Replacement policies for shared caches on symmetric multicores : a programmer-centric point of view

Pierre Michaud

▶ **To cite this version:**

Pierre Michaud. Replacement policies for shared caches on symmetric multicores : a programmer-centric point of view. [Research Report] PI 1908, 2008, pp.25. inria-00340545

HAL Id: inria-00340545
https://inria.hal.science/inria-00340545

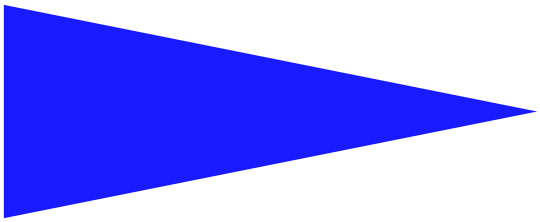
Submitted on 21 Nov 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1908



REPLACEMENT POLICIES FOR SHARED CACHES ON SYMMETRIC MULTICORES : A PROGRAMMER-CENTRIC POINT OF VIEW

PIERRE MICHAUD

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Replacement policies for shared caches on symmetric multicores : a programmer-centric point of view

Pierre Michaud

Systèmes communicants
Projets CAPS

Publication interne n ˚ 1908 — Novembre 2008 — 23 pages

Abstract: The presence of shared caches in current multicore processors may generate a lot of performance variability when several applications execute simultaneously. For the programmer of an application with quality-of-service goals, this performance variability may lead to a very pessimistic tuning. To solve this problem, there must be a way for the programmer to define a reasonable performance target and make sure that the actual performance is greater than or close to the target. We propose that the performance target be defined as the performance measured when each core runs a copy of the application, which we call self-performance. This study characterizes self-performance and explains how the shared-cache replacement policy can be modified for self-performance to be meaningful.

Key-words: Symmetric multicore processor, quality of service, self-performance, shared cache, replacement policy, memory bandwidth

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Politiques de remplacement sur les caches partagés des multi-coeurs symétriques

Résumé : La présence de caches partagés dans les processeurs multi-coeurs est une source importante de variabilité de performance lorsque plusieurs applications s'exécutent simultanément. Pour le programmeur d'une application avec des objectifs de qualité de service, cette variabilité de performance peut conduire à un dimensionnement très pessimiste de l'application. Afin de résoudre ce problème, on doit donner au programmeur la possibilité de définir un objectif raisonnable en performance, et on doit faire en sorte que la performance réelle soit supérieure ou proche de cet objectif. Nous proposons que l'objectif en performance soit défini comme la performance mesurée lorsque chaque coeur exécute une copie de l'application. Nous appelons cette mesure l'auto-performance. Cette étude caractérise l'auto-performance et explique comment la politique de remplacement des caches partagés peut être modifiée pour que l'auto-performance soit un objectif atteignable.

Mots clés : Processeur multi-coeur symétrique, qualité de service, auto-performance, cache partagé, politique de remplacement, bande passante mémoire

1 Introduction

There exists an implicit performance contract between the processor and the programmer. When the programmer writes a program and measures its performance by running it, he assumes that the performance is approximately deterministic, hence reproducible. There can be some performance variations, some due to the operating system (e.g., different physical page allocation that changes cache conflicts), some due to the microarchitecture (e.g., different initial branch predictor states). But, before the multicore era, these variations were generally small.

With multicore processors able to execute several applications simultaneously, performance variations can have a much larger magnitude. This is mainly due to shared microarchitectural resources, especially shared caches. Depending on workload characteristics, the actual performance of a particular application may be much smaller than the performance measured by the programmer. For applications with quality-of-service goals, this leads to very pessimistic tuning.

Previously proposed solutions to this problem involve the use of programmable priorities or quotas [5, 3, 10, 1, 8, 2, 4]. With these solutions, programmers who want a performance guarantee must ask for resources they are sure to obtain. In practice, this requires either to partition shared resources evenly between cores or to keep some cores unused.

We propose a new solution, which is to have an explicit contract between the microarchitecture and the programmer. The programmer measures the application performance by running simultaneously a copy of the application on each core. This defines what we call *self-performance*. This study characterizes self-performance and shows that, for self-performance to be meaningful, the microarchitecture must manage shared resources carefully. In particular, we show that conventional cache replacement policies are not compatible with the self-performance contract. We propose some replacement policies that are compatible with self-performance. One of our replacement policies, called B2, is simpler to implement in hardware than previously proposed quota-based solutions.

The paper is organized as follows. Section 2 explains the concept of self-performance and the motivations behind it. We show in Section 3 that conventional cache replacement policies are not compatible with self-performance and we provide insights as to why this is so. We also show that, even without considering the self-performance contract, conventional cache replacement policies lead to the paradoxical situation that increasing the memory bandwidth may decrease the performance of some applications. In Section 4, we propose sharing-aware replacement policies that solve the problems emphasized in the previous section. Section 5 discusses some implications of our proposition for throughput and for multi-threaded programs. Finally, Section 6 concludes this work.

Simulations. The simulation results presented in this study correspond to a multicore with 4 identical cores, depicted in Figure 1. The 4 cores share a 4 MB 16-way set-associative level-2 (L2) cache. The main characteristics of the simulated microarchitecture are summarized in Table 1. More details about the simulator and about benchmarks are provided in Appendix A. Unless stated otherwise, each simulated IPC (instructions retired per cycle) reported in this study corresponds to the IPC of the thread running on core #1 for 10 million CPU cycles while other threads run on cores #2,#3 and #4.

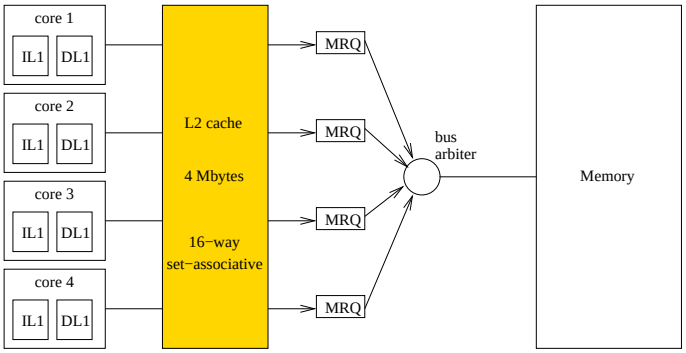


Figure 1: Symmetric multicore simulated in this study.

multicore	4 dynamically-scheduled cores
core fetch	2 instructions per cycle (x86)
core retire	2 instructions per cycle (x86)
reorder buffer	64 instructions (x86)
branch predictor	YAGS, 12 Kbytes, 25-bit global history, 8-bit tags
branches	10-cycle misprediction penalty (minimum), solved at retirement
IL1 cache	private, 32 Kbytes, 4-way set-associative, 64-byte blocks, LRU latency 1 cycle, 1 block refill & 2 instructions read per cycle
DL1 cache	private, 32 Kbytes, 4-way set associative, 64-byte blocks, LRU, write-back write-allocate, latency 2 cycles, 1 block refill & 1 load/store per cycle
L2 cache	shared, 4 Mbytes, 16-way set associative, 64-byte blocks, LRU, write-back write-allocate, latency 15 cycles, bandwidth 1 block/cycle (refill or block read or block update)
MRQ	20 pending L2 misses
memory bus	8 bytes per CPU cycle
memory latency	300 CPU cycles
hardware prefetch	disabled

Table 1: Simulated microarchitecture : default configuration

2 Self-performance

In this study, we consider independent sequential tasks. Currently, most programs executing on existing multicores are sequential programs. Though it is hoped that more and more parallel applications will be developed, sequential programming is still very important. We explain in Section 5.3 what are the implications of our proposition for multi-threaded programs.

2.1 The problem

For applications with quality-of-service (QoS) goals, it is important that the performance measured at programming time be deterministic, or appears to be so. In a multicore, several resources are shared : physical memory, caches, buses, power supply, etc. Because of resource sharing, when several independent applications run concurrently on different cores, the performance of each application depends on the characteristics of the other applications. On a single CPU, the operating system (OS) can control the amount of physical memory and CPU time allotted to each task, in particular tasks with QoS goals. On a multicore, the OS can decide which applications to run simultaneously and for how long, but it has no control on microarchitectural resource sharing. The notion of CPU time is not accurate, as the quantity of work done during a fixed period of time may vary drastically depending on resource sharing. What we need is a way for the programmer to specify a performance target and a microarchitecture that minimizes the possibility for the actual performance to fall significantly below the performance target. An obvious solution would be to assume that the application runs alone on the multicore. But the multicore would be underused.

The solution that has been proposed so far is to let the OS have a fine control of shared microarchitectural resources [5, 3, 10, 1, 8, 2, 4]. Each shared resource is associated with priorities or quotas that are programmable. For example, the programmer defines his microarchitectural needs, i.e., the resources he wants (cache size, bus bandwidth, etc.), and the OS tries to give to each application the resources it asks for. However, this raises a question : what if the sum of resources asked by applications running concurrently exceeds the processor's resources ? With programmable quotas, each application is given a share of resources that is a function of but is not necessarily equal to what the application asks for [10]. This implies that the applications for which it is important to obtain a performance guarantee must ask for quotas that they are sure to obtain. In practice, this means that when a resource is shared by up to N threads, the programmer must ask for $1/N$ (or less) of the resource in order to obtain a performance guarantee.

Based on this observation, we propose a viable alternative to programmable quotas [1]. We call it *self-performance*. Self-performance is less flexible than programmable quotas but is simpler to implement.

2.2 Self-performance

Obtaining a performance guarantee is a two-stage problem :

- We need a way to define a performance target.

[1] To our knowledge, programmable quotas have not been adopted by the industry yet.

- We must minimize the possibility for the actual performance to fall below the targeted performance.

On the one hand, we do not want the performance target to be too pessimistic. On the other hand, the performance target must be a value that is possible to enforce. If it is too optimistic, it may be impossible to reach the performance targets of all the applications running simultaneously. If we measure the application performance when it runs simultaneously with some other random applications, we may obtain a performance target that is too optimistic. If we choose misbehaving applications to stress shared resources, we may obtain a performance target that is too pessimistic. Instead, we propose to define the performance target of an application by running copies of this application on all cores. More precisely, we define the *self-performance* contract as follows :

> *The self-performance of a sequential program on a symmetric multicore processor is the performance measured for one instance of the application on a* **symmetric run***, i.e., when running simultaneously and synchronously copies of that program on all cores, using the same inputs. The actual performance must be greater than or close to the self-performance, whatever the applications running on the other cores.*

The rationale is as follows. If the application uses few resources, its self-performance is very close to the performance when it runs alone. But if the application asks for a lot of resources, it competes with copies of itself and gets a share that is equal to the resource size divide by the number of cores. The performance target defined this way is neither too optimistic nor too pessimistic. Self-performance can be measured by the programmer without requiring any knowledge of the microarchitecture internal details (e.g., which resources are shared, how the resource arbitration works, etc.). The programmer does not even need to know the number of cores. The only thing that the programmer must be aware of is the self-performance contract. For the convenience of the programmer, the OS should provide a *selfperf* utility for launching symmetric runs. Programmers who do not need a performance guarantee can measure performance as usual, without using the *selfperf* utility. But it is an optimistic performance in this case.

System resources. In this study we focus on shared microarchitectural resources, and more particularly shared caches. We do not address the problem of system resources, like physical memory. For example, if the programmer has QoS goals and wants a high self-performance, he should prevent the application memory demand from exceeding the memory size divided by the number of cores. We assume that the OS is always able to give this amount of memory to the application.

3 Shared caches and self-performance

Unlike for system resources, the operating system has little control on shared microarchitectural resources. It is possible to have some control by carefully choosing which application to run simultaneously (provided such choice exists). But existing processors do not allow the OS to control microarchitectural resources more finely.

Among shared microarchitectural resources, caches exhibit the most chaotic and difficult-to-predict behavior. For example, on a set-associative cache with *least-recently-used* (LRU) replacement, a small decrease of the number of cache entries allotted to a thread may result in the miss ratio suddenly going from 0 to 100%. The most obvious way to avoid the erratic behaviors due to shared caches is to avoid shared caches. Nevertheless, shared caches have some advantages. On a multicore with private caches, whenever a single thread is running, the cache capacity of idle cores is generally wasted. When a cache is shared between several cores, the whole cache capacity is accessible to a single running thread. This is particularly interesting for the last on-chip cache level, as off-chip accesses are costly. There are other advantages when several threads from the same application communicate with each other. With private caches, several copies of the same data may be replicated. Not only does this decrease the effective cache capacity, but this means potentially a cache miss for each copy. For these reasons, several recent multicores have shared level-2 (L2) or level-3 (L3) caches. However, to our knowledge, there is no mechanism in these multicores to control the way the cache capacity is partitioned between different threads running concurrently. The partitioning is simply the result of the cache replacement policy, that is why we call it *natural partitioning* in this study.

3.1 Under natural cache partitioning, the self-performance can exceed the actual performance

The model of cache partitioning proposed in [12], though inaccurate in practice, is useful for understanding some qualitative aspects of natural cache partitioning. We present a simplified version of the model, which we will use to help understand our simulation results.

Let us consider n threads numbered from 1 to n running simultaneously, and a fully-associative shared cache with a capacity of C blocks. The number of cached blocks belonging to thread i is w_i. It is assumed that the cache capacity is saturated, i.e., $C = \sum_{i=1}^{n} w_i$. The miss rate of thread i, in misses per cycle, is m_i. The total miss rate is $m = \sum_{i=1}^{n} m_i$. The model assumes that, on a miss from any thread, the probability that the victim block belongs to thread i is proportional to the total number of cached blocks belonging to thread i, i.e., it is w_i/C. During T cycles, $m_i T$ blocks from thread i are inserted in the cache and $mT \times w_i/C$ blocks from thread i are evicted from the cache. It is assumed that an equilibrium is eventually reached, such that w_i is stable. It means that, for each thread, the number of block insertions equals the number of block evictions. Hence we have $m_i T = mT \times w_i/C$, that is,

$$\frac{m_i}{w_i} = \frac{m}{C} \tag{1}$$

This quantity, m_i/w_i, was not identified in [12]. We call it the *cache pressure* of thread i. Equation (1) means that the equilibrium partitioning is such that all threads have equal cache pressure. Figure 2 shows on an example how the concept of cache pressure is useful for finding the equilibrium cache partition from the threads miss rate curves (misses per cycle as a function of the number of cached blocks). On this example, the cache is shared between 2 threads. Thread 1 needs less than half the cache capacity to have a null miss rate. However, because it shares the cache with thread 2, thread 1 has a non-null miss rate. The example of Figure 2 explains why natural cache partitioning

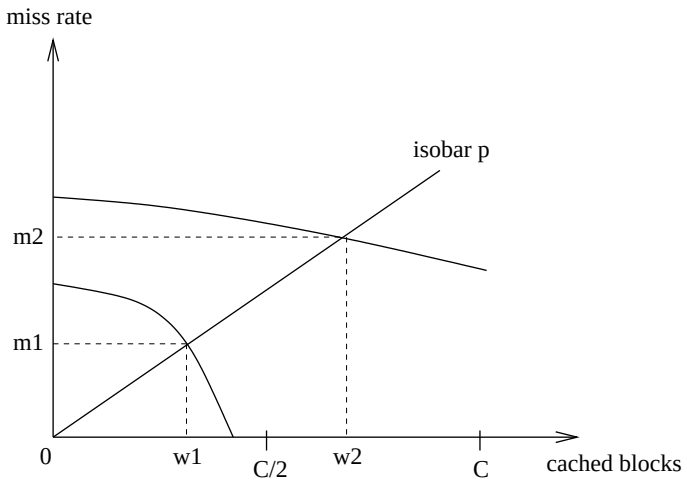


Figure 2: Example with 2 threads. The miss rate m_i of thread i (in misses per cycle) is assumed to be a function of the number w_i of cached blocks. The equilibrium partition (w_1, w_2) is such that the two threads have equal cache pressure $p = m_1/w_1 = m_2/w_2$, hence the points (w_1, m_1) and (w_2, m_2) lie on the same *isobar*, which is represented by a straight line whose slope is the pressure p. To find the equilibrium partition, rotate the isobar around the origin until $w_1 + w_2 = C$.

cannot guarantee that the actual performance will reach the self-performance target. In particular, the performance of a thread may be severely decreased when the other threads have high miss rates.

Experiment on a real multicore. We did a simple experiment on a MacBook Pro featuring an Intel Core 2 Duo processor and 2 GB of memory. This processor has 2 cores and a 4 MB shared L2 cache. We ran benchmark *vpr* from the SPEC CPU2000. The measured execution time was approximately 51 seconds. Then we measured the self-performance of *vpr* by running simultaneously two instances of *vpr*. The execution time of *vpr* was 53 seconds, which means that the self-performance of *vpr* is close to its performance when it runs alone. Then we ran *vpr* simultaneously with benchmark *mcf* from the SPEC CPU2000. The execution time of *vpr* was 73 seconds, i.e., 38% worse than the self-performance. Then we ran *vpr* simultaneously with a microbenchmark that we wrote and which we denote *999*. Microbenchmark *999* is provided in Figure 3. It has a very high miss rate (1 miss every 4 instructions) and evicts cache blocks very aggressively. The execution time of *vpr* was 101 seconds, i.e., 90% worse than the self-performance. We used the Apple tool *shark* to access the performance counters of the Core 2 Duo and we checked that the decrease of performance comes from an increase of L2 cache misses. This experiment shows that, under natural cache partitioning, the actual performance may be significantly smaller than the self-performance.

```
int a[SIZE];

main()
{
  int i,n;
  int x = 0;
  for (n=0; n<1000000; n++) {
    for (i=0; i<SIZE; i+=STEP) {
      x += a[i];
    }
  }
  printf("%d\n",x);
}
```

Figure 3: Microbenchmark *999* (compiled with *gcc -O3 -DSIZE=160000000 -DSTEP=16*)

3.2 Self-performance is not necessarily defined under natural cache partitioning

In our definition of self-performance, we made the implicit assumption that, on a symmetric run, performance is the same on all cores. With identical cores, this is indeed the case most of the time. According to the cache pressure model (cf. Figure 2), if threads have the same miss rate curve, they get the same share of the cache capacity. Therefore, a symmetric run on 4 cores should result in each thread getting one fourth of the cache capacity. However, the cache pressure model is only an approximation of reality. From our experiments and simulations, the LRU replacement policy is unlikely to generate strange performance variations on symmetric runs. But this is not necessarily true with other replacement policies. Though we present results only for LRU in this study, we also did simulations with the DIP replacement policy.

DIP was recently proposed as a substitute for LRU in L2 and L3 caches [9]. DIP is a very attractive proposition that may be implemented in future processors. All our observations and conclusions with LRU are the same with DIP, except that natural cache partitioning under the DIP policy can lead to strong performance asymmetry on symmetric runs. The DIP policy was originally proposed for private caches, but it can be adapted easily to shared caches. Instead of having a single PSEL counter for the cache, we have one PSEL counter for each core. Figure 4 shows the result of running 4 instances of microbenchmark *999* compiled with $SIZE = 2^{19}$ (cf. Figure 3) when the L2 replacement policy is DIP and the memory bandwidth is 4 bytes per cycle. The plot shows the number of retired instruction on each core as a function of time. Despite cores being identical, this example exhibits a strong performance asymmetry, the performance of core #3 being higher than the other cores. Our simulator uses a pseudo-random number generator (RNG), which is used in the DIP policy and in the bus arbitration policy. Actually, the leading core varies with the RNG seed. This phenomenon can be understood as follows. Going back to the cache pressure model, we expect threads with identical miss rate curves (in particular identical threads) to converge to a state where the shared cache is equally partitioned between threads. The reason is that there is a negative feedback : the more cached blocks belong to a given thread, the higher the probability for that thread to have its blocks evicted. Though we have no formal proof, a negative feedback seems to be at work

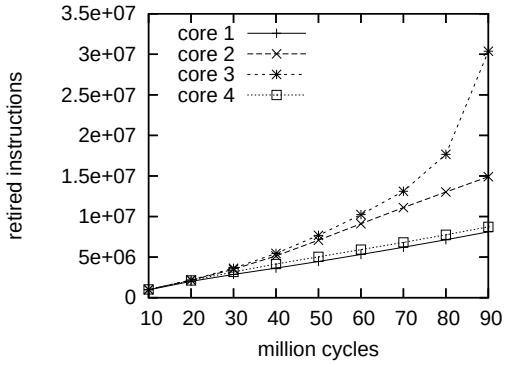


Figure 4: Symmetric run of microbenchmark *999* with $SIZE = 2^{19}$ (Figure 3). Memory bandwidth is 4 bytes/cycle and the L2 replacement policy is DIP. The plot shows the number of retired instructions on each core as a function of time.

with the LRU policy as well. Under LRU, we did not encounter a single example of a symmetric run leading to significant performance asymmetry. DIP may have a completely different behavior. When the BIP policy generates fewer misses than LRU, it is selected by the PSEL counter as the best policy. Under BIP replacement, a block inserted in the cache recently has a high probability to be the next victim. It will be the next victim if it is not re-referenced before the next cache miss (from any thread). In such case, the BIP policy has a tendency to evict blocks belonging to the thread with the highest miss rate, i.e., on a symmetric run, the thread with the smallest number of cached blocks. Hence we have a positive feedback where small divergences get amplified with time. This is a case of sensitivity to initial conditions : before the divergence occurs, we are unable to predict the future evolution. Such chaotic behavior is of course incompatible with providing a performance guarantee. The SAR policies proposed in Section 4 solve this problem.

3.3 A symmetric run is not equivalent to a static partitioning of shared resources

One of our counter-intuitive findings is that self-performance is not exactly the performance one would measure with programmable quotas by partitioning each resource statically and equally between cores. Actually, when memory bandwidth limits performance, self-performance exceeds the performance of a single run with statically partitioned resources.

Figure 5 shows the IPC (instructions retired per cycle) for a subset of our benchmarks whose performance is limited by memory bandwidth. For each benchmark we show results for 4 configurations, where SGL denotes single runs (i.e., there are 3 idle cores) and SYM denotes symmetric runs. SGL-1 is for a memory bandwidth of 1 byte per CPU cycle and a 1 MB shared cache. SYM-4 is for a bandwidth of 4 bytes/cycle and a 4 MB cache. SGL-2 is for a memory bandwidth of 2 byte

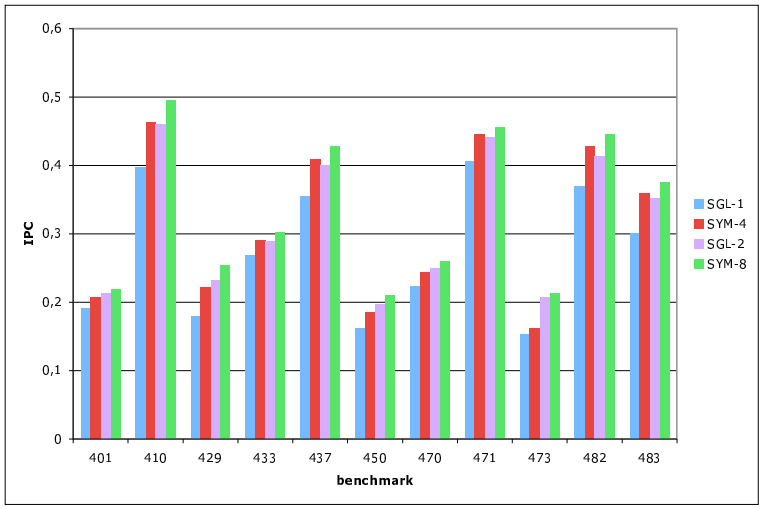


Figure 5: IPC for a subset of our benchmarks. For each benchmark, the IPC of 4 different configurations is shown. Configuration SGL-1 is for a single run (i.e., there are 3 idle cores) with a memory bandwidth of 1 byte per CPU cycle and a 1 MB shared cache. Configuration SYM-4 is for a symmetric run with a bandwidth of 4 bytes/cycle and a 4 MB cache. SGL-2 is for a single run with a bandwidth of 2 bytes/cycle and a 1 MB cache. SYM-8 is for a symmetric run with a bandwidth of 8 bytes/cycle and a 4 MB cache.

per CPU cycle and a 1 MB shared cache. SYM-8 is for a bandwidth of 8 bytes/cycle and a 4 MB cache. The shared-cache associativity remains constant and equal to 16. As can be seen the performance of SYM-4 is higher than the performance of SGL-1, and the difference is not negligible (23% for *429.mcf*). A similar conclusions holds for SYM-8 versus SGL-2, but the difference is less pronounced. The explanation of these counterintuitive results lies in memory bandwidth sharing. It is illustrated by Figure 6 with an artificial example. In our definition of a symmetric run, copies of the same program are run synchronously, meaning that they are launched at the same time. However in practice, the execution on the different cores is not exactly synchronous. In fact, perfect synchronization would be very difficult to obtain and would actually decrease self-performance. Perfect synchronization implies that if we launch the program copies exactly at the same cycle, they should finish exactly at the same cycle. But even when all cores have exactly the same microarchitectural state at the beginning of the symmetric run, and assuming the microarchitecture behavior is deterministic, the program copies do not finish exactly at the same time because certain shared resources cannot be accessed by all threads simultaneously. Consequently, there is a slight desynchronization of cores on a symmetric run. Because cache misses are often bursty, a slight desynchronization permits obtaining a more uniform utilization of the bus bandwidth. This is what Figure 6 illustrates.

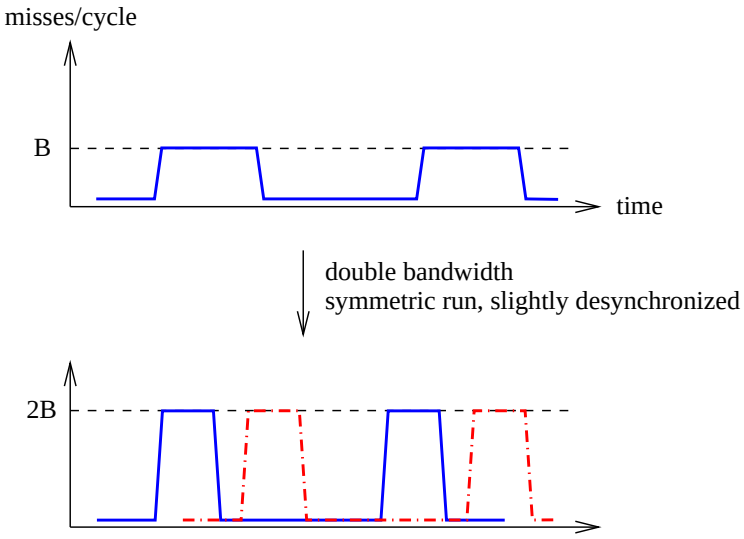


Figure 6: Example for explaining why self-performance can exceed the performance of a single run with memory bandwidth statically partitioned. This example assumes 2 cores.

3.4 Increasing the memory bandwidth may decrease performance.

Once there is an agreement between the programmer and the microarchitect that self-performance represents the minimum performance, the microarchitect must try to minimize the possibility of this not being the case. For the microarchitect, this means a special attention to each shared resource. In our simulations, only two resources are shared : the L2 cache and the bus bandwidth. The focus of this study is the cache replacement policy. But for our results to be meaningful we had to be careful with the cache indexing and with the bus arbitration policy.

L2 and L3 caches are generally indexed with physical addresses. On a symmetric run, physical indexing utilizes cache sets more uniformly than virtual indexing, so self-performance is likely to be higher than what would be measured by partitioning the cache statically and equally between cores. We already observed an analogue phenomenon with memory bandwidth in Section 3.3. However, it is difficult to exploit this phenomenon in the cache without sacrificing the performance guarantee. The self-performance would be too optimistic. Instead, the OS should implement a page coloring scheme such that the cache indexing is equivalent to using the virtual address. [2] Our simulations in this study assume a virtual indexing.

As for the bus arbitration policy, we initially implemented a simple *least-recently-selected* (LRS) scheme, which we thought would be sufficient. The LRS arbiter selects, among non-empty request

[2] For avoiding having too many constraints on page allocation, page coloring may be active only when measuring performance with the *selfperf* utility. But for a stronger performance guarantee, page coloring should be the default behavior (some operating-systems like FreeBSD already use page coloring).

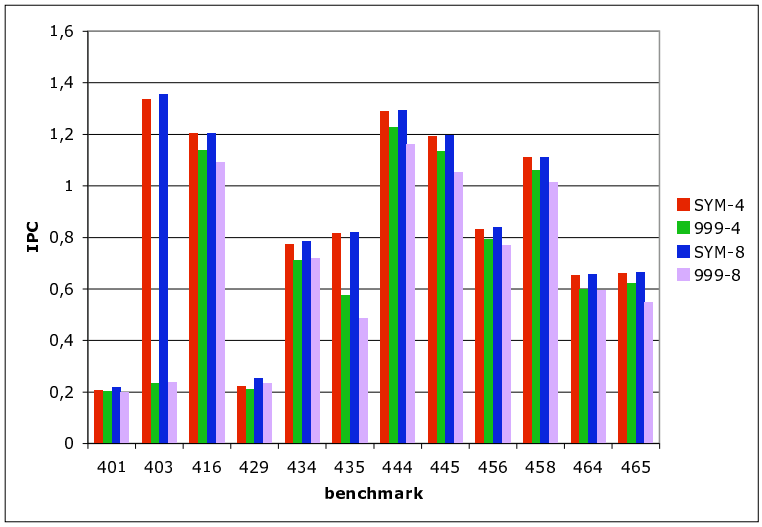


Figure 7: IPC for a subset of our benchmarks. The benchmark is run on core #1. Two workloads are considered for the 3 remaining cores : workload SYM runs a copy of the benchmark on each core (symmetric run) and workload 999 runs a copy of microbenchmark 999 on each core. For both workloads, we show the IPC when memory bandwidth is 4 bytes/cycle (SYM-4 and 999-4) and when it is 8 bytes/cycle (SYM-8 and 999-8).

queues, the least recently selected one. LRS arbitration is commonly used for arbitrating resource conflicts between threads in some multi-threaded processors like the Sun UltraSPARC T1 [6]. But we found that, when LRS is used for the bus, we cannot guarantee self-performance. To see why, consider the case of an application with a low average miss rate but whose misses occur in bursts. On a symmetric run, the desynchronization of cores permits avoiding most bus conflicts (cf. Figure 6). But when the application is run simultaneously with threads having a high average miss rate, it is granted bus access again only after each of the competing threads has accessed the bus once. Thus the application suffers from bandwidth saturation despite having a low average miss rate. To solve this problem, we have implemented a different bus arbitration policy. We associate a 4-bit up-down saturating counter with each request queue. This counter represents a *score*. To select which queue should access the bus, the arbiter chooses, among non-empty queues, the one with the lowest score. If a selection occurs (at least one queue is not empty), the score of the selected queue is incremented by X, where X is the number of running threads minus one ($X = 3$ in this study), and the score of **each** non-selected queue is decremented by 1. Moreover, to facilitate desynchronization on symmetric runs, we introduced a little randomness by not updating the scores once every 1000 selections on average. With this arbitration policy, an application with a low average miss rate has a low score and its requests can access the bus quickly even if the other threads have a high miss rate.

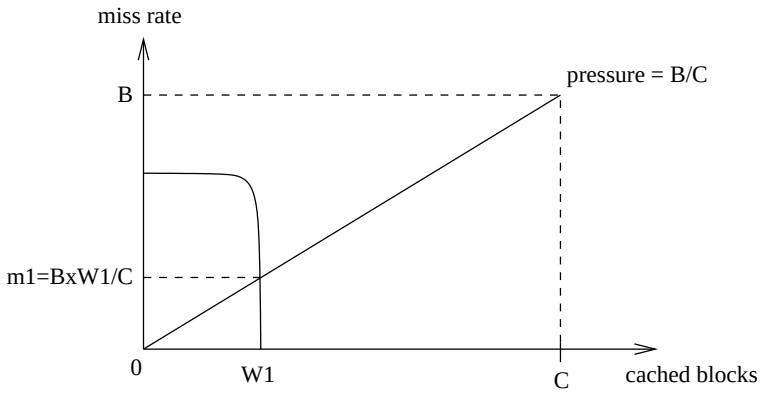


Figure 8: Cache pressure model applied for a shared cache of capacity C, assuming a bandwidth B (maximum number of misses per cycle). On this example, thread #1 has a working set of size W_1 and a miss rate curve that drops suddenly when W_1 blocks are cached. If the other threads are able to saturate the bandwidth, the miss rate m_1 of thread #1 is $\frac{W_1}{C}B$. Thus an increase of bandwidth decreases the performance of thread #1.

Figure 7 shows the IPC on core #1 when the 3 other cores run a copy of the benchmark (symmetric run) and when they run instances of microbenchmark 999. In both cases, we show the IPC when memory bandwidth is 4 bytes/cycle (SYM-4 and 999-4) and when it is 8 bytes/cycle (SYM-8 and 999-8). We show results only for benchmarks whose performance suffers from running simultaneously with microbenchmark 999. As can be seen, the actual performance can be much smaller than the self-performance. This is particularly striking for *403.gcc* and *435.gromacs*. For *403.gcc*, the actual performance can be 6 times worse than the self-performance.

Another striking observation is that increasing the memory bandwidth can decrease the performance of an application. For example, when running with microbenchmark 999, *435.gromacs* experiences a 16% performance drop when memory bandwidth goes from 4 to 8 bytes/cycle. By limiting the rate at which blocks can be evicted from the cache, a smaller bandwidth offers a better protection against aggressive cache evictions, but only to a certain extent. The cache pressure model confirms this observation. On Figure 8, we consider a thread #1 with a working set of W_1 blocks and a miss rate curve that drops suddenly when W_1 blocks are cached. The bandwidth is B (maximum number of misses per cycle). If the other threads are able to saturate the bandwidth, the miss rate of threads #1 is $m_1 = \frac{W_1}{C}B$. If we increase the bandwidth B, we increase the performance of the threads for which bandwidth is a bottleneck, but we also increase the miss rate of thread #1, hence decreasing its performance.

This situation where an obvious structural improvement (making the bus wider or faster) may decrease the performance of an application is not a healthy situation. The microarchitect does not expect an application to experience a slowdown when the memory bandwidth is increased.

4 Sharing-aware replacement (SAR) policies

Sharing-aware replacement (SAR) is intended to solve the problems we highlighted in Section 3. SAR can be applied to any replacement policy, e.g., LRU, pseudo-LRU, DIP, etc. But the details of the implementation depend on the underlying replacement policy. In this study, we use LRU SAR policies and we describe an implementation corresponding to this case. The basic idea of SAR is to take into account the cache space occupied by each thread. This requires that the *thread identifier* (TID) be stored along with each block in the cache. With 4 cores, each TID is 2-bit wide. We say that a TID is *inactive* if there are fewer running threads than cores and the TID does not correspond to a thread currently running on a core. (an inactive TID typically corresponds to a thread that has finished execution of that is waiting for an event or a system resource). A SAR policy selects a victim block as follows :

- Each TID proposes a potential victim block in the cache set
- If there is at least one invalid block in the set, we take an invalid block as the victim.
- Otherwise, the SAR policy selects a *victim TID* and the actual victim block is the victim block proposed by the victim TID.
- If the cache set contains some blocks belonging to an inactive TID, such inactive TID is chosen as the victim TID. This is for being able to exploit the full cache capacity when there are fewer running threads than cores.

For a LRU SAR policy, we must first describe how the LRU stack is implemented. The LRU stack consists of the blocks in the cache set ordered from MRU (most-recently-used) to LRU. There are several possible ways to implement a LRU stack in hardware. A solution consuming no storage at all would be to maintain a physical order among blocks, from MRU to LRU. Promoting a block to the MRU state consists in moving the block to the MRU position and shifting the other blocks accordingly. However, such implementation would consume a lot of cache bandwidth and a lot of energy. Instead, it is possible to use short pointers to the blocks and to move the pointers instead of the blocks themselves. For an associativity of 16, this requires a 4-bit pointer per block, pointing to a location in the cache set. Pointers are stored in a separate table, which we call the R-table. [3] There is one R-table entry for each cache set. Each R-table entry contains sixteen 4-bit pointers, ordered from MRU to LRU. Moreover, we assume that the 2-bit TIDs are stored in the R-table. So each block in the R-table is represented by $4 + 2 = 6$ bits. Updating the LRU stack requires an associative search among the 16 pointers and moving the matching block to the MRU position. The victim block proposed by a given TID is the block belonging to that thread whose position in the stack is closest to the LRU position. To obtain the victim proposed by a given TID, sixteen 2-bit comparators provide a 16-bit vector where each bit indicates whether or not the corresponding block belongs to the thread. Then a priority encoder finds, in the 16-bit vector, the "1" closest to the LRU position. [4]

[3] These pointers are not part of the SAR hardware cost, they implement the LRU policy.

[4] The hardware we have described so far is not more complex than what would be necessary to implement programmable quotas. But papers describing quota-based solutions sometimes skip these details.

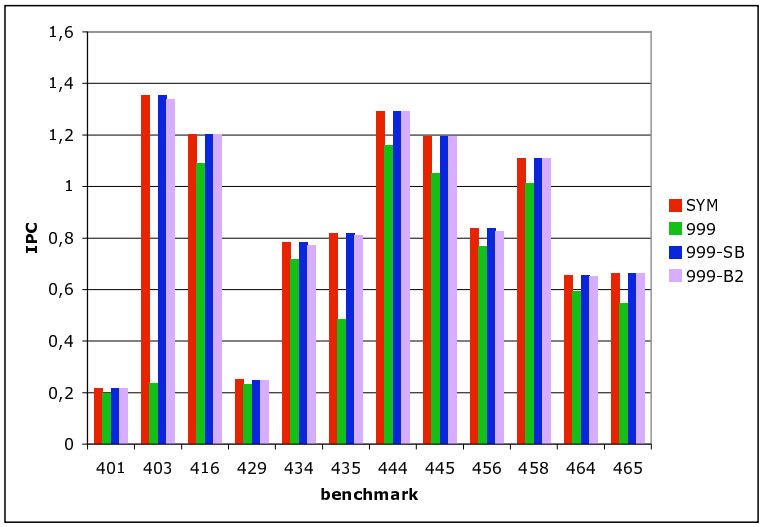


Figure 9: The SB policy makes the worst-case performance (999) close to self-performance (SYM). The B2 policy, simpler than SB, is almost as effective.

4.1 The SAR SB policy

A possible solution for ensuring that a thread gets the cache space it would get on a symmetric run is to give the same amount of cache space to each thread. This can be done by choosing as victim TID the TID with the largest number of cached blocks. In case of equality, we choose the TID whose proposed victim is closest to the LRU position. Such policy should progressively converge to an equilibrium partition where all threads get an equal share. There are two possible options. The number of blocks may be computed either for the whole cache or just for the cache set. We denote the first policy *global-biggest* (GB), and the second one *set-biggest* (SB). The GB policy chooses as victim TID the TID with the largest number of blocks in the whole cache, while the SB policy chooses as victim TID the TID with the largest number of blocks in the cache set where the missing block goes. The GB policy can be implemented by maintaining 4 counters giving the total number of blocks belonging to each thread. On a miss, one or two counters are updated. The SB policy can be implemented by counting blocks on-the-fly while the miss request is being processed. [5]

Simulation results for the SB policy are shown in Figure 9. The SB policy is successful at making worst-case performance close to self-performance. This was expected, as the SB policy converges relentlessly to a state where each cache set is evenly divided between competing threads. Actually, we found that the GB policy is not safe and we do not show results for it. We have mentioned it just to emphasize the necessity of working at the set level. The main reason why the GB policy is not

[5] Actually, when counting blocks, we consider the 17 blocks consisting of the 16 blocks in the cache set plus the missing block.

safe is that it does not guarantee that each cache set is evenly divided between threads. Indeed, some applications do not use cache sets uniformly. For example, we simulated benchmark 429.mcf with 3 instances of microbenchmark 999 compiled with $STEP = 32$, i.e., using only even cache sets. With a GB policy, the performance of 429.mcf is 22% lower than the self-performance. The fact that one must work at the set level to obtain a strong performance guarantee has already been observed in [10].

4.2 The SAR B2 policy

The SB policy requires to find the TID that has the most blocks in a set. With 4 cores, this requires 3 comparisons. We propose a simpler SAR policy, that we call *biggest-of-two*, or B2 for short. Like the SB policy, the B2 policy counts the 17 blocks in the cache set concerned by the miss (16 cached blocks plus the missing block). While processing the cache miss, the B2 policy chooses a random block in the set. The TID of this block is denoted the *random TID*. The TID of the missing block is denoted the *missing TID*. The B2 policy chooses the victim TID between the missing TID and the random TID, choosing the one that has the largest number of blocks among the 17 blocks. In case of equality, the random TID is chosen as victim TID. In other words, the victim is the random TID unless the missing TID has more blocks in the sets. Unlike the SB policy, on a 4-core processor, the B2 policy requires a single comparison. Counting blocks is not necessary if we have a circuit that compares two 17-bit vectors and tells which one contains the most 1's. As can be seen in Figure 9, the B2 policy is practically as efficient as the SB policy.

It should be noted that the B2 policy is simpler to implement than programmable quotas. With programmable quotas, the per-set share allotted to a thread depends on the number of contenders in that set (which may be less than the number of threads). In [10], computing the per-set share for the SQVP policy requires determining the number of contenders (i.e., threads that have at least one block in the set) and doing a division. The hardware for computing per-set shares is not described in [10].

5 Implications of our proposition

5.1 Programmable TIDs

Although our proposition is less flexible than programmable quotas, it is possible to have some control on the shared cache (and more generally on shared microarchitectural resources). Until now, we have assumed that threads running simultaneously had different TIDs. But if the TIDs are programmable, we are not constrained to using different TIDs. For example, if we know that the applications running have no QoS requirements, it is not necessary to guarantee self-performance. In this case, if we want the cache to behave like a conventional shared cache (for whatever reason), we can give the same TID to all threads. As another example, consider the case where we have 4 threads and, for whatever reason, we want to give half of the shared cache capacity to one of the threads. To do this, we use one TID for the thread we want to advantage, and a second TID that is shared by the 3 other threads.

400 429 437	401 433 444	403 434 445	410 435 447
416 436 450	429 437 453	433 444 454	434 445 456
435 447 458	436 450 459	437 453 462	444 454 464
445 456 465	447 458 470	450 459 471	453 462 473
454 464 482	456 465 483	458 470 400	459 471 401
462 473 403	464 482 410	465 483 416	470 400 429
471 401 433	473 403 434	482 410 435	483 416 436

Table 2: 28 workloads running on cores #2, #3 and #4 respectively

5.2 Impact on average performance

We have mentioned in Section 5.1 that having programmable TIDs permits emulating a conventional shared cache. The machine owner may prefer this configuration if applications have no explicit performance targets, if there are more jobs than cores, and if he wants to take advantage of symbiotic jobscheduling to maximize throughput [11]. On the other hand, if some applications have QoS requirements, different TIDs should be given to different threads. Yet, the machine owner still wants a high throughput. Until now, we have focused exclusively on making the worst performance as close as possible to the self-performance, so that self-performance can serve as a measure of performance when the multicore workload is unknown at programming time. However, for maximizing throughput, what is important is the average performance. The average IPC of an application can be estimated by computing the arithmetic mean of the application IPC when the application runs with various workloads. There is a direct relation between average performance and throughput. If the multicore is time-shared between a given set of applications and if each application gets the same fraction of CPU time, the average throughput is equal to the number of cores times the arithmetic mean of the average IPCs of applications.

Compared with natural cache partitioning, the SB and B2 policies should increase the performance of applications with a low miss rate and a small working set, but should decrease the performance of applications with a high miss rate and a working set whose size is larger than the equal-partition share but smaller than the cache (cf. the cache pressure model). To measure the average IPC, we ran each benchmark on core #1 and obtained its IPC when the 3 other cores run the 28 different workloads given in Table 2 (with 28 benchmarks, this requires $28 \times 28 = 784$ simulations). The average IPC of each benchmark is the arithmetic mean of the 28 different IPCs measured for this benchmark on the 28 workloads. Results are given in Figure 10. As expected, the SAR SB and B2 policies decrease the average IPC on a few benchmarks (401,429,450) and increase it on a few others (434,435,456). Overall, the SAR policies do not have a significant impact on the average performance. They just provide a different trade-off. This means that, from the point of view of throughput, SAR policies are practically equivalent to conventional replacement policies.

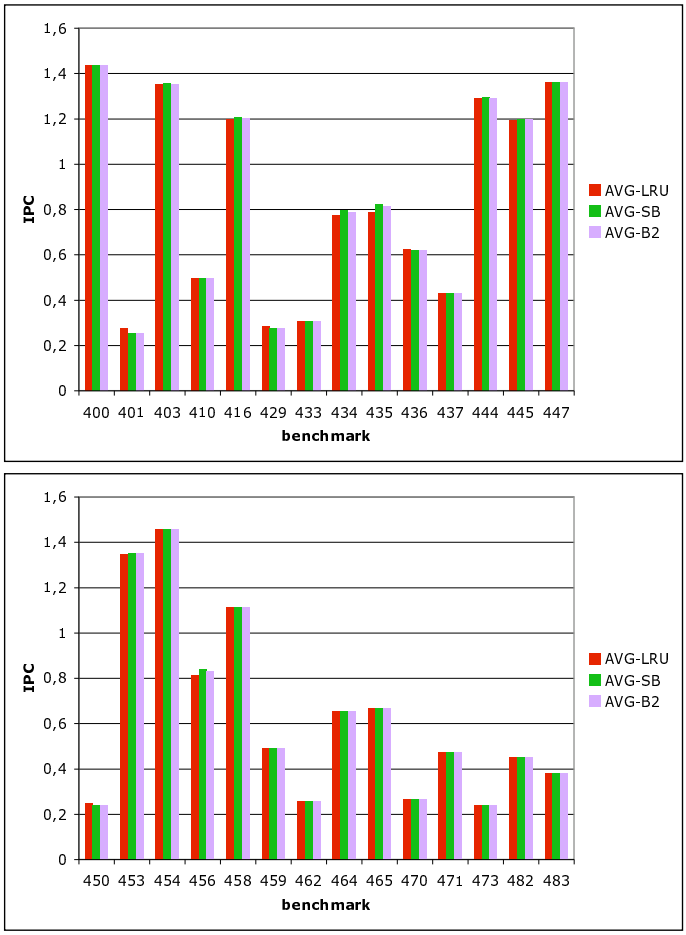


Figure 10: Average IPC for each benchmark. The average is computed over the 28 workloads of Table 2, under natural partitioning (AVG-LRU), SAR SB (AVG-SB) and SAR B2 (AVG-B2).

5.3 Multi-threaded programs

In this study, we have focused on sequential applications. But providing the means to obtain a performance guarantee for multi-threaded programs is also very important. Indeed, multi-thread programming is difficult, and the programmer is willing to invest effort in multi-thread programming provided the level of performance he has striven to obtain can be reproduced. Performance may be difficult to reproduce when the application has fewer threads than cores. If the number of threads is a divisor of the number of cores, the concept of self-performance applies and performance can be measured with the *selfperf* utility. If the number of threads is not a divisor of the number of cores, the concept of self-performance does not apply, and the only way to obtain a performance guarantee is to reserve all the cores.

It should be noted that programmable TIDs offer some flexibility to the programmer. If the programmer is more comfortable (performance wise) with a shared cache, the same TID can be given to all the threads. On the other hand, if the programmer wants to optimize the cache locality of each thread separately, he may prefer to give a different TID to each thread to emulate a partitioned cache.

6 Conclusion

We introduced the concept of self-performance, which is a contract between the programmer and the microarchitecture. The programmer measures performance by running a copy of the application on each core, and the microarchitecture guarantees this level of performance independently of the characteristics of the applications running on the other cores. For the programmer, the advantage of self-performance is that it is conceptually simple and does not require any knowledge of internal microarchitectural details. For the microarchitect, respecting the self-performance contract means paying attention to each microarchitectural resource that is shared between threads. In this context, shared caches are critical. We have shown that unmanaged sharing is incompatible with the self-performance contract. We have proposed sharing-aware cache replacement (SAR) policies that are compatible with self-performance. The SAR B2 policy is simpler to implement than solutions based on programmable quotas. This simplification of the hardware was obtained by sacrificing some flexibility. Nevertheless, with programmable thread-IDs, our solution allows the programmer and the OS to have some control on the cache behavior.

The performance guarantee offered by SAR policies is not absolute, in the sense that it is very difficult to prove the guarantee mathematically without getting rid of resource sharing (this is the case also for quota-based solutions). Nevertheless, our experiments and simulations have shown that the situation is much better with our proposition than without it.

A Simulation methodology and benchmarks

Our simulator is trace-driven, using traces generated with Pin [7]. We have one trace per benchmark listed in Table 3. To obtain each trace, we run the application without any instrumentation for several seconds, then we send a signal that triggers instrumentation.

benchmark	input	skip
400.perlbench	checkspam.pl	30 s
401.bzip2	liberty.jpg	30 s
403.gcc	166.i	30 s
410.bwaves		30 s
416.gamess	cytosine.2.config	30 s
429.mcf		30 s
433.milc		30 s
434.zeusmp		30 s
435.gromacs		30 s
436.cactusADM		30 s
437.leslie3d		30 s
444.namd		30 s
445.gobmk	13x13.tst	30 s
447.dealII		30 s
450.soplex	pds-50.mps	20 s
453.povray		30 s
454.calculix		30 s
456.hmmer		30 s
458.sjeng		30 s
459.GemsFDTD		30 s
462.libquantum		30 s
464.h264ref	foreman_ref_encoder_baseline.cfg	30 s
465.tonto		30 s
470.lbm		30 s
471.omnetpp		30 s
473.astar	BigLakes2048.cfg	30 s
482.sphinx3		30 s
483xalancbmk		30 s

Table 3: One trace was obtained for each SPEC CPU2006 benchmarks, except 481.wrf that we could not compile. For each benchmark, we start instrumenting after a certain execution time has elapsed.

We simulate 4 identical cores, each with dedicated L1 caches. The L2 cache is 4 MB, 16-way set-associative and is shared between the 4 cores. The next level after the L2 cache is the off-chip DRAM. The main characteristics of the simulated microarchitecture are summarized in Table 1. The simulator does not model all details of the execution core. In particular, we do not model data dependencies between instructions. But the memory hierarchy is simulated with a lot of details. In particular, we simulate contention for the L2 cache, contention for the memory bus, and write-back traffic. All caches are non blocking, i.e., they continue to be accessed even if a previous request generated a miss. All misses are fully pipelined. Unless stated otherwise, the bus bandwidth to DRAM is 8 bytes per CPU cycle. There is a separate memory request queues (MRQs) for each core. Each MRQ has room for 20 pending requests. Once a request is selected by the arbiter and is sent on the bus, there is a latency of 300 cycles for getting the requested block. The request is removed from the MRQ after the block has returned from memory. Blocks evicted from write-back caches are buffered in write-back queues. When arbitrating for a resource (cache or bus), reads have priority over writes. Cache refills are blocked when the associated write-back queue is full. A miss request is not schedulable if it cannot get a cache refill queue entry (i.e., the cache refill queue is full).

References

[1] F. Guo, H. Kannan, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis. From chaos to QoS : case studies in CMP resource management. *ACM SIGARCH Computer Architecture News*, 35(1), 2007.

[2] G. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multiprocessors. In *Proceedings of the 40th Annual International Symposium on Microarchitecture*, 2007.

[3] R. Iyer. CQoS : a framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the International Conference on Supercomputing*, 2004.

[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 *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2007.

[5] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[6] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *IEEE Micro*, March/April 2005.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, 2005.

[8] K.J. Nesbit, J. Laudon, and J.E. Smith. Virtual private caches. In *Proceedings of the 34th International Symposium on Computer Architecture*, 2007.

[9] M. Qureshi, A. Jaleel, Y.N. Patt, S.C. Steely Jr, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[10] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proceedings ot the 15th International Conference on Parallel Archietctures and Compilation Techniques*, 2006.

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

[12] H.S. Stone, J. Turek, and J.L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(4):1054–1068, September 1992.