HAL
archives-ouvertes.fr

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. 6th International Conference on High-Performance and Embedded Architectures and Compilers, Jan 2011, Heraklion, Greece. 2011, <10.1145/1944862.1944890>. <inria-00531188>

HAL Id: inria-00531188
https://hal.inria.fr/inria-00531188

Submitted on 2 Nov 2010

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.

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

Pierre Michaud
INRIA Rennes - Bretagne Atlantique
Campus de Beaulieu
35042 Rennes Cedex, France
Pierre.Michaud@inria.fr

ABSTRACT

The presence of shared caches in current multicore processors may generate a lot of performance variability in multiprogrammed environments. For applications with quality-of-service requirements, this performance variability may lead the programmer to be overly pessimistic about performance and reduce the application features and/or spend a lot of effort optimizing the algorithms. To solve this problem, there must be a way for the programmer to define a reasonable performance target and a guarantee that the actual performance is very unlikely to be below the targeted performance. 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.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Miscellaneous

General Terms

Performance

Keywords

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

1. INTRODUCTION

There exists an implicit contract between the programmer and the processor. When a programmer writes a program for which performance matters and evaluates performance by running the program, he often assumes that performance is deterministic, hence reproducible. Actually, there can be some performance variability, some due to the operating system, some due to the runtime system. But this performance variability can generally be tackled in software (e.g., by minimizing the number of system calls, by choosing the appropriate programming language, etc.). With multicore processors able to execute several applications simultaneously, performance variability may stem from the sharing of microarchitectural resources by several applications running concurrently. Shared caches, for instance, are among the microarchitectural resources that incur the most performance variability. But the programmer has little control on this performance variability. Moreover, the microarchitecture of today's processors is not always documented and it is difficult for the programmer to understand what is happening.

Depending on workload characteristics, the actual performance of a particular application may be much smaller than the performance measured at programming time. For applications with quality-of-service requirements (e.g., a video decompressor), this may lead the programmer to be overly pessimistic about performance and reduce the application features and/or spend a lot of effort optimizing the algorithms.

Previously proposed solutions to this problem involve the use of programmable priorities or quotas [7, 4, 14, 2, 12, 3, 5]. 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 to this problem, which is to have an explicit contract between the programmer and the microarchitecture. 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.

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. Finally, Section 6 concludes this work.

Simulations.

The simulation results presented in this study correspond to a simulated 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

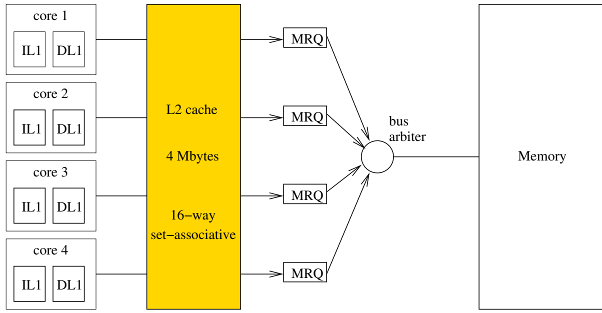

Figure 1: Symmetric multicore simulated in this study.

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 KB, 25-bit global history, 8-bit tags
branches	10-cycle minimum misprediction penalty, solved at retirement
IL1 cache	private, 32 KB, 4-way SA LRU, 64-byte block, 1 cycle lat, 1 block refill & 2 instructions read per cycle
DL1 cache	private, 32 KB, 4-way SA LRU, 64-byte block, write-back write-alloc, 2 cycles lat, 1 block refill & 1 ld-st/cycle
L2 cache	shared, 4 MB, 16-way SA LRU, 64-byte block, write-back write-alloc, 15 cycles lat, bandwidth 1 block/cycle
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

microarchitecture are summarized in Table 1. Our simulator is trace-driven, using traces generated with Pin [9]. We obtain one trace for each CPU 2006 benchmark[1]. To obtain each trace, we run the application without any instrumentation for 30 seconds, then we send a signal that triggers instrumentation. More details about our approximate simulation methodology can be found in [10]. We denote each SPEC CPU2006 benchmark by its SPEC number. 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.

2. SELF-PERFORMANCE

In this study, we consider independent sequential tasks. 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) requirements, it is important that performance measured at programming time be deterministic, or close to deterministic. 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 requirements. On a multicore, the operating system 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 [7, 4, 14, 2, 12, 3, 5]. 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 requested resources. However, this raises a question : what if the sum of resources requested 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 requests [14]. 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[2]. We call it *self-performance*. Self-performance is less flexible than programmable quotas but we believe it is simpler for the programmer and more practical.

2.2 Self-performance

Obtaining a performance guarantee is a two-stage problem :

- We need a way to define a performance target.
- 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

[1] Except 481.wrf, that we could not compile.

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

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 concurrently with some 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 internal microarchitecture 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 constraints 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, to our knowledge, existing processors do not allow the OS to control microarchitectural resources more finely.

Among shared microarchitectural resources, caches exhibit the most chaotic and hard-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 in performance and energy. 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, self-performance can exceed actual performance

The model of cache partitioning proposed in [15], 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_iT 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 $m_iT = mT \times w_i/C$, that is,

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

This quantity, m_i/w_i, was not identified in [15]. 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 cannot guarantee that the actual performance will reach the self-performance

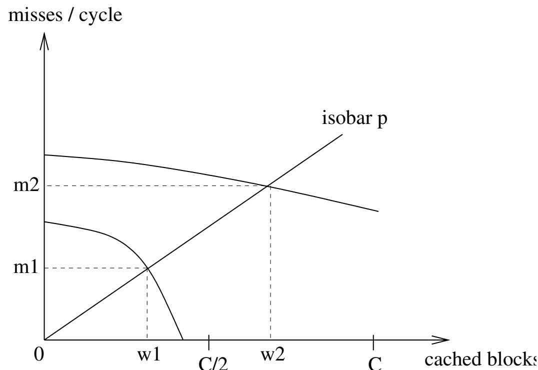

Figure 2: Example with 2 threads. The miss rate m_i of thread i is assumed to be a function of the number w_i of cached blocks. The equilibrium partition (w_1, w_2) is such that the two threads have equal cache pressure $p = m_1/w_1 = m_2/w_2$, hence the points (w_1, m_1) and (w_2, m_2) lie on the same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

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.

3.2 Self-performance is not always 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 should get the same share of the cache capacity. Therefore, a symmetric run on 4 cores should result in each

```
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*)

thread getting one fourth of the cache capacity. However, the cache pressure model is only an approximation of reality. It is useful for explaining certain qualitative phenomena, but it makes no assumption on the replacement policy and therefore cannot model policy-dependent behaviors.

Empirically, from our experiments and simulations, we believe that LRU is unlikely to generate strange performance variations on symmetric runs. But this is not necessarily the case with other replacement policies. Though we present results only for LRU in this study, we also did simulations with a variant of the DIP replacement policy where each core has a distinct PSEL counter.

DIP was recently proposed as an improvement over LRU for L2 and L3 caches [13]. DIP (or DIP-inspired policies [6, 11]), is likely to be implemented in future processors. All our observations and conclusions with LRU are the same with DIP, except that natural cache partitioning under DIP may lead to strong performance asymmetry on symmetric runs. We found that natural cache partitioning does not always guarantee that a symmetric run leads to a balanced cache partitioning.

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 explained as follows. From 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. The reason is that there is a negative feedback at work : 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 with the LRU policy. 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. DIP is based on a Bimodal Insertion Policy (BIP). With BIP, 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).

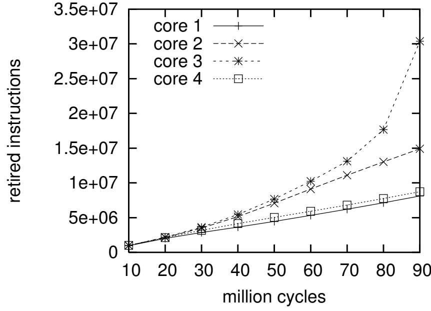

Figure 4: Symmetric run of microbenchmark *999* with $SIZE = 2^{19}$ (Figure 3). Memory bandwidth is 4 bytes/-cycle and the L2 replacement policy is a variant of DIP (one PSEL counter per thread). The plot shows the number of retired instructions on each core as a function of time.

In such case, BIP 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. Such chaotic behavior is *a priori* incompatible with providing a performance guarantee. The SAR policies proposed in Section 4 solve this potential 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 isolated 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 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 counter-intuitive results lies in memory bandwidth sharing. This is illustrated in Figure 6 with an artificial example. With 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

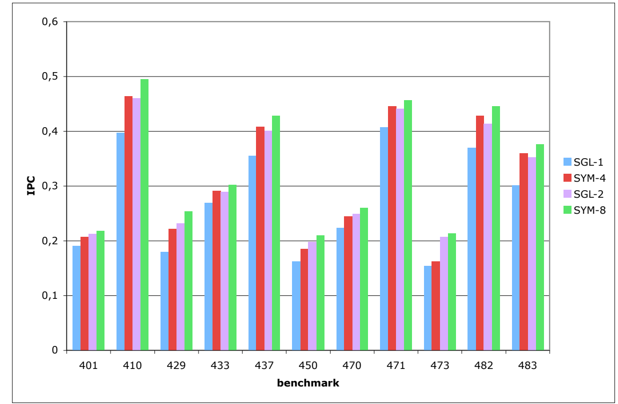

Figure 5: IPC per benchmark. Configuration SGL-1 is for an isolated run with a memory bandwidth of 1 byte per CPU cycle and a 1 MB shared cache. Configuration SYM-4 is for a symmetric run with a bandwidth of 4 bytes/cycle and a 4 MB cache. SGL-2 is for an isolated run with a bandwidth of 2 bytes/cycle and a 1 MB cache. SYM-8 is for a symmetric run with a bandwidth of 8 bytes/cycle and a 4 MB cache.

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.

3.4 Increasing 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

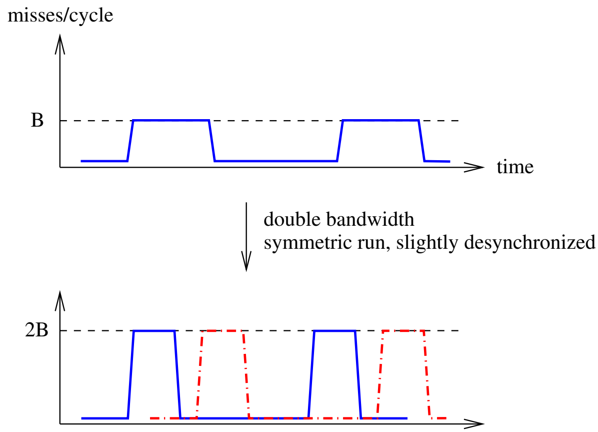

Figure 6: Example for explaining why self-performance can exceed the performance of an isolated run with memory bandwidth statically partitioned. This example assumes 2 cores.

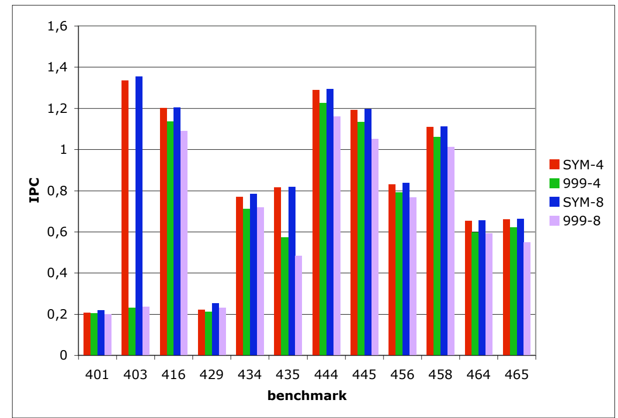

Figure 7: IPC for a subset of our benchmarks. The benchmark is run on core #1. Two workloads are considered for the 3 remaining cores : workload SYM runs a copy of the benchmark on each core (symmetric run) and workload 999 runs a copy of microbenchmark 999 on each core. For both workloads, we show the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

to using the virtual address[3]. 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 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 [8]. 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. 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 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 bandwidth, the cache pressure is B/C (cf. equation (1)) and the miss rate of threads #1 is $m_1 = \frac{W_1}{C} B$. If we increase bandwidth B, we increase cache pressure and the miss rate of thread #1, hence we decrease thread #1 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 memory bandwidth is increased.

4. SHARING-AWARE REPLACEMENT POLICIES

Sharing-aware replacement (SAR) is intended to solve the

[3] 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).

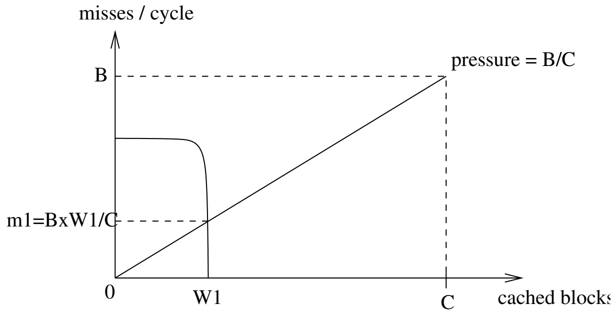

Figure 8: Cache pressure model. Cache capacity is C and bandwidth (maximum number of misses per cycle) is B. On this example, thread #1 has a working set of size W_1 and a miss rate curve that drops suddenly when W_1 blocks are cached. If the other threads are able to saturate bandwidth, the cache pressure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

problems highlighted in Section 3.

The basic idea of SAR is to take into account the cache space occupied by each thread. This requires that a *thread identifier* (TID) be stored along with each block in the cache. For instance, with 4 logical cores, each TID is 2-bit wide. We say that a TID is *inactive* if there are fewer running threads than logical cores and the TID does not correspond to a thread currently running (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.
- 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.

The last point is for being able to exploit the full cache capacity when there are fewer running threads than cores

The implementation of the first point (having each thread proposing a potential victim block) is very dependent of the underlying replacement policy. A SAR policy requires little storage overhead compared to a conventional policy. For instance, SAR policies based on a LRU stack, like DIP [13], or those based on the Not-Recently-Used (NRU) policy, like DRRIP [6], do not require any extra storage apart from per-block TIDs. The victim selection logic, though, is slightly more complex. SAR policies based on CLOCK [16, 11] require one clock hand per thread and incur some storage overhead.

In this study, we assume LRU SAR policies : the potential victim block for a thread is the least-recently used among blocks belonging to that thread.

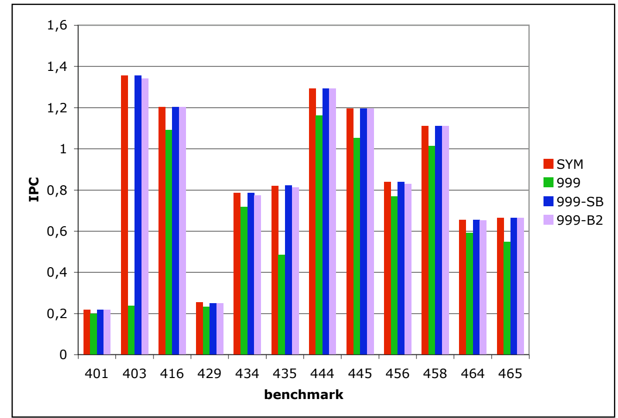

Figure 9: The SB policy makes the worst-case performance (999) close to self-performance (SYM). The B2 policy, simpler than SB, is almost as effective.

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[4].

Simulation results for the SB policy are shown in Figure 9. The SB policy is successful in 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 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 performance guarantee has already been observed in [14].

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

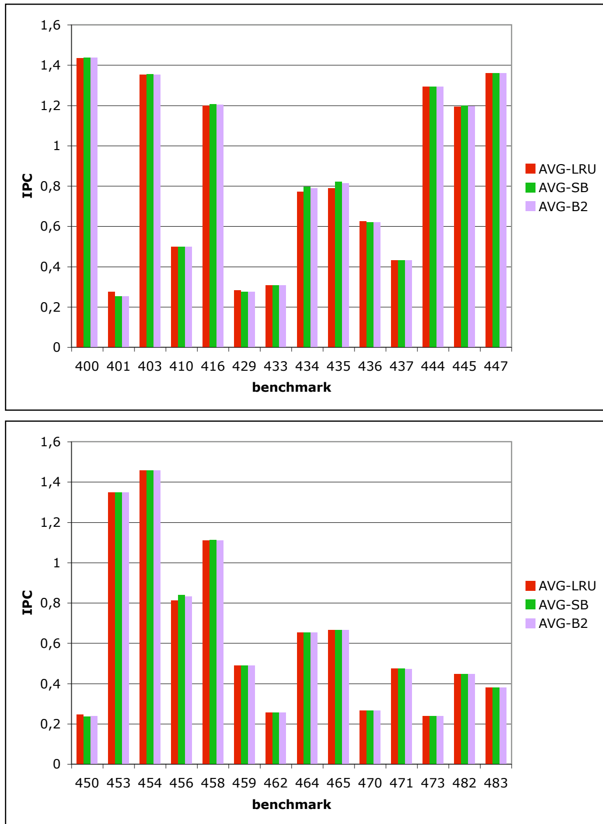

Figure 10: Average IPC for each benchmark. The average is computed over the 28 workloads of Table 2, under natural partitioning (AVG-LRU), SAR SB (AVG-SB) and SAR B2 (AVG-B2).

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 how many blocks belong to each thread among the 17 blocks (16 blocks in the cache set, plus the missing block).

While processing the cache miss, the B2 policy chooses a random block in the cache 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 may not be 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 effective as the SB policy.

5. IMPLICATIONS

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

5.1 Programmable TIDs

Although our proposition is less flexible than programmable quotas, it is still possible to have some control on the shared cache (and more generally on shared microarchitectural resources), though this is not the spirit of our proposition.

So far, we have assumed that threads running simultaneously had different TIDs. But if TIDs are programmable, we are not constrained to using different TIDs. If applications running have no QoS requirements and if we want the cache to behave like a conventional shared cache (for whatever reason), we can give the same TID to all threads. Also, it is possible to favor a thread. For instance, if we have 4 threads and if we want to give half of the shared cache capacity to one of the threads (for whatever reason), we can use one TID for the thread we want to favor, and a second TID that is shared by the 3 other threads.

5.2 Impact on throughput

When several applications run simultaneously, and compared with conventional LRU, the LRU SAR policies should increase the performance of some applications and decrease the performance of some others. 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. 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). On average, SAR LRU policies have little impact on total throughput.

Of course, it may be possible to devise replacement policies specifically aiming at maximizing total throughput, like some recently proposed policies [1]. But in general, maximizing throughput is incompatible with providing performance guarantees.

5.3 Multi-threaded programs

So far we have considered sequential applications only. Obtaining a performance guarantee for a sequential application is not so easy, but obtaining a performance guarantee for a parallel application is even more difficult.

A possibility would be to reserve all the cores for one application. But if the number of threads in the application is less than the number of logical cores, processing resources are wasted.

Another possibility would be to let the programmer define a maximum number of threads n_t for the application. Self-performance can be measured with the *selfperf* utility running $\lfloor n_c/n_t \rfloor$ copies of the application simultaneously, with n_c the number of logical cores. Then, enforcing self-performance for that application requires to limit the number of applications running simultaneously to $\lfloor n_c/n_t \rfloor$. If we try to run simultaneously several applications requiring a performance guarantee, it is the application with the largest n_t that determines how many applications can run.

Of course, this would address only one of the many problems one would need to solve to provide a performance guarantee to parallel applications. The non-determinism inherent to some parallel programs seems *a priori* incompatible with obtaining a performance guarantee in the general case.

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 performance guarantee offered by SAR policies is not strict, 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 better with our proposition than without it.

Yet, as the number of on-chip core increases, we believe that it will be more and more difficult to exploit the computing power of multicores without introducing significant performance variability. Increasing the number of on-chip cores without increasing (or worse, decreasing) the guaranteed performance of existing applications may limit the applicability of future multicores. Moreover, will programmers accept to spend some effort parallelizing their applications if they cannot have a performance guarantee ? We believe microarchitects should pay attention to these questions.

7. REFERENCES

[1] 1st JILP Workshop on Computer Architecture Competitions, Cache Replacement Championship, 2010. In conjunction with ISCA-37. http://www.jilp.org/jwac-1/.

[2] 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.

[3] 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.

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

[5] 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.

[6] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, June 2010.

[7] 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.

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

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. 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.

[10] P. Michaud. Replacement policies for shared caches on symmetric multicores: a programmer-centric point of view. Technical Report PI-1908, IRISA, Nov. 2008. http://hal.inria.fr/inria-00340545/en.

[11] P. Michaud. The 3P and 4P cache replacement policies. In *Proceedings of the 1st JILP Workshop on Computer Architecture Competitions, Cache Replacement Championship*, June 2010.

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

[13] 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.

[14] 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.

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

[16] J. Zebchuk, S. Makineni, and D. Newell. Re-examining cache replacement policies. In *Proceedings of the IEEE International Conference on Computer Design*, 2008.