Protection, Utilization and Collaboration in Shared Cache Through Rationing

Raj Parihar Jacob Brock[†] Chen Ding[†] Michael C. Huang
{parihar@ece. jbrock@cs. cding@cs. michael.huang@}rochester.edu

Dept. of Electrical & Computer Engineering [†]Dept. of Computer Science

The University of Rochester
Computer Science Department
Rochester, NY 14627

Technical Report TR-995

November 2013

Abstract

Shared cache is generally optimized to maximize the overall throughput, fairness, or both. Increasingly in shared environments, especially compute clouds, users are unrelated to one another. In such circumstances, an overall gain in throughput does not justify an individual loss. This paper explores conservative sharing, which protects the cache occupancy for individual programs, but still enables full cache sharing whenever there is unused space.

Specifically, we present a new hardware based mechanism called cache *rationing*. Each core/program is assigned a portion of the shared cache as its ration. The hardware support protects the ration so it cannot be taken away by peer programs while in use. However, a program can exceed its allocated ration only if another program has unused blocks in its ration. This paper shows that rationing provides both full protection and full utilization of the cache. In addition, the same hardware support can enable energy-efficient caching and hardware-software collaborative caching.

The University of Rochester Computer Science Department supported this work.

1 Introduction

As multi-core processors become commonplace and cloud computing gains acceptance, more applications are run sharing the same cache hierarchy. Managing cache sharing is crucial not just for achieving good performance, but also for ensuring stable performance in a dynamic environment; and not just for parallel programs but also for sequential programs co-run with each other.

The basic task of cache management is to allocate cache blocks to co-running applications. There are two extremes in this allocation: cache partitioning, and free-for-all cache sharing. Taking an analogy with resource allocation in economics, Hsu *et al.* called them *communist* and *capitalist* policies (Hsu et al., 2006).

Neither strategy is perfect; Much previous work has devised more effective strategies of sharing (Section 5). The goal has been to optimize throughput, fairness, QoS, or some combination of the three. The mechanism is adaptive sharing — giving some programs more cache and other programs less. These solutions are *optimistic* since the allocation is based on the prediction of the favorable aggregate performance.

One problem with optimistic sharing is that it is intrusive; that is, it re-allocates space based on the collective needs of all programs, even if certain individual programs are hurt. In this paper we explore the approach of *conservative* sharing, in which cache re-allocation is only done if no program is hurt by it.

Increasingly in shared environments, especially compute clouds, users are unrelated. A collective gain does not justify an individual loss. Conservative sharing is to ensure the resource allocation for non-cooperative users yet still enable resource sharing when possible.

We present a hardware technique called *cache rationing*. Each program is allocated a ration which can be shared, but only under a strict condition. In particular, a program can share the cache ration of a peer program only if that peer program does not need its full ration. If a program needs all of its ration, no other program can take its space away, and if it does not, it shares the surplus space. In this way, rationing provides full protection and full utilization.

Hardware support is needed to solve two problems. The first is accounting, *i.e.*, keeping track of how much cache space is used by each CPU core. The second is usage tracking, *i.e.*, whether a program needs all its ration. To solve the second problem, we use a solution similar to OS memory allocation. In particular, an access bit to indicate whether a cache line has been actively used in recent past.

Traditionally, cache is managed by usage. All real machines approximate LRU. For protection, we must allocate cache based on who accesses the data rather than when the accesses happen. By using the per-cache-line access bit, we will show that cache rationing can combine entitlement and usage-based cache allocation, and hence obtain the benefit of partitioning and sharing while avoiding their disadvantages.

The rationing support also allows hardware-software collaborative caching (Wang et al., 2002). In particular, we add a single bit to indicate a special memory load and store. A special operation marks the cached data as "evict-me" by clearing the access bit upon the

special access. We will show that the interface allows collaborative caching and can further improve the performance of rationed cache.

Continuing with the free-market analogy of Hsu *et al.*, let's consider an economic concept called the Pareto optimality, for example in income distribution. A resource allocation is optimal if *one cannot make any one individual better off without making at least one individual worse off.* Cache rationing is to effect Pareto optimality for cache allocation. Sharing of an unused ration is a Pareto improvement, since it helps a program without making anybody else worse off. Hardware-software collaboration is also a Pareto improvement, since one program can optimize the use of its ration without affecting others.

The rest of the paper is organized as following: Section 2 presents the details of our proposed cache rationing mechanism along with the storage overhead needed to implement the proposed solution. Section 4 presents the experimental setup and discusses the performance results. Section 5 overviews related work and finally Section 6 summarizes the paper.

2 Cache Rationing

A ration for a CPU core or program is the guaranteed effective size of space in the shared cache that is allocated to the core/program. The ration can be specified by software, *e.g.*, through privileged instructions. We define two new objectives in cache sharing and then show how cache rationing is designed to meet these objectives.

2.1 Objectives of Rationed Cache

Traditionally, in science and engineering computing we strive to maximize performance. In a shared facility, *e.g.*, a university computing center, we also strive to ensure fairness. In such cases, the hardware is fully owned by the user or the organization who is running the applications.

Cloud computing operates on a retail model; cloud processors are rented on the open market. A user temporarily takes ownership of part of a machine. For example Amazon Elastic Compute Cloud (EC2) lets a user rent a virtual computer. The size of the computer is elastic and measured by how fast it is. The speed is measured by *elastic compute units (ECU).* Amazon defines the ECU by the average performance of a standard CPU configuration.[1]

In the cloud context, the fairness and efficiency are different than in the workstation and computing center. We next define two new objectives and compare them with throughput and fairness.

[1] According to Amazon, "We use several benchmarks and tests to manage the consistency and predictability of the performance of an EC2 Compute Unit. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. This is also the equivalent to an early-2006 1.7 GHz Xeon processor referenced in our original documentation." (source: EC2 wiki page)

Resource Protection vs. Fairness The rented computing power is measured by the dedicated use of hardware, including the size of cache. When sharing a processor among multiple users, the goal is not to evenly divide the shared cache but to guarantee that a rented core has cache at or above the specified size. We call this guarantee *resource protection*.

Resource protection is not the same as fairness. In fairness, a group of tasks have an equal partition of the cache (resource fairness), or an unequal partition so they have a similar performance loss (performance fairness). In resource protection, we reserve a cache partition at least the size of the ration, regardless of the demand by peer programs.

Rationing guards against unexpected performance loss. Based on total demand, a user may reduce the ration given to a program. There is a performance loss due to reduced ration, as happens when more programs are added to share the same machine. But unlike traditional cache sharing, cache rationing bounds the worst-case resource allocation and the worst-case loss. It provides a "safety net" for performance.

Utilization vs. Throughput A program may not use all of its ration. It depends on the cache demand of the program, and the size of ration it is given. When there is unused ration, we want to utilize it if there is a co-run program that can benefit from having more cache space. We call the amount of cache used by a program its *cache utilization*.

Maximal utilization is different from maximal throughput. To maximize overall performance, we allocate each chunk of cache space to the program that can gain the most by having the additional space. To maximize utilization, we detect unused rations and make them and only them available for sharing. For throughput we need global optimization, while for utilization we need just local detection.

Quality of Service Cache rationing helps to support quality of service (QoS), if the quality is defined by a hardware configuration. For QoS, we need dedicated allocation of resources including CPU and memory and I/O bandwidth. Rationing provides guaranteed cache allocation.

Cache rationing may be used to obtain the traditional goals of throughput or fairness, which would require additional analysis and optimization to convert performance requirements into resource requirements. The same goes for the QoS optimization, *i.e.*, minimal resource use for a bounded loss. This paper focuses on the hardware support for cache rationing.

2.2 Ration Counter

To implement rationing, we add two sets of additional storage: a *ration counter* for each core or application, and a *ration owner* for each cache block.

A ration counter stores two integer values: the number of cache blocks rationed for the counter owner, and the number of resident cache blocks that were loaded into the cache by the counter owner. In the following, we show rationing among CPU cores. With the OS support, the same rationing can be done for tasks or task groups.

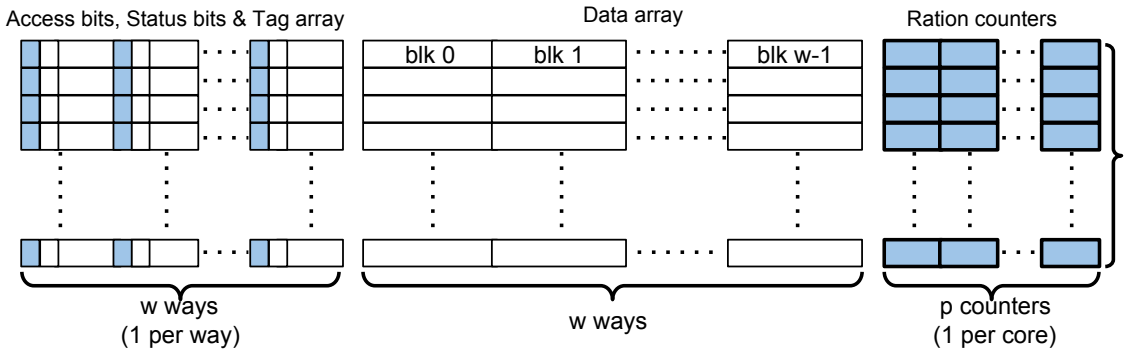


Figure 1: The space diagram of cache showing ration counters (one per set per core) and access bits (one per cache block).

Each cache block stores its "ration owner", which is a reference to one of the ration counters. For example, in 4-core sharing, the owner record needs 2 bits per cache block to indicate which core owns the block as part of its ration. It also needs a way to indicate if a valid block is unowned. One solution is to have a default ration counter that tracks all unowned blocks.

Rationing can be done in two scopes. The first is across the whole cache. Each core has a total allocation. The second is within each cache set. Each core has an allocation in every set. Whole-cache allocation allows more flexible partitioning since a ration can be any integer between 1 and the cache size. It needs just one set of ration counters. However, it has weaker protection against interference since local hotspots can happen. Cache-set allocation provides fine-grained protection but requires a set of ration counters in every cache set. We will compare these two flavors of rationing in evaluation section.

```
# blk is fetched by p at a miss
fetch( blk, p )
  blk.owner's_counter = p.ration_counter
  blk.owner's_counter ++

# blk is evicted
evict( blk )
  access( blk, p )
  blk.owner's_counter --

# blk is accessed by p
access( blk, p )
  if blk.owner's_counter != p.ration_counter
    blk.owner's_counter --
    blk.owner's_counter = p.ration_counter
    blk.owner's_counter ++
```

Figure 1 shows the space diagram of a cache set. The ration counters are shown in the shaded (blue) color on the right side.

The maintenance of the ration count requires knowledge of the identity of the CPU core responsible for each memory access. The maintenance logic is shown in the pseudo code as follows. There are three cases: a cache load (upon a miss), an eviction, and a normal access (a hit).

At a cache load, we set the owner record to point to the ration counter of the loader and increment it. At an eviction, we follow the owner record to decrement the ration counter.

There are two cases when the ration usage becomes inconsistent. The first is when there is per-core rationing but a task migrates. The second is running multi-threaded code with data sharing. In both cases, a block is loaded by one core but evicted by another core. The problem may be solved in two ways.

The first is counter association. If we assign a ration counter per task, then migration does not cause inconsistency. If we assign the same ration counter for all threads of a program, then the problem is solved for data sharing. We can design more elaborate schemes where the ration counters are dynamically coalesced and split.

The second solution is ownership update. At a cache (hit) access, we check whether there is a mismatch and update the owner record accordingly. The preceding pseudo code shows the second solution.

2.3 Access Bit

The second hardware extension is the access bit for detecting an unused ration. Each block has an access bit, as shown in Figure 1. It is set whenever the block is referenced. Periodically, the access bits are reset. The time between consecutive resets is called the *reset interval*. A rationed cache block is deemed unused if either it is not owned or the access bit is zero.

2.4 Rationing Control

We augment the cache management logic to implement rationing. The description is based on set-associative LRU cache. Adaptations in other types, *e.g.*, pseudo LRU, can also be made. The description here focuses on how to use the ration counter and access bit.

LRU is a stack algorithm and can be modeled by a stack storing the most recently accessed block at the top and down to the least recently accessed block at the bottom.

For each ration, we keep track of the least recently used block (LRU block) whose access bit is one. For each cache hit, the stack is adjusted as in a traditional cache. In addition, the rationed cache records the LRU block for each ration (whose access bit is 1).

The following algorithm shows the replacement logic at a miss. For simplicity, we assume two programs, $p1$ and $p2$, share the cache. Without loss of generality, let the miss be caused by $p1$.

The problem is finding the victim at a miss. The preceding algorithm first checks whether there is a cache block that is invalid or valid but unused (access bit is 0). If there is none, it checks if $p1$ is at or over its ration. If both checks are false, then $p2$ is over its ration.

In a two-core system, checking if a program is over its ration is easy. In the general case, the else clause needs to find in a set of cores one that is over ration. This can be done by an associative search and picking the counter with highest overuse of its ration.

```
# p1 needs to load blk at a miss
miss( blk, p1 )
 if repl exist s.t. !repl.valid or !repl.accessed
   replace repl
 elsif p1.at_or_over_ration?
   replace LRU block of p1
 else  # p2 over ration
   replace LRU block of p2
```

2.5 Example Illustration and Comparison

Rationing has two goals: cache resource protection, and cache resource utilization. We show an example of each case in Figure 2. Assume we have two cores, each accessing a separate set of data, and an evenly rationed cache with two blocks for each core.

The figure shows the access trace for each core on the left hand side. It also shows the content of access bits (one for each block, 4 total) and ration counters (one for each core, 2 total). Not shown is the owner pointer pointing to one of the two ration counters. In the interleaved execution, if two requests come at the same time, we arbitrarily assume that the cache sees the request from core 1 first.

The first example (Figure 2(a)) shows resource protection. In this case, core 1 uses 2 blocks, which it can hold entirely within its ration. However, in free-for-all sharing, *i.e.*, the capitalist policy, data from core 2 can evict data used by core 1. In contrast, the partitioned cache (communist) and the rationed cache do not permit core 2 to intrude on the ration of core 1.

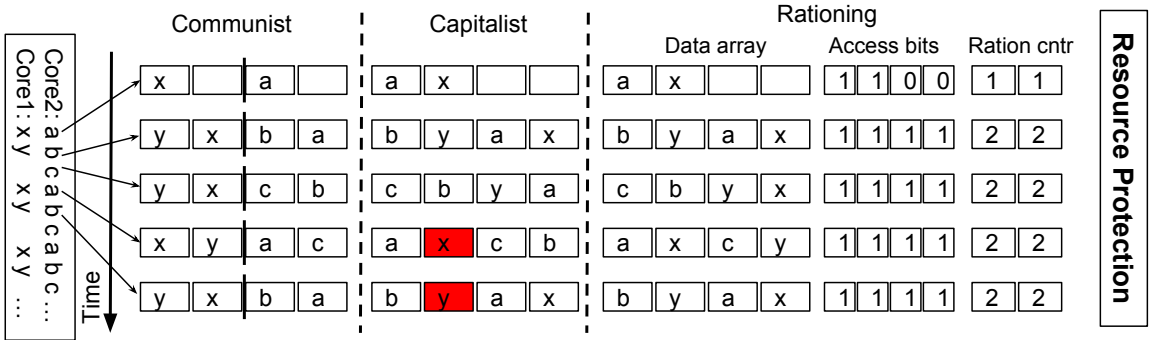
Due to the lack of protection, the capitalist policy causes the most misses. In the figure, we shade the misses (excluding compulsory ones). Partitioning and rationing perform equally well by providing resource protection. However, the mechanisms are different. Rationing permits sharing, as the next example shows.

The second example (Figure 2(b)) shows cache utilization. If core 1 uses just 1 block, and core 2 uses 3, the fixed partitioning would under-utilize the partitioned space for core 1. The capitalist sharing and the cache rationing, in contrast, can fully utilize the 4 cache blocks for the 4 program blocks.

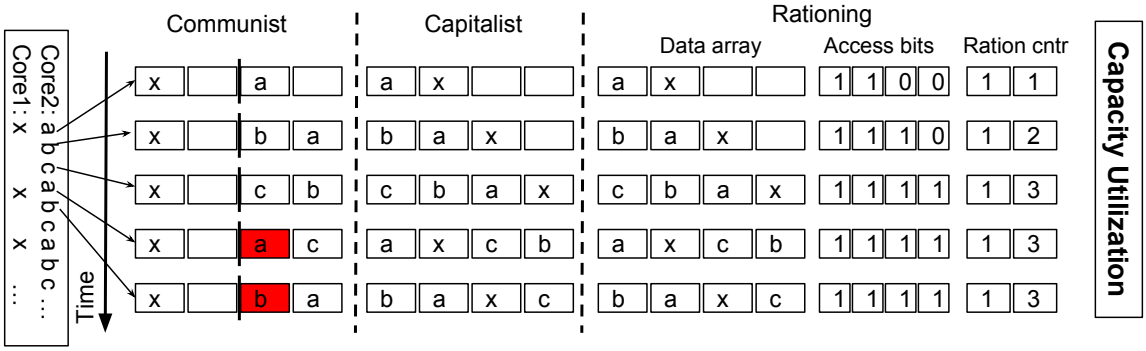
The two examples demonstrate that cache rationing can combine the advantages of cache partitioning and sharing while avoiding their problems.

3 Interaction with Other Optimizations

Our proposed rationing based design lends itself to integration with other designs. In this section, we discuss some of the designs and the ease of integration with rationing technique.



(a) Rationing performs as well as partitioning and better than sharing because rationing protects core 1 against the interference by core 2.



(b) Rationing performs as well as sharing and better than partitioning because rationing utilizes the unused ration of core 1.

Figure 2: Resource protection (a) and utilization (b) in evenly rationed cache, in comparison with communist (hard partitioning) and capitalist (free-for-all sharing) policies.

3.1 Hardware-Software Collaborative Caching

A number of processors provide special load/store instructions that a program can use to influence hardware cache management. These include the placement hint on Intel Itanium (Beyls and D'Hollander, 2005), bypassing access on IBM Power series (Sinharoy et al., 2005), the evict-me bit (Wang et al., 2002), and non-temporal instructions on IBM and Intel processors (Rus et al., 2011; Yang et al., 2011; Brock et al., 2013). Wang *et al.* called a combined software-hardware solution *collaborative caching* (Wang et al., 2002). Here we call a special memory instruction a *cache hint*.

There are two common uses of cache hints. The first is to mark accesses whose data will have no chance of reuse before eviction. This can be done using compiler analysis (Wang et al., 2002; Beyls and D'Hollander, 2005) or reuse-distance profiling (Beyls and D'Hollander, 2005). Taking the hints, the hardware would choose not to cache those data and hence save the space for data that may be reused. A second solution was developed recently to use the OPT stack distance (*i.e.*, the OPT stack position) instead of the reuse distance (LRU stack distance) as an indicator of whether a block should be marked for eviction (Brock et al.,

Thread 1	a	b	c	a	b	c	a	b	c
Hint Bit	0	1	0	1	0	1	0	1	0
Access Bit	1	0	1	0	1	0	1	0	1
Misses	M	M	M		M		M		M
Thread 2	x	y	z	x	y	z	x	y	z
Hint Bit	0	1	0	1	0	1	0	1	0
Access Bit	1	0	1	0	1	0	1	0	1
Misses	M	M	M		M		M		M
Post-Access Cache Content	a	b	c	a	b	c	a	b	c
	x	y	z	x	y	z	x	y	z
		a	a	c	c	b	b	a	a
		x	x	z	z	y	y	x	x

Figure 3: An example of cache rationing with a hardware-software collaboration hint bit. If the hint bit is set, the access bit is zeroed so that the accessed blocks will not be kept in the cache. The contents of the cache are shown after each pair of accesses, and blocks with their access bit zeroed are underlined.

2013). The benefit of this approach is that it can select a part of the working set to cache if the whole working set is too large.

Regardless of what the software does, it needs the hardware instruction in order to mark a data block and tell the hardware to replace it before replacing other blocks. Such an instruction can be readily supported by cache rationing.

We add a hint bit to load/store instructions. At the access, the processing is exactly as we have defined before. The only effect happens when setting the access bit. In the default logic, the access bit is set after the access. With the new interface, the access bit is set only if the hint bit is not. In other words, the software can tell the rationing hardware not to set the access bit if it knows that the block will have no more cache reuse, or if its eviction would free cache space for other blocks. The block then becomes unused ration and will be favored for immediate eviction (before every block whose access bit is 1).

As an example, consider two cores sharing a four-block cache. Let the access traces be "xyzxyz..." for one core and "abcabc..." for the other. With equal rationing, neither core has enough cache to obtain any reuse. However, with cache hints, the software can free up cache space by zeroing some access bits (where the hint bit is set). In Figure 3, every other access has its hint bit set, so the access bit is zeroed. In this case, the non-compulsory miss ratio is reduced from 1 to 1/2. In (Gu et al., 2008), it is shown that a hint-based solution can achieve optimal caching, and its application for single threads is demonstrated in (Brock et al., 2013).

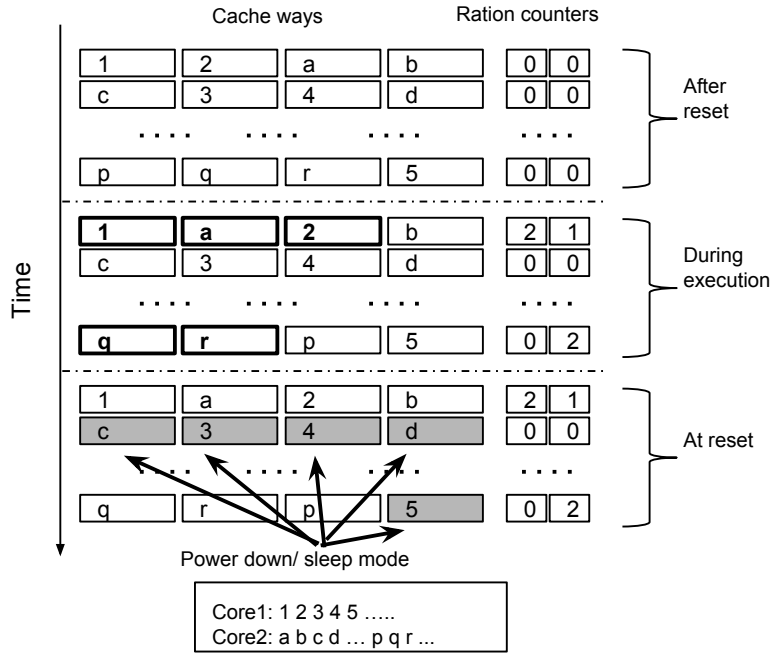


Figure 4: Energy efficient rationing: after all the ration counters and access-bits are reset, a still zero-valued ration counter at next reset point can be used to either evict the lines belonging to respective core or send them to a power-saving mode to reduce leakage.

3.2 Energy Efficient Rationing

Kaxiras *et al.* have showed that cache accesses exhibit generational behavior (Kaxiras et al., 2001). When we access a cache line then typically the first access is a miss, followed by a series of hits. When the line is dead, it relies on LRU to evict it even thought it can be evicted right away. The study shows that about 80% of the time a cache line is dead but waiting for eviction. Many hardware designs (Flautner et al., 2002; Kaxiras et al., 2001; Khan et al., 2010) have been proposed that are built to predict the last reuse of a cache line and then either evict the line immediately or shut off the ways to save leakage power.

We can leverage the cache rationing support for dead block eviction to save energy. To enable energy efficient cache rationing, we use the ration counter for two purposes. As shown in the Figure 4, when resetting the access bits, all the ration counters are also reset. During the execution, an access increments the respective ration counter. At the next reset point, if a ration counter is still zero then it indicates that no access was made to this particular set by this core. This information can be used to evict all the lines that belong to the core with zero ration counter value. With our preliminary analysis we have found that it is safe to use the access pattern of previous reset interval to predict the access behavior of the next reset interval with good accuracy.

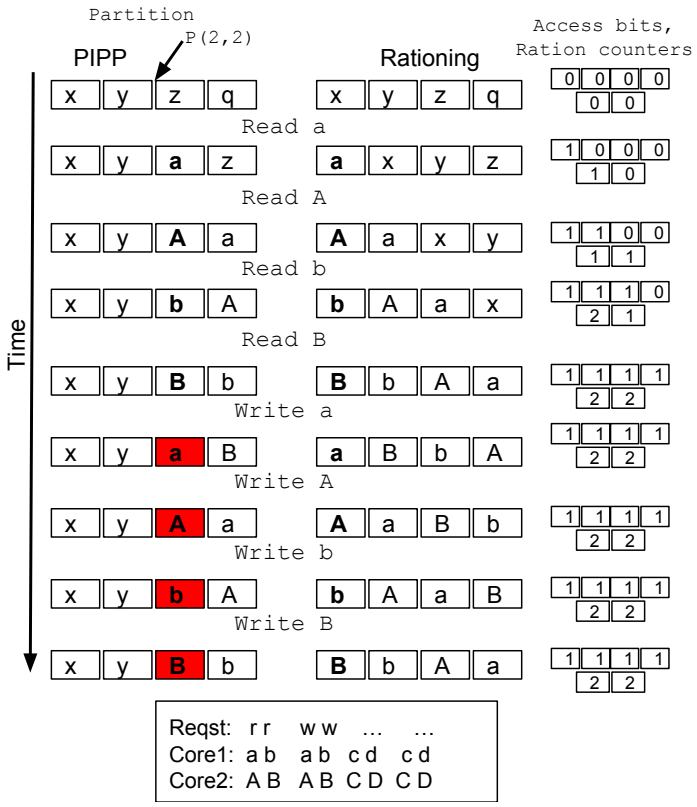


Figure 5: Comparison of cache rationing with baseline PIPP policy. PIPP-equal (PIPP with equal partitioning) for symmetric co-run incurs more misses and allows dead line to stay and increases more contention on the second half of the set. shaded blocks in red are the ones that incur conflict misses.

3.3 Comparison with Promotion/Insertion Pseudo Partitioning (PIPP)

In this section, we compare and differentiate our technique with several other designs. The recently proposed PIPP design (Xie and Loh, 2009) tries to achieve partitioning with the help of intelligent insertion and promotion policies. Because PIPP does not explicitly and pro-actively partition the cache, it is pseudo-partitioning as the name suggests. The baseline PIPP design works as following: For n cores, it assumes that there exists a set of target partitions $P = \{p1, p2, ..., pn\}$ such that $\sum pi = w$, where w is the set associativity of the cache. Simple baseline PIPP implements three policies. On insertion, $core_i$ simply installs all new incoming lines at priority position pi. On a cache hit, the promotion policy for PIPP promotes the cache line by a single priority position. Finally, the victim selection always chooses the line from the lowest-priority position – similar to conventional LRU. PIPP does not strictly enforce the target partitioning and does not guarantee resource protection, but the combination of targeted insertion and incremental promotion creates results similar to an explicit partition.

In a symmetric co-run where multiple copies of same application are sharing the cache, PIPP can effectively reduce the cache capacity and may allow certain dead blocks to occupy

the first half of a cache set forever while the other half of the cache experiences more conflict misses. This case is shown in the Figure 5; The rationing based policy evicts the dead blocks more effectively then PIPP. Most of the cache partitioning proposals (Qureshi and Patt, 2006; Xie and Loh, 2009; Jaleel et al., 2008) do not guarantee resource protection and the overall speedup may come at the cost of slowing down the less aggressive thread/program. In contrast, our rationing based cache policy does not slow down less aggressive programs and ensures resource protection.

4 Evaluation and Analysis

4.1 Experimental Setup

We perform our experiments using an in-house simulator that models true execution-driven, cycle-accurate simulation. We also model support for multi-program workloads on chip multiprocessors with extensive coherence transitions including faithful modeling of transient states.

Microarchitecture and configuration The simulator models major microarchitectural components such as issue queues, register renaming, ROB, and LSQ. Features like load-hit speculation (and scheduling replay), load-store replays, keeping a store miss in the SQ while retiring it from ROB are all faithfully modeled. Our baseline core is a generic out-of-order microarchitecture loosely modeled after POWER5 (Sinharoy et al., 2005). Other details of the configurations are shown in the Table 1.

Applications and inputs We use applications from SPEC CPU 2000 and 2006 benchmark suites compiled for Alpha using a cross-compiler that is based on gcc-4.2.1. We use *ref* inputs and simulate *at least* 200 million instructions per core after skipping over the initialization portion as indicated in (Sherwood et al., 2002). If an application completes 200 million instruction it continues to run until the last one completes 200 million instructions.

Core and Cache Configurations	
Fetch/Decode/Issue/Commit	8 / 4 / 6 / 6
ROB	128
Functional units	INT 2 + 1 mul + 1 div
	FP 2 + 1 mul + 1 div
Fetch Q/ Issue Q / Reg. (int,fp)	(32, 32) / (32, 32) / (80, 80)
LSQ(LQ,SQ)	64 (32,32) 2 search ports
Branch predictor	Gshare – 8K entries, 13 bit history
Br. mispred. penalty	at least 7 cycles
L1 data cache (private)	32KB, 2-way, 64B line, 2 cycles
L1 inst cache (private)	32KB, 2-way, 64B, 2 cycles
L2 cache (shared)	512KB per core, 8-way, 64B, 15 cycles
Memory access latency	150 cycles

Table 1: Microarchitectural configurations.

SPEC CPU2000 applications	
INT	1-gzip 2-vpr 3-gcc 4-mcf 5-crafty 6-parser 7-eon 8-perlbmk 9-gap 10-vortex 11-bzip2 12-twolf
FP	13-wupwise 14-swim 15-mgrid 16-applu 17-mesa 18-galgel 19-art 20-equake 21-facerec 22-ammp 23-lucas 24-fma3d 25-sixtrack 26-apsi
SPEC CPU2006 applications	
INT	1-perlbech 2-bzip2 3-gcc 4-mcf 5-gobmk 6-hmmer 7-sjeng 8-libquantum 9-h264ref 10-omnetpp 11-astar 12-xalancbmk
FP	13-bwaves 14-milc 15-zeusmp 16-namd 17-dealII 18-soplex 19-lbm 20-sphinx3

Table 2: SPEC 2000 and 2006 applications used in experiments and their index used in the result figures.

This ensures that other co-running applications continue to interfere for the whole execution of any specific application. For performance comparison, we account only 200 million instructions. In our experiments, many applications often go well beyond one billion instructions. Due to space constraints we cannot label the application names in the figures in Section 4.4 so we use numbers to denote the applications. Corresponding applications are shown in Table 2.

Co-run Test Suites For k programs, there are k^m choices for m-program co-runs. We cannot test them exhaustively so we choose the symmetric tests plus a few groups that require protection (co-runs with *equake* and *mcf*) and offer opportunities of utilization (co-runs with *eon*). In particular, we present results for five pair-run test suites:

- *SPEC 2000/2006, symmetric:* each programs co-run with itself.
- *SPEC 2000 with eon/equake/mcf:* either *eon* or *equake* or *mcf* co-runs with each of SPEC 2000 programs including itself.

We also create 2 four-program co-run test suites:

- *SPEC 2000, symmetric:* each of the 26 programs co-runs with three clones of itself.
- *SPEC 2000 with 2 equake:* for each of SPEC 2000 programs, we run two clones of the program and two *equake*.

Comparisons We compare rationing with partitioning (communist), sharing (capitalist) and a flavor of pseudo-partitioning (PIPP). For PIPP policy we assign the partition equally in a 2 program co-run and call it *PIPP-equal*. This is a reasonable design choice for symmetric co-run because the two copies of program are identical. For asymmetric co-run, PIPP will require additional profiling to estimate the right partition that is typically based on working set and other information. Because for rationing we use equal rations even for

Apps	**Co-run damage (Unhealthy co-run)**: 2-core system							
	Communist		Capitalist		Rationing		PIPP-equal	
	Damaged pairs	Average slowdown	Damaged pairs	Average slowdown	Damaged pairs	Average slowdown	Damaged pairs	Average slowdown
spec2k - symmetric	1	1.03%	15	3.34%	1	1.01%	16	3.47%
spec2k - with eon	0	0.0 %	9	5.65 %	0	0.0%	6	4.23%
spec2k - with equake	1	1.05%	17	9.76%	3	1.16%	15	9.21%
spec2k - with mcf	0	0.0 %	24	10.54 %	2	1.77%	21	8.64%
spec2k6 - symmetric	0	0.0%	7	4.09%	0	0.0%	–	–
	4-core system							
spec2k - symmetric	1	1.07%	13	4.52%	1	1.14%	–	–
spec2k - w/ 2 equake	3	1.33%	20	9.67%	2	4.82%	–	–

Apps	**Damage-free gain (Healthy co-run)**: 2-core system							
	Communist		Capitalist		Rationing		PIPP-equal	
	Healthy pairs	Average speedup	Healthy pairs	Average speedup	Healthy pairs	Average speedup	Healthy pairs	Average speedup
spec2k - symmetric	0	0.0%	1	2.51%	2	1.67%	0	0%
spec2k - with eon	0	0.0%	7	9.38%	14	14.46%	9	9.44%
spec2k - with equake	0	0.0%	5	4.64%	5	6.73%	3	3.56%
spec2k - with mcf	0	0.0% %	1	3%	1	26.12%	0	0.0%
spec2k6 - symmetric	1	2.19%	0	0.0%	1	2.25%	–	–
	4-core system							
spec2k - symmetric	1	1.17%	2	18.75%	2	23.73%	–	–
spec2k - w/ 2 equake	1	1.47%	4	6.59%	6	7.57%	–	–

Table 3: Overall protection and utilization for all test suites.

asymmetric co-runs, we use equal partition for PIPP as well. Our primary goal is equal resource allocation so it is intuitive to allocate equal partition and ration for PIPP and rationing policies respectively. As we will see later, the correct "initial" partition is not critical for the success of rationing whereas for PIPP it is quite crucial to allocate the right partition to begin with.

There are many, often more elaborate techniques. We pick these three policies to compare with our rationing policy because they are simple and representative. On average, PIPP has been shown to perform better than other proposals (*e.g.*, UCP and TADIP (Xie and Loh, 2009)). Also the communist and capitalist suit our purposes in that they are the extreme cases of protection and utilization. No policy can offer more absolute protection than communist, and capitalist policy uses global LRU and is one of the best and robust caching policies. If rationing can have similar protection as partitioning and utilization as sharing, then there is little room for further improvement. Furthermore, past techniques target throughput and fairness, which is different from protection and utilization. It is difficult to determine which past work is most competitive for our combined purpose. Instead, we compare with the best possible method for each single purpose, as represented by the combined strength of these two extreme policies along with the PIPP technique.

4.2 Chip Space Overhead

Let p be the number of cores sharing the cache, s be the number of sets in the cache, w be the number of ways in each set, and b be the cache block size in bytes. The total number of bits required for the data array can be computed from the following expression:

$$bits(data\_array) = s * w * b * 8 \tag{1}$$

Assuming a 40-bit physical address, the tag storage in a physically indexed cache would be

$$bits(tag\_array) = s * w * (40 - log_2 s - log_2 b + 2) \tag{2}$$

2-bit (last factor in expression (2)) storage is required to maintain the *valid* and *dirty* bits for each block assuming a writeback cache.

To implement our proposed cache rationing technique on top of the baseline cache architecture, we require 1-bit of storage for the access-bit per cache block and p ration counters per set. Each counter needs to count only up to the associativity of the cache. The total storage overhead of the rationed cache is

$$bits(ration\_array) = s * (w + p * log_2 w) \tag{3}$$

After computing the storage overhead (from expression (3)) for various configurations of caches and numbers of cores, three key observations about the storage overhead of our cache rationing policy are as following:

- Storage overhead is *almost constant* for any cache size if the associativity and number of cores sharing the cache are fixed. For 2 cores sharing an 8-way associative cache of any size, the storage overhead is 0.26% of the total cache storage.
- Storage overhead *increases slightly with the number of cores* for a given cache. For a 16-way, 1 MB cache, storage overhead for 2 cores is 0.23% and increases slightly to 0.88% for 16 cores.
- Storage overhead *decreases slightly with the associativity* for a fixed number of cores and fixed size cache. 1 MB cache shared among 2 cores has an overhead of 0.28% when it is 2-way associative. The same cache with 16-way associativity has 0.23% overhead.

The last property of rationing overhead is particularly attractive because last level shared caches tend to be highly associative. *A highly associative cache amortizes the storage overhead of rationing.*

4.3 Measure of Protection and Utilization

We identify a set of performance markers to quantify the level of resource protection and utilization, the two goals defined in Section 2.1. We will use them later to evaluate rationing and other cache-sharing techniques.

The following defines the solo-run performance. In this paper, performance is quantified by instructions per cycle (IPC).

- *Baseline:* The baseline is the performance when running solo on a single-core with the size of cache equal to the ration. It is the performance without sharing and the desired lower bound with sharing.
- *Maximum gain:* Maximum gain is the performance of an application running solo using 100% of the shared cache.

Next we define the performance in a co-run group.

- *Unhealthy co-run:* A co-run group is unhealthy if one or more of its members loses performance by a threshold. In this sense, we consider slowdown to be "damage". In the evaluation, we set the damage threshold to be 1%. The *damage* of an unhealthy co-run is the worst slowdown among its group members.

For co-run groups that are *not* unhealthy, we further divide them into two groups.

- *Healthy co-run:* A co-run group is healthy if it is not unhealthy and if one or more members see a significant benefit from cache sharing. We set the benefit threshold to be 1% over the baseline. The *benefit* of a healthy co-run is the highest gain by a group member.
- *Neutral co-run:* If a co-run is neither healthy nor unhealthy, it is neutral, meaning that running in a group does not have a significant impact one way or the other.

Finally we have the metrics for protection and utilization. Given a test suite, which is a set of co-run groups, we evaluate a cache-sharing technique as follows:

- *Level of protection:* A low number of unhealthy co-runs and low average damage inflicted in these co-runs indicates a high level of protection.
- *Level of utilization:* Indicated by the number of healthy co-runs and the average benefit enjoyed in these co-runs.

4.4 Overall Performance and Analysis

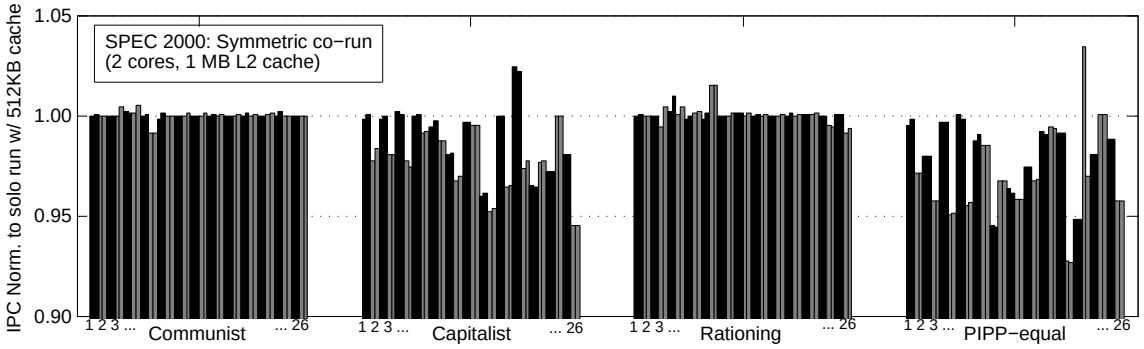
We first summarize the overall protection and utilization results in Table 3 before analyzing individual performance gains and slowdowns.

Symmetric co-run suites Figure 6 shows the performance for the four symmetrical co-run suites. As shown (and numbered) in Table 2, the left-hand side applications are integer applications, and the right-hand size are floating point applications.

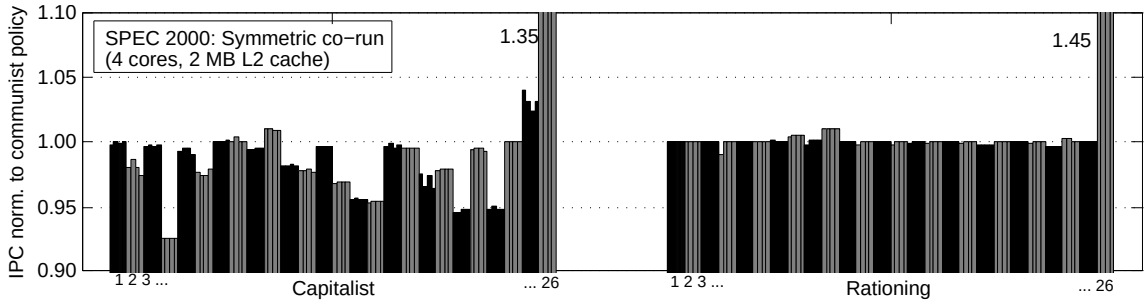
Exactly same access patterns in identical programs tends to cause contention for the same part of the cache. We first discuss these results and then elaborate with the changes caused by adding per-program address offsets (to avoid conflicts).

Due to the strong interference, the capitalist policy sees 15 damaged pairs out of 26 for 2-core co-run system, and 13 damaged quartets out of 26 for 4-core co-run system, shown individually in Figure 6 and summarized in Table 3. Similarly, PIPP also damages 16 pairs out of 26 for 2-core system.

Rationing protects most pairs from damage. It has 1 damaged pairs out of 26, and 1 damaged quartets out of 26. The protection of communist is similar, which has 1 each in 2-core and 4-core system. The reason for damage in the communist cache is the number of memory ports. In both the (solo) baseline and the partitioned co-run, the number of memory ports is the same, which is 2. In a few cases, the contention causes slight damage.



(a) SPEC 2000: 2-core symmetric co-run sharing 1MB L2 cache.



(b) SPEC 2000: 4-core symmetric co-run sharing 2MB L2 cache.

Figure 6: Symmetric co-runs. Co-run applications are clustered by color. Capitalist sharing and PIPP-equal see frequent damages in pair runs. Rationing performs as well as communist in resource protection but has a slightly better utilization.

The protection by rationing is almost as absolute as by partitioning. The average damage by rationing is similar, 1.01% vs 1.03%, and 1.14% vs 1.07%. In comparison, the average damages by capitalist sharing for the two test suites are 3.34%, 4.52%, respectively.

For lack of space, Figure 6(b) shows only the individual results for the capitalist and rationed caches. Here the performance is normalized to the communist cache performance. Here we see 3, and 2 cases of benefits, including a large gain. In 4-program co-runs, sharing and rationing improve the pair of *apsi* sharply by 1.35x and 1.45x. This is because its working set is around 550 KB, close to a quarter of the cache size and extremely sensitive to any slight increase in space (from sharing of unused ration).

Symmetric programs have the same access pattern. The V-way cache (Qureshi et al., 2005) is a hardware solution to relieve the contention (in a solo run) by assigning data blocks away from the contended area. Since our problem is specific, we use a simple solution. The idea is to add a clone-specific offset to the set calculation. We use a prime number series and call them `offsets`, one for each of the p cores. The new calculation is:

```
set = (set + offset[pid]) % num_sets
```

After the above modification, we see significant improvements in three application pairs for rationed cache. *apsi* improves by 80%, *perlbmk* improves by 4.3% and *ammp* improves

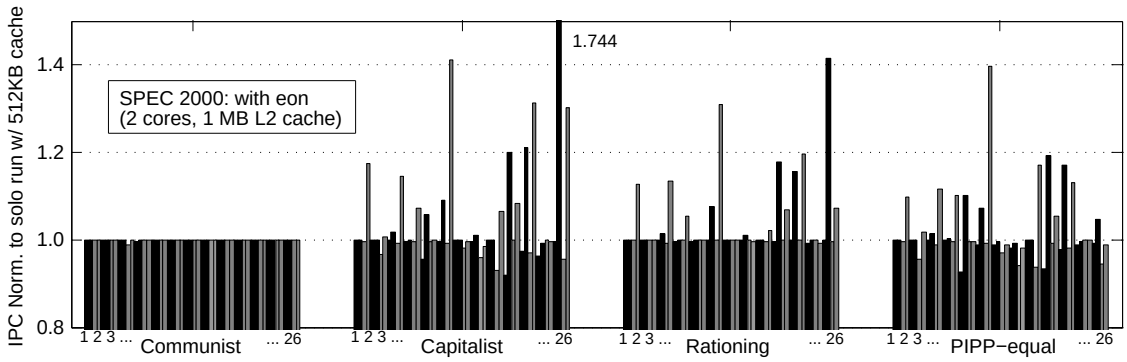


Figure 7: Co-run with *eon* which has a small working set. The first bar in each pair represents *eon*. To maximize utilization, capitalist sharing can lead to interference even for its very small working set. Rationing utilizes the unused portion but protects the remaining portion that is being used.

by 2.8%. However *art* and *lucas* slightly degrade (1%). The capitalist policy sees a similar improvement for *apsi*, 3% for *perlbmk* and a slight slowdown (-0.7%) for *ammp*.

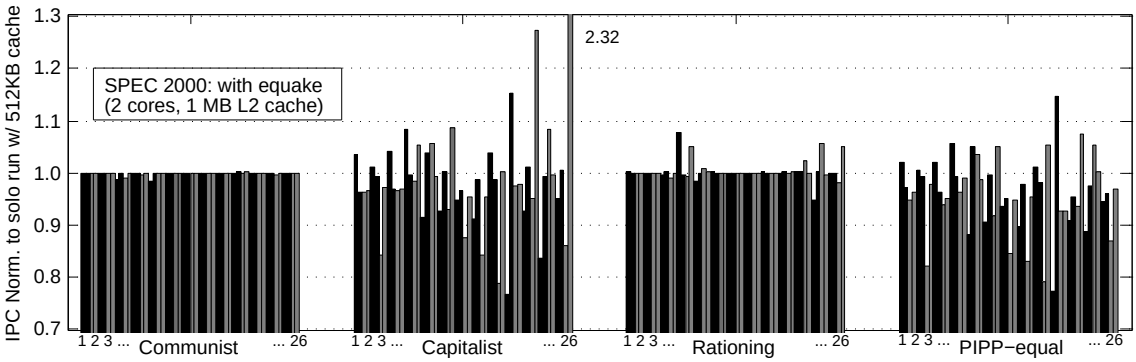
Co-run with eon *eon*'s working set is smaller than 128KB. Co-run programs can benefit from unused cache if the cache is rationed or shared. Figure 7 shows the performance for the *eon* pair-run test suite. On average, the capitalist policy speeds up the peer by 9.38% but slows down *eon* by 5.65%. In comparison, rationing does not slow down *eon* yet it achieves 14.46% speedup for other applications.

The damage to *eon* shows that sharing can lead to interference even for programs with very small working sets. By removing the damage, rationing shows the benefit of utilizing the unused portion of a ration while protecting the remaining portion that is being used.

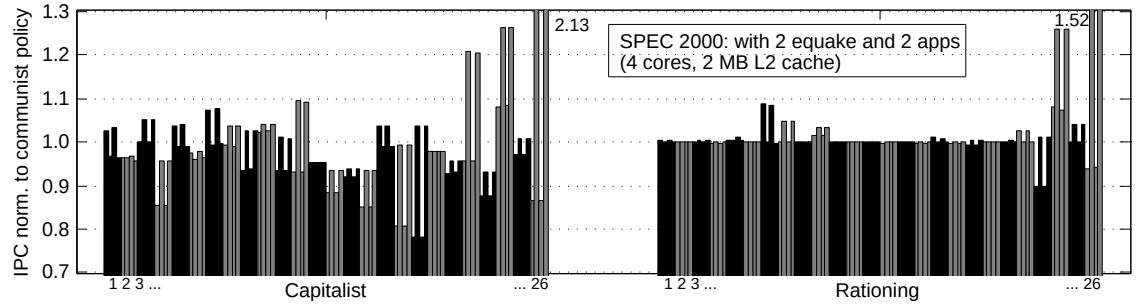
Co-run with equake *equake* has significant streaming accesses and causes significant interference in shared cache. The capitalist policy slows down the victim and speeds up the aggressor. Rationing protects the victim yet gives more space to the aggressor when it does not cause damage.

In this experiment, shown in Figure 8, we run two copies of equake with two copies of another application. Rationing loses protection for a few quartets but shows better utilization than partitioning. Compared to sharing, rationing improves fewer applications but with far fewer damages.

Sharing with highly disruptive peer (mcf) When an application shares cache with a highly disruptive peer such as *mcf*, the best policy is to confine the disruptive application using hard protection. From Figure 9 it seems that rationing improves over capitalist but still is not as good as hard partitioning. The inability to fully protect happens only for *mcf* among the total of 46 programs tested.



(a) SPEC 2000 with equake: 2-core co-run with equake sharing 1MB L2 cache.



(b) SPEC 2000 with equake: 4-core co-run with equake sharing 2MB L2 cache.

Figure 8: (a). Pair-run with *equake*. The first bar in each pair is *equake*. (b). 2 identical applications co-run with 2 equake programs. The first and third bars are *equake*, whereas the second and forth bars represent two clones of a SPEC 2000 program.

Resetting interval sensitivity We tested a range of resetting intervals for the access bit. If it is too frequent, sharing dominates protection. If it is reset at every instruction, rationing becomes capitalist. If it is too infrequent, protection dominates sharing. If we never reset, rationing becomes a type of partitioning.

Figure 10 shows reset intervals from 1 instruction to infinite (never reset). For each case, the performance is measured by the number of healthy and unhealthy pairs (out of 26). The change in these numbers confirms the reasoning above. The general trend shows that keeping the reset interval around 50K to 200K instructions is a good choice.

Cache Rationing in SPEC 2006 SPEC 2006 applications have significant larger footprints and working set sizes compared to SPEC 2000 applications. We simulated around 20 applications (rest we could not compile for Alpha) and ran 20 symmetric co-run. For other combinations the results and findings are very similar to SPEC 2000, so we only present the highly contentious symmetric co-runs. Three policies for the SPEC 2006 applications are compared in Figure 11.

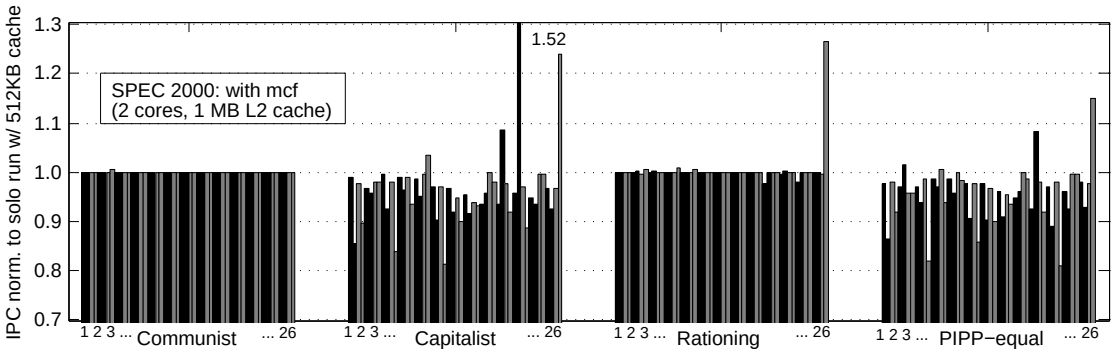


Figure 9: Co-run with extremely high cache pressure (mcf). Rationing improves over capitalist but still not as good as hard partitioning.

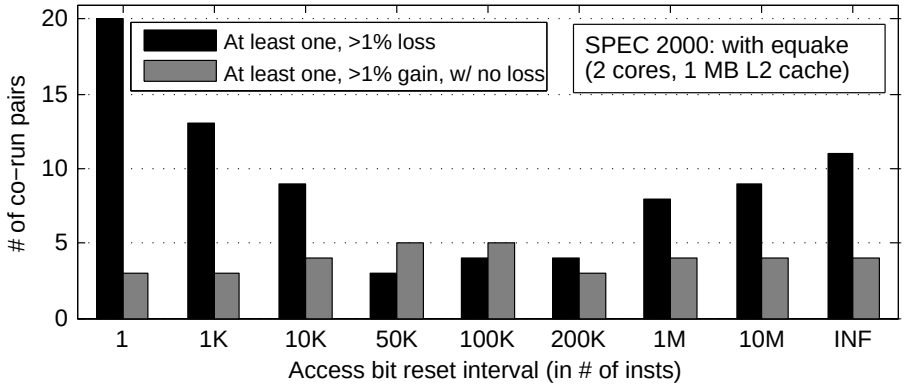


Figure 10: Sensitivity of *equake* pair-run tests to the reset interval of the access bit.

Space-time measurement Space-time working set has been used to accurately measure memory utilization (Denning and Slutz, 1978). Here we use it to measure the cache utilization. A co-run peer that takes a larger share of cache has a greater space-time product. The space-time product, when divided by the execution time, gives the average cache occupancy.

Figure 12 shows the relative space-time footprint of applications when they share the cache with *equake*. Most of the applications that see a dramatic speedup tend to have a much larger space-time. On the other hand, applications with smaller space-time footprint suffer because the cache space is hogged by the aggressive application. Rationing does not allow the aggressive applications to increase their space-time beyond a certain limit and achieves space-time footprint very similar to the communist policy. In contrast in the capitalist policy, the space-time is not equally distributed.

4.5 Additional Optimizations

Cache rationing supports other optimizations with little additional costs. We briefly discuss some of them in this section.

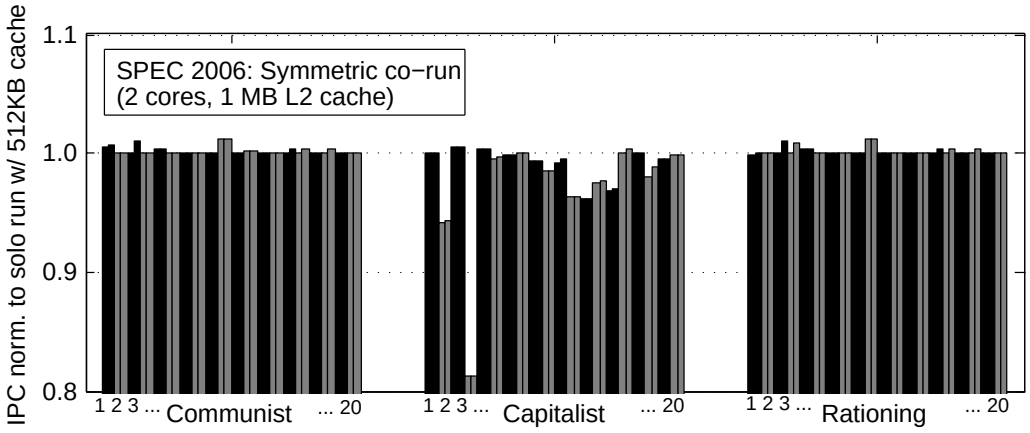


Figure 11: Symmetric co-run of SPEC 2006 applications.

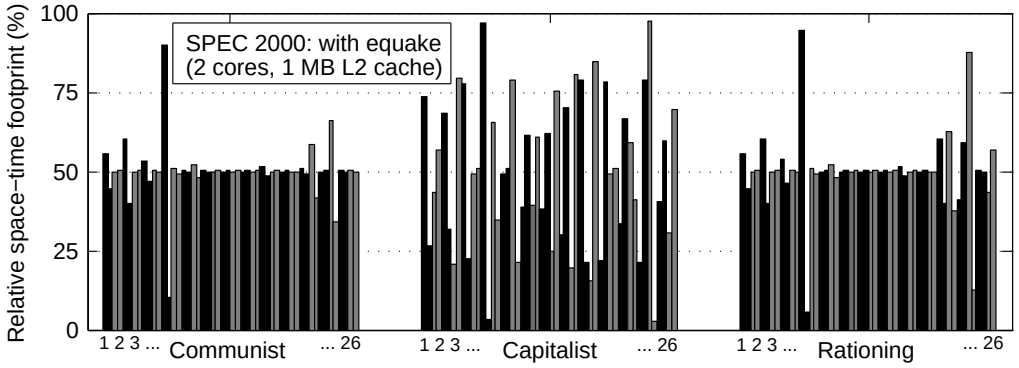


Figure 12: Space-time footprints in various cache management policies.

Coarse-grain rationing So far we have considered rationing with a ration counter maintained for each core and each cache set. Per-set rationing enables fine-grained protection but incurs a higher storage overhead. A simpler and more efficient alternative is to maintain a single ration counter for each core. The counter consolidates all the usage of a particular core. The protection is coarse grained. It can victimize a less aggressive application in the sets in which its data occupies all cache ways. The overall storage overhead now is 0.019% (1 access-bit per 64 bytes and per-core 14-bit ration counters for 1MB L2 cache).

Figure 13 shows the performance of two rationing techniques for symmetric co-run. For the combination of applications the results are similar and for brevity we are showing only symmetric co-runs. As we can see the impact of relaxing the per-set precision is not that bad and we are still able to achieve resource protection in nearly all cases.

Hardware-software Collaborative Caching We make a hypothetical study and assume that software has complete knowledge about its locality, in particular, the forward reuse distance distribution for every memory instruction. If a memory reference tends to make accesses whose reuse distance is greater than the cache size, we mark it with the

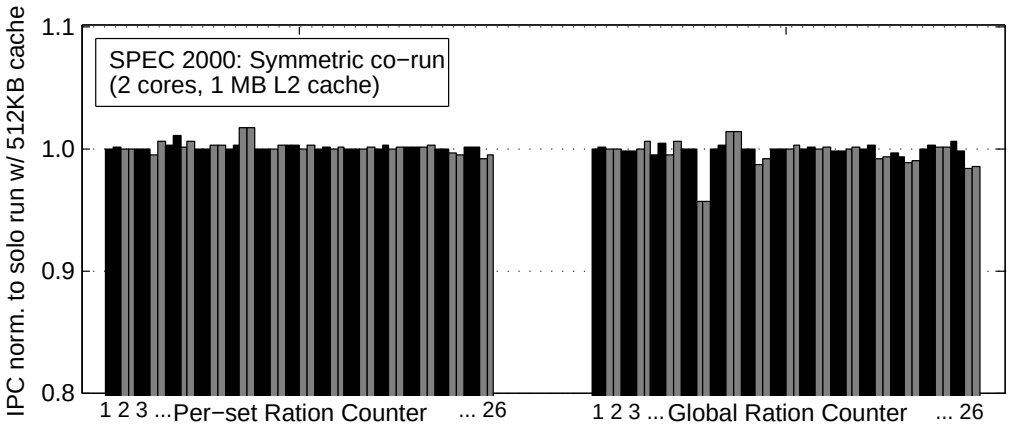


Figure 13: Comparison of fine-grain per-set rationing and coarse-grain global rationing.

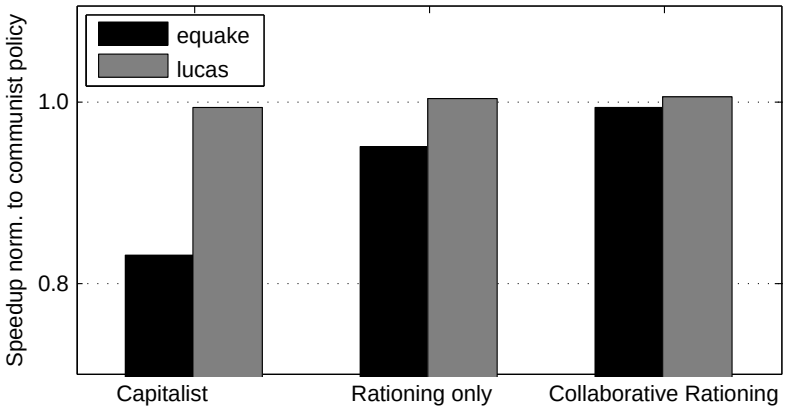


Figure 14: *equake* and *lucas* co-run. Collaborative rationing obtains the full protection while the hardware-only rationing does not.

evict-me bit, so the rationed cache would clear the access bit for the accessed data and cause them to be victimized.

In preliminary testing, we found that a program pair benefits significantly from the software hint. Figure 14 compares the *equake-lucas* co-run performance under capitalist, rationing, and collaborative rationing (with software hints). *lucas* has a large working set of around 2MB. Much of it has zero reuse in the 1MB cache. This means that even if *lucas* brings the data into the cache we should place it at the bottom of the LRU stack. When *lucas* is paired with *equake*, because of the aggressive nature of *lucas*, *equake* suffers badly under the capitalist policy. *lucas* speeds up by 16% whereas *equake* slows down by 23%. Rationing reduces the slow down of *equake* to around 10%. Collaborative rationing further improves by marking some *lucas* references with the software hint. Then the slowdown of *equake* becomes negligible (0.4%) and *lucas* still improves by 5%.

5 Related Work

Cache Partitioning There have been a plethora of work to efficiently partition the cache among multiple threads and programs (Qureshi and Patt, 2006; Moreto et al., 2008; Suo et al., 2008; Lin and Balasubramonian, 2011; Suh et al., 2004; Chang and Sohi, 2007; Kim et al., 2004; Stone et al., 1992; Sanchez and Kozyrakis, 2011; Kandemir et al., 2011). There are two ways to partition the cache. The first is to divide the ways in each cache set (Balasubramonian et al., 2000). The second is cache data placement through OS supported page coloring (Lin et al., 2008). Page coloring requires software support, while set-based partitioning may be controlled in either software or hardware. In both cases, fixed cache partitioning cannot guarantee full utilization, since a program may not fully utilize its partition.

The problem of fixed partitioning can be ameliorated through dynamic partitioning. One solution is to label each task with a quota, and the OS partitions the cache space based on the demand of all running programs (Rafique et al., 2006). If there is only one program running, it will have the entire cache regardless of the actual quota it declares. When a task comes or leaves, or its quota is changed, the OS will dynamically adjust the cache partitions.

Even in dynamic partitioning, the actual demand of a task may not match the space it is assigned to. A program can be given a large space (by declaring a large quota) but use only a fraction. Incomplete utilization would still ensue.

Cache ration is a type of dynamic allocation like quota partitioning. Unlike fixed partitioning, the ration is "soft" in that the unused ration of one program could be given to other programs. The support happens in hardware. It is more responsive and may change at each cache access. In comparison, quota is managed by the OS and does not change without OS intervention.

Cache Management Many techniques have been proposed to improve caching effectiveness by choosing victims more carefully, for example by detecting streaming accesses. In particular, many studies focus on shared caches under multi-programming workloads. For instance, restricting memory-intensive threads (Liu and Yeung, 2009) or giving more space to programs that can best use the additional space to reduce misses (Qureshi and Patt, 2006). Other factors such as memory-level parallelism can also be factored in so that the heuristics can directly target performance gain (Moreto et al., 2008; Suo et al., 2008; Lin and Balasubramonian, 2011). Such management can also take on a temporal dimension (Suh et al., 2004; Chang and Sohi, 2007) or trying to improve fairness (Kim et al., 2004). Adaptive insertion policies have also been studied to share the cache more effectively among multiple threads/applications (Qureshi et al., 2007; Jaleel et al., 2008). Few proposals try to manage cache by set-pinning (Srikantaiah et al., 2008). In the contexts of CMP there has been few proposals (Du et al., 2010; Sundararajan et al., 2012; Sharifi et al., 2012; Duong et al., 2012).

There has been effective techniques for improving QoS in CMP environments with large last level cache (Iyer, 2004; Guo et al., 2007; Iyer et al., 2007).

The aforementioned techniques are adaptive but all use heuristics. Because of the nature of heuristics, *e.g.*, predicting low-locality data, they may be wrong and hence counter productive in some cases. The occasional loss is permitted as long as the overall throughput or eventual fairness is improving. For throughput, performance loss in minority tasks is permitted if it is compensated by a higher gain by others. For fairness, a task can lose performance temporarily even frequently if the loss is canceled by sufficient gains in other periods of execution. These solutions emphasize dynamic feedback and control.

Cache rationing has a different goal, which is to ensure resource protection in all cases at all times. It does not try to discern between high- and low-locality data. Instead, its hardware support is to protect against interference by all peer cache accesses. Rationing is dynamic in the detection of an unused cache block. For this purpose, it adds an access bit and revises the replacement logic based on the access bit. The goal is not overall throughput but the so-called Pareto optimality where we cannot make one program better off without making another program worse off.

Collaborative Caching Previous hints included placement, bypassing, or evict-me and were designed for solo-use or capitalist shared cache (Beyls and D'Hollander, 2005; Wang et al., 2002; Sinharoy et al., 2005). They may be used in partitioned cache, partitioned either by cache or page coloring. However, the possibility has not been studied or evaluated. Moreover, partitioning and collaboration use separate mechanisms. In rationed cache, we show an integrated design — the access bit provides both the rationing mechanism and the hint mechanism. We believe that this is the first design to combine the two mechanisms. The integration expands the scope of software-hardware collaboration to include not just ration utilization of individual programs but also better ration protection between programs, as shown in the evaluation section by the *equake-lucas* pair run (Figure 14).

6 Summary

This paper has presented rationing for shared-cache management. The new cache hardware protects the cache ration of each program and at the same time finds unused ration to share among the co-run programs. A core is allowed to use another core's ration *only if/when* the ration is not being used. The paper shows that the new support can be added on top of existing cache architecture with minimal additional hardware and scales well with the cache size, number of cores and the cache associativity. When an application does not use all its ration, rationing achieves good utilization similar to free-for-all shared cache. When a program exerts strong interference, rationing provides good protection similar to partitioned cache. In addition, rationing provides an integrated design for cache sharing and software-hardware collaboration.

References

Balasubramonian, R., D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd International Symposium on Microarchitecture*. Monterey, California.

Beyls, K. and E.H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250.

Brock, Jacob, Xiaoming Gu, Bin Bao, and Chen Ding. 2013. Pacman: Program-assisted cache management. In *Proceedings of ISMM*.

Chang, Jichuan and Gurindar S. Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of ICS*, pages 242–252.

Denning, Peter J. and Donald R. Slutz. 1978. Generalized working sets for segment reference strings. *Communications of ACM*, 21(9):750–759.

Du, Jianjun, Yixing Zhang, Zhongfu Wu, and Xinwen Wang. 2010. Management policies analysis for multi-core shared caches. ADMA'10, pages 514–521.

Duong, Nam, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving cache management policies using dynamic reuse distances. In *Proceedings of MICRO*, pages 389–400.

Flautner, Krisztian, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. 2002. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of ISCA*, pages 148–157.

Gu, Xiaoming, Tongxin Bai, Yaoqing Gao, Chengliang Zhang, Roch Archambault, and Chen Ding. 2008. P-OPT: Program-directed optimal cache management. In *Proceedings of the LCPC Workshop*, pages 217–231.

Guo, Fei, Hari Kannan, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. 2007. From chaos to qos: case studies in cmp resource management. *SIGARCH Comput. Archit. News*, 35(1):21–30.

Hsu, Lisa R., Steven K. Reinhardt, Ravishankar R. Iyer, and Srihari Makineni. 2006. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT*, pages 13–22.

Iyer, Ravi, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. 2007. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 25–36.

Iyer, R.R. 2004. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of International Conference on Supercompupting*, pages 257–266.

Jaleel, Aamer, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr., and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In *Proceedings of PACT*, pages 208–219.

Kandemir, Mahmut, Taylan Yemliha, and Emre Kultursay. 2011. A helper thread based dynamic cache partitioning scheme for multithreaded applications. pages 954–959.

Kaxiras, Stefanos, Zhigang Hu, and Margaret Martonosi. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of ISCA*, pages 240–251.

Khan, Samira Manabi, Yingying Tian, and Daniel A. Jimenez. 2010. Sampling dead block prediction for last-level caches. In *Proceedings of MICRO*, pages 175–186.

Kim, Seongbeom, Dhruba Chandra, and Yan Solihin. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of PACT*, pages 111–122.

Lin, Jiang, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of HPCA*, pages 367–378.

Lin, Xing and Rajeev Balasubramonian. 2011. Refining the utility metric for utility-based cache partitioning. In *Workshop on Duplicating, Deconstructing, and Debunking*.

Liu, Wanli and Donald Yeung. 2009. Using aggressor thread information to improve shared cache management for CMPs. In *Proceedings of PACT*, pages 372–383.

Moreto, Miquel, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. 2008. MLP-aware dynamic cache partitioning. In *Proceedings of HiPEAC*, pages 337–352. Springer-Verlag, Berlin, Heidelberg.

Qureshi, Moinuddin K., Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel S. Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of ISCA*, pages 381–391.

Qureshi, Moinuddin K. and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of MICRO*, pages 423–432.

Qureshi, Moinuddin K., David Thompson, and Yale N. Patt. 2005. The v-way cache: Demand based associativity via global replacement. In *Proceedings of ISCA*, pages 544–555.

Rafique, N., W. Lim, and M. Thottethodi. 2006. Architectural support for operating system-driven cmp cache management. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

Rus, Silvius, Raksit Ashok, and David Xinliang Li. 2011. Automated locality optimization based on the reuse distance of string operations. In *Proceedings of CGO*, pages 181–190.

Sanchez, Daniel and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of ISCA*, pages 57–68.

Sharifi, Akbar, Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. 2012. Courteous cache sharing: Being nice to others in capacity management. In *Proceedings of the 49th Annual Design Automation Conference*, pages 678–687.

Sherwood, T., E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of ASPLOS*. San Jose, CA.

Sinharoy, B., R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. 2005. Power5 system microarchitecture. *IBM J. Res. Dev.*, 49:505–521.

Srikantaiah, Shekhar, Mahmut Kandemir, and Mary Jane Irwin. 2008. Adaptive set pinning: Managing shared caches in chip multiprocessors. In *Proceedings of ASPLOS*, pages 135–144.

Stone, Harold S., John Turek, and Joel L. Wolf. 1992. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068.

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

Sundararajan, Karthik T., Timothy M. Jones, and Nigel P. Topham. 2012. Energy-efficient cache partitioning for future cmps. In *Proceedings of PACT*, pages 465–466.

Suo, Guang, Xuejun Yang, Guanghui Liu, Junjie Wu, Kun Zeng, Baida Zhang, and Yisong Lin. 2008. Ipc-based cache partitioning: An ipc-oriented dynamic shared cache partitioning mechanism. In *Proceedings of ICHIT*, pages 399–406.

Wang, Z., K. S. McKinley, A. L.Rosenberg, and C. C. Weems. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of PACT*. Charlottesville, Virginia.

Xie, Yuejian and Gabriel H. Loh. 2009. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of ISCA*, pages 174–183.

Yang, Xi, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. 2011. Why nothing matters: the impact of zeroing. In *Proceedings of OOPSLA*, pages 307–324.