and multiple instances of `mcf` decreases by *20%* and *2%* respectively, compared to when running alone. Among all capacity management techniques that we have evaluated, the decay-based one works the best for `libquantum`.

As illustrated in Figure 6, we observe *13%* better performance of `libquantum` compared to the static way-partitioning technique and *2%* performance degradation for the multiple instances of `mcf`. This confirms what we have observed in Figure 8, where data from multiple `mcf` processes in the cache have more temporal locality in the static way-partitioning scheme.

Figure 8 illustrates the average cache line reuse distance in cycles and recognizes the temporal locality of data residing in different sizes of the shared cache. For the multiple instances of `mcf`, the static way-partitioning technique retains more temporal data in the shared cache than the decay-based management technique. This is because, in the high contention scenario, data in the shared cache must exhibit strong temporal locality already due to LRU replacement. As a result, `mcf`'s data in decay-based management scheme does not show as much temporal locality. This also indicates that the reuse distance of `mcf`'s cache lines is less than `mcf`'s assigned decay interval.

In this section, we have shown `libquantum` and the multiple instances of `mcf` experience severe performance degradation of *80%* and *52%* when there is no capacity management. This performance degradation is alleviated significantly when capacity management is applied. This is because memory footprint characteristics are taken into account for shared resource management and, in addition, performance isolation is considered.

4.2 Results for General Workloads

We model scenarios with general workloads using `bzip2`, `mcf`, `sjeng`, `astar`, `libquantum`, `gcc`, `xalanc`, `hmmer`, `lbm`, `soplex`, `povray`, `omnetpp`, and `namd`. In the first scenario, we demonstrate how static way-partitioned

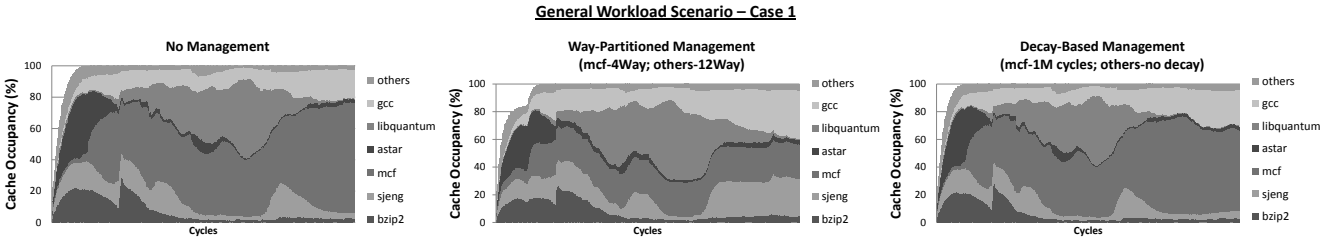

Figure 10. **Cache space utilization for the general workload scenario in baseline, static way-partitioning, and decay-based capacity management schemes.**

and decay-based management schemes can help constrain the resources devoted to high memory footprint applications. In the second scenario, we show how each management technique can be used to ensure enough of the shared cache is allocated to the high priority process.

4.2.1 Case 1: Constraining a Memory-Intensive Application

We run `bzip2`, `mcf`, `sjeng`, `astar`, `libquantum`, and `gcc` in a multiprogrammed environment to generate heterogeneous memory requests to the shared cache. Here we demonstrate how static way-partitioning and decay-based management schemes are used to isolate memory requests from a memory-intensive application, `mcf`, and to achieve performance isolation.

Figure 9 compares when each benchmark is running alone to when all benchmarks are running simultaneously without cache capacity management. Here we see that mixing applications causes performance degradation from *4%* for `libquantum` to *25%* for `sjeng`. This is because `mcf` is actively contending for the shared cache space; as a result, performance of other benchmarks is impacted.

Figure 10 illustrates the cache space distribution for all benchmarks under no capacity management and two cache capacity management schemes. In the beginning `bzip2`, `astar`, and `libquantum` occupy most of the shared cache space. Then `mcf` and `libquantum` start requesting more of the shared resource throughout the execution. This is the main cause of performance degradation experienced by other applications. Compute-intensive benchmarks, `bzip2`, `sjeng`, `astar`, `libquantum`, and `gcc`, are left with an insufficient portion of the shared cache space.

In order to isolate `mcf`'s memory interference on other running applications, we constrain its cache occupancy to 4 ways out of the 16-way set-associative cache, while other benchmarks share the remaining 12 ways. Figure 10 illustrates that `mcf` uses 25% of the shared cache exclusively and other benchmarks share the remaining portion. As a result of this constraint on `mcf`'s cache space, its performance is degraded by *5%* compared to when no capacity management is present. In return, the performance of `bzip2`, `astar`, and `libquantum` is improved. Next, consider a decay-based scheme.

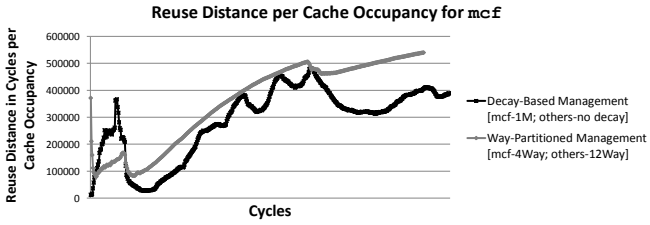

Figure 11. **Reuse-distance per cache space occupancy of `mcf` with decay-based and static way-partitioned management techniques**: Decay-based management technique prefers to retain data exhibiting more temporal locality than static way-partitioned management technique does.

In order to limit the amount of the shared cache that `mcf` occupies, a 1-million-cycle decay interval is assigned to cache lines associated with `mcf` and no decay is imposed on other benchmarks. We observe that a *2%* performance degradation is experienced by `mcf` compared to when there is no cache capacity management. In exchange, the performance of `bzip2`, `sjeng`, `astar`, `libquantum`, and `gcc` is improved by *7%*, *2%*, *2%*, *2%*, and *2%* respectively. Compared to static way-partitioning, `mcf`'s performance degradation is lessened because decay-based management retains more temporal reuse data in the shared cache. Figure 11 shows the average reuse distance per cache space occupancy for `mcf`. In the decay-based scheme, `mcf`'s data remaining in the shared cache exhibit more temporal locality than in the static way-partitioning scheme, as discussed previously in Section 2.4.

In this section, we have shown the benefits that constraining one memory-intensive application can have in a mixed workload environment. Although static way-partitioning achieves performance isolation between `mcf` and the other 5 applications, its coarse granularity of cache way allocation trades off *5%* performance degradation for `mcf` with an average of *1%* performance improvement for the rest of applications. In contrast, for the decay-based scheme, `mcf`'s performance is degraded by only *2%*, and the performance of the other 5 ap-

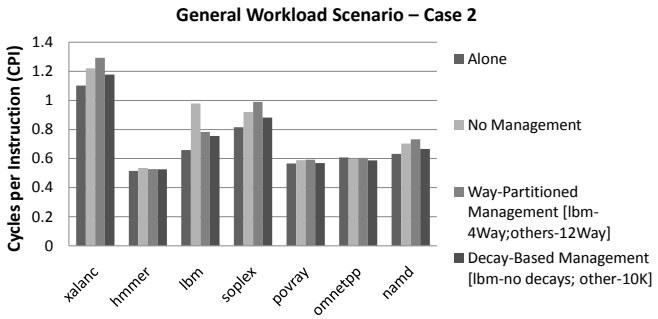

Figure 13. **Management policies and CPI for the general workload scenario – case 2.**

plications is improved even more. The decay-based scheme's benefits come from its fine granularity and improved ability to exploit data temporal locality.

4.2.2 Case 2: Protecting a High Priority Application

We use a different set of general workloads, `xalanc`, `hmmer`, `lbm`, `soplex`, `povray`, `omnetpp`, and `namd`, to model heterogeneous memory requests to the shared cache in a multiprogrammed environment. Here, assuming `lbm` is a high priority application, we demonstrate how static way-partitioned and decay-based management schemes can be used to allocate enough of the shared cache to `lbm` with minimum performance tradeoff for other concurrent processes.

Figure 13 compares each benchmark running alone to when all benchmarks are running simultaneously without cache capacity management. In the simultaneous case, performance degradation is experienced from *3%* for `hmmer` to *49%* for `lbm`. Performance impact to `lbm`, the high priority application, is the most severe among all applications. As illustrated in Figure 12, when there is no capacity management, `lbm` requests the most of the shared cache space in the first half of the execution. Then `xalanc`, `soplex`, and `omnetpp` start demanding more cache space in the second half of the execution, leaving `lbm` with less of the shared cache.

Next, consider static way-partitioning. In order to ensure enough of the shared cache space is allocated to the high priority application throughout execution, `lbm` is allocated 4 ways out of the 16-way set-associative cache. Other benchmarks share the remaining 12 ways. As a result, the performance of `lbm` is improved by *30%* compared to when no capacity management is applied, although there is an average of *3%* performance degradation among other benchmarks: *7%* for `xalanc`, *9%* for `soplex`, *1%* for `omnetpp`, and *4%* for `namd`.

In the decay-based scheme, cache lines associated with all benchmarks, except `lbm`, are set to decay every *10,000* cycles. Data associated with `lbm` does not decay. This leaves `lbm` more cache space allocation as needed throughout execution. Compared to when no capacity management is applied, performance of `lbm` increases by *34%*. At the same time, performance of other benchmarks improves by *3.5%* on average. Again, this performance gain comes from the fine granularity control of the decay-based management technique.

In this section, we have presented static way-partitioning and decay-based management techniques to protect a high priority application, `lbm`, from memory footprint interference of other concurrent applications. In this scenario, decay-based technique is a better capacity management choice because it not only provides enough of the shared cache space to the high priority application throughout the execution, but, at the same time, its fine granularity control of cache line allocation helps to improve performance of the other 6 applications as well.

4.3 Results Summary

We have presented how static way-partitioned and decay-based management schemes help distribute the shared L2 cache space effectively and achieve performance isolation, particularly for high priority processes.

The high contention scenario shows how each technique reduces interference of memory references between `libquantum` and the multiple instances of `mcf`. In the general workload scenarios, we demonstrate approaches for constraining and protecting individual applications. The decay-based management has fine granularity control that effectively partitions and distributes the shared cache space to requesting processes. Moreover, it retains data exhibiting two critical characteristics in the shared cache: *high priority* and *temporal locality*.

5 Related Work

5.1 Fair Sharing and Quality of Service

Many cache capacity management techniques have been proposed, targeting cache fairness [6, 12]. Thus far our work focuses mainly on process throughput. We discuss how static way-partitioned and decay-based management can be used to prioritize memory accesses based on process priority and memory footprint characteristics. Further cache fairness policies can be incorporated into both capacity management mechanisms discussed in this work.

Iyer [3] has focused on priority classification and enforcement to achieve differentiable quality of service in CMP systems. Both static way-partitioned and decay-based management mechanisms can be used to satisfy the desired quality of service goal discussed in Iyer's work. Similarly, Hsu et al. [2] have proposed performance metrics, such as cache miss rates, bandwidth usage, IPC, and fairness, to evaluate various cache policies. This additional information can assist the operating system to determine shared resource allocation better. Moreover, Iyer et al. [4] have suggested an architectural support for QoS-enabled memory hierarchy that optimizes performance of high priority applications with minimal performance degradation of low priority applications. Nesbit et al. [8] also address

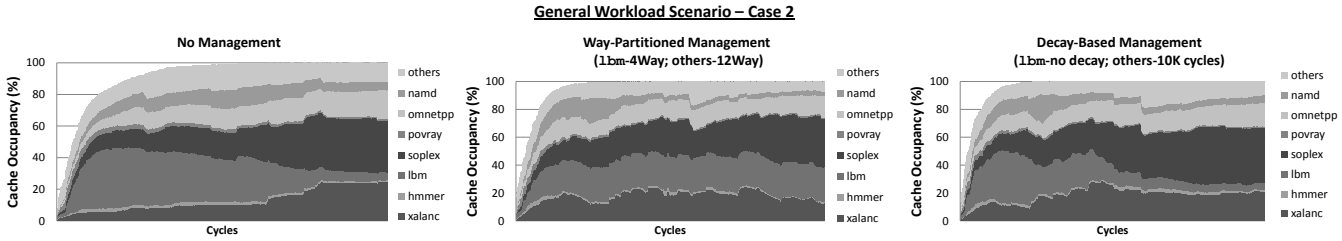

Figure 12. **Cache space utilization for the general workload scenario – case 2 in baseline, static way-partitioning, and decay-based capacity management schemes.**

resource allocation fairness in virtual private caches, where its capacity manager implements static way-partitioning.

5.2 Dynamic Cache Capacity Management

Dynamic cache capacity management has been initially proposed by Suh et al. [14]. In his proposal, the operating system distributes equal amount of cache space to all running processes, keeps cache statistics in flight, and dynamically adjusts cache space distribution among all running processes. This is a dynamic version of way-partitioned management. Other dynamic techniques based on way-partitioned management include [6, 11]. In addition to static way-partitioning mechanisms, Srikantaiah et al. [13] have proposed adaptive set pinning to eliminate inter-process misses, and hence to improve aggregate throughput in targeted CMP systems. Petoumenos et al. [10] offers a statistical model to predict thread behaviors in a shared cache and proposes capacity management through cache decay. To the best of our knowledge, however, there has not been any prior work based on decay management taking full system effects into account.

6 Discussion and Future Work

Thus far, we have presented two hardware mechanisms, static way-partitioning and decay-based management, which can be used to enforce cache capacity management. While the operating system offers more flexibility in defining quality of service goals, it plays a significant role in annotating its policies or specific goals to the underlying hardware mechanisms. The underlying hardware mechanisms then interpret the policies defined by the operating system and guarantee some level of quality of service by adjusting shared resource allocation. In the case of way-partitioning, concurrent processes can start with an equal number of cache ways allocated. Then the hardware can dynamically adjust cache way allocation to processes to fulfill quality of service goals defined by the operating system. Similarly, in the decay-based management scheme, the underlying hardware can vary decay intervals associated to each process in flight to satisfy policies specified by the operating system. While certain quality of service goals can be achieved by shared cache capacity management, strict application response time requirements remain a challenge.

7 Conclusion

In this work, we investigate a variety of shared resource management techniques and compare two of these mechanisms: static way-partitioned management and decay-based management. These two mechanisms adapt to unique characteristics of an application's memory footprints, take into account process priority, and distribute the shared cache space accordingly. We have offered illustrative examples of memory reference streams with a variety of replacement policies, and two mechanisms that effectively manage the shared cache. Our study shows that performance isolation is better achieved by the static way-partitioned management scheme, while temporal characteristics of applications are better captured by the decay-based management scheme in the general workload environment. In addition, although static way-partitioning has simple hardware complexity and a straightforward management, its coarse granularity of cache ways imposes a huge constraint on effective resource allocation. Whereas, decay-based technique offers a more flexible shared cache capacity management, as illustrated in our study.

Our simulation results demonstrate that in the high contention scenario, aggregate throughput of the targeted CMP system is improved by *50%* using static way-partitioning and by *55%* using decay-based management over a system without shared resource management. In the general workload environment, aggregate throughput of the targeted CMP system on average is improved by *1%* using static way-partitioning and by *8%* using decay-based management over a system without capacity management. Finally, we have offered a comparative study on when decay-based management technique is preferable to static way-partitioning, and demonstrated the importance of capacity management of the last level on-chip cache in future CMP cache designs.

8 Acknowledgements

This work was supported in part by the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. In addition, this work was supported by the National Science Foundation under grant CNS-0720561.

References

[1] Standard Performance Evaluation Corporation. Spec benchmarks. http://www.spec.org/cpu2006/CINT2006/, 2006.

[2] 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 *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 13–22, New York, NY, USA, 2006. ACM.

[3] R. Iyer. CQoS: a framework for enabling QoS in shared caches of cmp platforms. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266, New York, NY, USA, 2004. ACM.

[4] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2007. ACM.

[5] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 240–251, 2001.

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

[7] M. M. K. 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. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[8] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. *SIGARCH Comput. Archit. News*, 35(2):57–68, 2007.

[9] P. Petoumenos, H. Z. G. Keramidas, 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.

[10] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. Modeling cache sharing on chip multiprocessor architectures. *2006 IEEE International Symposium on Workload Characterization*, pages 160–171, Oct. 2006.

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

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

[13] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *SIGARCH Comput. Archit. News*, 36(1):135–144, 2008.

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