Preventing Denial-of-Service Attacks in Shared CMP Caches

Georgios Keramidas, Pavlos Petoumenos, Stefanos Kaxiras, Alexandros Antonopoulos, and Dimitrios Serpanos

Department of Electrical and Computer Engineering, University of Patras, Patras, Greece

Abstract. Denial-of-Service (DoS) attacks try to exhaust some shared resources (e.g. process tables, functional units) of a service-centric provider. As Chip Multi-Processors (CMPs) are becoming mainstream architecture for server class processors, the need to manage on-chip resources in a way that can provide QoS guarantees becomes a necessity. Shared resources in CMPs typically include L2 cache memory. In this paper, we explore the problem of managing the on-chip shared caches in a CMP workstation where malicious threads or just cache "hungry" threads try to hog the cache giving rise to DoS opportunities. An important characteristic of our method is that there is no need to distinguish between malicious and "healthy" threads. The proposed methodology is based on a statistical model of a shared cache that can be fed with run-time information and accurately describe the behavior of the shared threads. Using this information, we are able to understand which thread (malicious or not) can be "compressed" into less space with negligible damage and to drive accordingly the underlying replacement policy of the cache. Our results show that the proposed attack-resistant replacement algorithm can be used to enforce high-level policies such as policies that try to maximize the "usefulness" of the cache real estate or assign custom space-allocation policies based on external QoS needs.

1 Introduction

In application domains that range from information access to electronic commerce, many services are susceptible to attacks by malicious clients that can significantly degrade their performance. One kind of attack, called Denial-of-Service (DoS) attack, is a malicious attempt by a single person or a group of people to cripple an online service. This can have serious consequences for companies such as Amazon and eBay which rely on their online availability to do business. In the past, many companies fell victim to DoS attacks resulting in a damage of million of dollars [14][15]. Moreover, service providers may be forced by the customer requirements to provide specific QoS guarantees. In this case, the providers must assure the service quality of their services by assigning a specific amount of resources (i.e. CPU cycles). On the architecture front, processor designers are fast moving towards multiple cores on a chip to achieve new levels of performance. The target is to hide as much as possible

S. Vassiliadis et al. (Eds.): SAMOS 2006, LNCS 4017, pp. 359 – 372, 2006.

the long memory latencies. CMPs are becoming the dominant architecture for many server class machines [8][9][10]. For reasons of efficiency and economy of processor area, the sharing of some chip resources is a necessity. The shared resources in CMPs typically include the lower level caches. Those shared resources in CMPs create a need for fair and efficient management policies. A trivial solution would be to statically partition the shared resources among the running threads. However, this design point is inefficient in resource utilization when the demand is not uniform.

From another point of view, having caches shared between threads provides a vastly more dangerous avenue of attack —a DoS attack [16]. A malicious application can abuse the shared cache rendering the whole system practically inoperative, since the L2s are a critical element in the performance of all modern computers. Furthermore, according to [18], even through a DoS attack is usually intentional and malicious, such types of attacks can sometime happen accidentally. For example, one person running a memory or CPU intensive program in a multiuser machine can cause all the other users of the system to experience an extreme slowdown even if the running program is not by nature malicious. Furthermore, poor programming, either in choice of algorithm or in implementation, can also cause programs to consume resources disproportionately. This is in accordance to the problem of attack detection: sometimes it is impossible to distinguish between memory or CPU intensive applications from DoS attacks, since they operate indentically. Hence, a desirable characteristic of all the methods against DoS is to manipulate the system threads in a fair and/or efficient manner without the need to distinguish between malicious and normal threads.

To model and understand cache sharing we have built a new theoretical framework that accurately describes applications interplay in shared caches. Our cache model, named StatShare, is derived from the StatCache statistical cache model [6], which yields the miss ratio of an application for any cache size. While the StatCache model uses the number of memory references as its unit of time, StatShare uses the number of cache replacements at the studied cache level [4] as the unit of time. This allows for a natural mapping of the cache statistics to the shared cache level. This further leads to a very efficient implementation of the StatShare which enables on-line analysis feeding a dynamic resource scheduler. StatShare can predict miss rate with great success as a function of the active cache ratio used by an application.

We also demonstrate how online StatShare results can be used as inputs to a resource sched-uler. We model and evaluate a cache resource sharing strategy based on Cache Decay, originally proposed for leakage reduction [7]. Our proposal introduces important differences. A decayed cacheline is simply available for replacement rather than turned-off for leakage. Thus, hits on decayed lines are allowed. Secondly, the decay interval is measured not in cy-cles but in CAT time.

Our modified attack resistant cache replacement algorithm has the added advantage that it does not need to classify a thread (client) as malicious or not malicious permanently, but instead computes this based on recent behavior. Hence, our algorithm performs a kind of dynamic check on thread's behavior. This is an important feature, since it is possible that a normal thread may be misclassified as malicious, through this classification will change with time. As an example, a thread that has poor locality may have a low hit rate (and try to hog the cache), resulting in

its being identified as malicious, by our approach, and its eventual compression into less space. However, this does not significantly impact the performance, because the thread is already experiencing a low hit rate and hence higher latencies.

Structure of this paper. Section 2 surveys related work and reviews the StatCache model. Section 3 presents our StatShare model. Section 4 describes how cache decay can be intergrated into the StatShare model and provide attack resistant high-level cache management policies. Section 5 presents implementations and Section 6 our results. Section 7 summarizes the paper.

2 Related Work

Cache Partitioning Schemes. The issue of cache fairness has been initially investigated by Kim et al. [2]. They introduce a set of metrics for fair cache sharing and they implemented a static partitioning algorithm for the OS scheduler, and a dynamic three-part algorithm (ini-tialization, rollback and re-partitioning) for shared-cache partitioning. Their algorithms are based on stack-distance counters but do not restrict the cache replacement algorithm to LRU. Their partitioning algorithm is based on counters and partitioning registers. When a process is under-represented in the cache it starts to pick its victims from other processes, while when it is over-represented, it picks its victims among its own lines.

In [3], Kim et al. extend their previous work with three performance models that predict the impact of cache sharing on co-scheduled threads. The input to the models is the isolated second-level cache stack distance of the applications and the output is the number of extra second-level cache misses for each thread due to cache sharing. Suh et al. [1] studied partitioning the cache among sharers by modifying the LRU replacement policy. The proposed mechanism used in their scheme is the same as the one used by Kim et al. [2], but their focus is in performance and not fairness.

Denial-of-Service at the Architectural Level. One of the initial attempts to prevent DoS attacks at the architectural level was the one introduced by Soderquist and Leeser [19]. The authors proposed the idea of cache locking where the locked cachelines were not allowed to be removed from the cache, quaranteeing freedom from DoS attacks. In their approach, a dynamic cache locking technique, aided by custom processor instructions, treat locked cache lines as additional registers.

Recently, many researchers studied the issue of DoS attacks in the context of SMT processors. Because multiple threads share many resources (pipeline, execution units etc.) in a SMT, there are many opportunities for a malicious thread to launch a DoS attack by abusing shared resources. Grunwald and Ghiasi describe a form of attack in which a malicious process repeatedly flushes the trace cache of a SMT by executing self modifying code. Because the trace cache is shared among all the processes, the flushing degrades the performance of all threads [16]. Hasan et al. study DoS attacks based on power density [17].

The above techniques try to address the DoS attacks by stalling the application that is suspected of malicious behavior. This may be a working solution for SMTs but it is less attractive for CMPs, because CMPs have most of their resources unshared. A stalled core

in a CMP environment will lead in underutilization of the whole system. Furthermore, a service-targeted system may become unable to provide services even if no malicious threads are running on it [3][18]. In this scenario, the previous techniques will not detect a DoS attack, rendering the whole system practically inoperative. The problem becomes more serious when specific services require QoS guarantees.

The Statcache Model. StatCache is a technique for estimating an application's miss rate as a function of cache size based on a very sparse and easily captured fingerprint of certain performance properties [6]. The application property measured is the reuse distance of the application's memory accesses, i.e., the number of memory references between two consecutive accesses to the same cachline. Unlike stack distance, which measures the number of unique memory references between two consecutive accesses to the same cacheline, the reuse distance can easily be captured using functionality supported in today's hardware and operating systems.

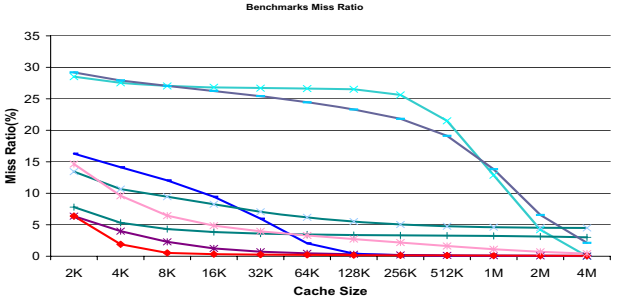

Fig. 1. StatCache results for selected SPEC2000

The reuse distances of an application's all memory accesses is most easily represented as a histogram *h(i)*, where *h(0)* is the number of references to the same cache line with no other intervening memory references, *h(1)* is the number of accesses with one intervening access, and so forth. The shape of this histogram is the performance fingerprint of an application. The shape can cheaply be approximated by randomly picking every the N^{th} access and measuring its reuse distance. Experiments have shown that sampling every $10^{7} th$ access is sufficient for long-running applications [6]. StatCache uses an application's histogram together with a simple statistical model of a cache and a simple numerical solver to derive the miss rate of the application as a function of cache size.

Figure 1 shows StatCache results for a number of SPEC2000 benchmarks for various cache sizes. This figure provides our motivation for managing the cache and prevent the hog of the cache by cache greedy applications (either they are by nature malicious or not).

As it is evident from Figure 1 many programs have flat areas in their miss-rate curves, where a change in their cache size results in virtually no change in their miss rate. Such areas can be exploited to release cache space for other programs than can benefit from more cache space (as suggested by their miss-rate curves).

3 StatShare: A Statistical Cache Model in Cat Time

In this section, we describe the basic principles of our statistical model. A necessary compromise to construct our model is to assume a fully-associative cache with random replacement.

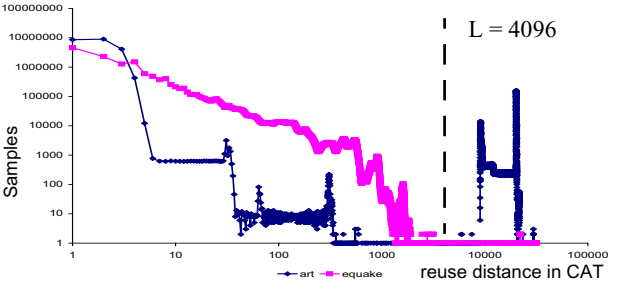

Fig. 2. CAT reuse distance histograms for art and equake (both axes log scale)

CAT Time. The reuse distance of a cacheline is measured as the number of intervening events -a notion of time- between two consecutive accesses to this cacheline. In [6], reuse distances are measured as the number of other intervening accesses. In contrast, we measure reuse distances with a different notion of "time." Our time is measured in Cache Allocation Ticks (CAT) [4], or in other words, cache replacements. The CAT clock can be advanced with two different ways: by snooping the cache replacements irrespective of the thread that causes the replacement in the shared cache (we call this a global CAT clock) or by using the replacements of the particular thread that is replaced (we call this local or per-thread counters). Our theory is independent of which clock, global or local, we use for a thread's histogram, as long as we always relate the global clock to the size of the cache and the local clock to the thread's footprint in the cache. Both ways have each own positives and negatives, but we omitted such analysis due to lack of space. For the didactic purpose of this section, we will assume global CAT as our notion of time. The importance of CAT time stems from the fact that it allows for a natural mapping of the cache statistics to the studied cache level.

CAT Reuse-Distance Histograms. The reuse distance histogram of a program measured in CAT time is denoted as: $h(i)$, $i = 0,\infty$. Figure 2 shows the histograms for two SPEC2000 programs, art and equake, sharing a 256KB cache. The histograms are collected in a time window of 200M instructions and in this case we see reuse distances of up to a few tens-of-thousands CAT.

As we can see from the histograms art shows a "binary" distribution of reuse distances, with the bulk of samples at short reuse distances, but also with a significant bulge beyond L (L=4096, the size of the cache in cachelines). This bulge signifies than many of the items that art accesses, do not "fit" in the cache and produces a significant number of misses. It is responsible for the behavior of art which hogs the cache and squeezes its companion thread to a very small footprint. In contrast, equake shows a

distribution of reuse distances that decreases slowly to the right. The meaning of this distribution, as we will show, is that equake is already in a compressed state (we cannot squeeze it further without serious damage to its miss ratio) but it can benefit from expansion to a larger footprint. In general many programs behave either like art or like equake. artlike programs are prime candidates for management-com-pression.

Basic Probability Functions. The centerpiece of the StatShare model are the *f* and *f-bar* functions. These functions give the probability of a miss (*f*) or a hit (*f-bar*) for an item in the cache with a given reuse distance. The *f*-functions coupled with the reuse-distance histograms of threads produce the rest of the information of our statistical model. The *f*-functions concern a specific replacement policy. As we have mentioned, for the didactic purposes of this section we will assume a fully-associative (FA), random replacement cache where the notion of time is given by a global CAT counter.

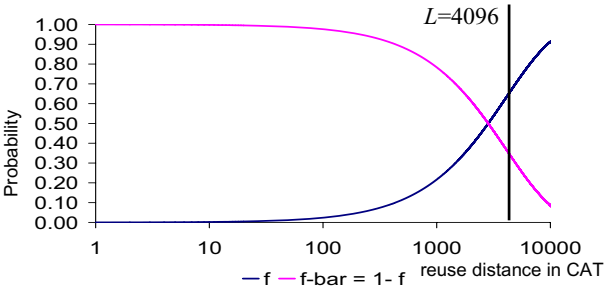

Fig. 3. *f* and *f-bar* for Random replacement in a FA cache

Under this scenario, any item in a such cache of size *L* (in cachelines) has *1/L* probability of being replaced at any miss or *(1–1/L)* probability of remaining in the cache. If an item has a CAT reuse distance of *i*, then after *i* misses (or replacements), it has a probability of remaining in the cache of $(1-1/L)^i$ and a probability of having been replaced of $1-(1-1/L)^i$. We call this miss probability function *f*, in contrast to the hit probability denoted as *f-bar:*

$$f(i) = 1-\left(1-\frac{1}{L}\right)^i \qquad \overline{f(i)} = 1-f(i) = \left(1-\frac{1}{L}\right)^i$$

Once we have a CAT reuse distance histogram for a thread it is easy to calculate its hits and misses by multiplying it with the *f-bar* and *f* functions respectively:

$$hits = \sum_{i=0}^{\infty} h(i)\times\overline{f(i)} \qquad misses = \sum_{i=0}^{\infty} h(i)\times f(i)$$

The results of these formulas agree with our simulation results with very high accuracy. However, in order to get an accurate count of misses we must take into account cold misses. Cold misses are estimated when we collect samples for the reuse

distance histograms of a thread. In short, dangling samples with no observed reuse distance correspond to cold misses [5].

4 Integrating Decay and LRU Replacement in the Model

The StatShare model gives us all the necessary theoretical information on which application we can "compress" to release space for the benefit of system as a whole. It is a good approximation for relatively large caches (greater than 64KB) of moderate to high associativity (greater than 2) with LRU replacement, such as the likely L2 or L3 caches in CMPs [8][9][10]. In this section we describe in abstract terms the StatShare model for LRU replacement and decay. We will not expand in details but give the basic information needed to support our decay-based management policies. In addition, as we will show in rest of this section, using *local* CAT counters in combination with a decay-driven replacement algorithm, we can precisely control the thread's cache footprint. This characteristic allows us not only to prevent malicious or cache-greedy applications to abuse the shared cache, but it can be used as a methodology to enforce high-level policies such as policies that try to assign custom cache-space-allocation based on external QoS.

4.1 Per-Thread Histograms

Since in this paper we are interested in indentifying unique cache-greedy applications in a shared cache, we use *per-thread* CAT clocks that are advanced by cache replacements of cachelines belonging to a specific thread, regardless of the thread that causes the replacement. In this way, the CAT clock is insensitive to the status of the whole shared cache, but dedicated to the status (cache requirements) of each individual thread. Collecting histograms of each thread using their own CAT counters creates "pure" histograms which accurately describe the cache behavior of the thread *confined to its space in the cache*. This means that the term L, which is the cache size in cachelines with the global CAT counter, is now replaced by the active ratio (in cachelines) of each thread.

LRU Replacement. With LRU replacement in a FA cache, the probabilities of a miss or a hit change with respect to those of random replacement. In short, the LRU *f*-functions are much more steep than the random *f*-functions and reach their bounds right at L. This is evident, for example, for the f function which reaches 0 just at L since nothing can remain in an LRU FA cache after seeing L replacements.

However, the shape of the *f*-functions before L is complex to derive. Because LRU, unlike random, is not memoryless, the miss and hit probabilities depend on the state of the cache which in turn implies that the f and *f-bar* functions depend on the reuse-distance histograms of the threads. In other words the behavior of LRU depends on the applications.

Assume that we have an application which has a miss rate of 1 —it has no hits. The *f-bar* (hit probability) function in this case is a step function: everything with a reuse distance larger than L is guaranteed to be a miss since an item that lives through L replacements is guaranteed to be thrown out of the cache. (However, the only

compatible histograms with this *f-bar* function have no samples inside *L* otherwise they would have hits.)

Now assume that we introduce hits into the cache by having some histogram samples inside

L. Hits affect how quickly an item with a given reuse distance moves down the LRU chain. Assume, for example, an item at a position *x* in the LRU chain (items enter in position 1 and fall out of the cache at position *L+1*). This item is pushed down in the LRU chain either from new items that enter at the top via replacements, or by hits on *older* items, located below the item in question, which bring them at the top of the LRU chain. The number of possible hits on items located after *x* is a function of the application's reuse distance histogram. The end result is that the more hits we have the faster an item with a reuse distance less than L can be evicted increasing the probability of misses in small reuse distances.

The end result is that f-function are very steep around *L* (or the equivalent active ratio), and their form at reuse distances less than L depends on the hit ratio and the thread's actual reuse-distance histogram.

Decayed *f* and *f-bar* Functions. Decay modifies the *f*-functions of the decayed applications. Once we apply decay to one of the threads that share the cache, the underlying replacement policy of the cache (LRU or Random) is changed, since decayed cachelines take precedence for eviction.

The effect of decay on LRU *f*-functions is to effectively make them step functions: the f-bar function is one almost up to the decay interval *D* and then rapidly falls to 0. The explanation is the following: if we decay a thread at a reuse distance *D*, all its items with smaller reuse distances can be hits as long as there are decayed lines available for replacements. Our modified replacement algorithm chooses a decayed line to replace if there is one available. In addition, decayed items are certain misses since they decay and are replaced. However, for performance reasons we allow hits on decayed items. This results in a discrepancy between our model and our implementation since the decayed f-functions are step function only if decayed items are misses. Thus our models are pessimistic in their assessment of performance.

Cache Management. StatShare gives us all the elements required to make informed decisions and construct high-level cache management policies. Using the StatShare outputs, we are able to understand which thread (malicious or not) can be "compressed" into less space with negligible damage and to drive accordingly the underlying replacement policy of the cache by selecting the appropriate decay intervals. This characteristic allows us not only to prevent malicious or cache-greedy applications to abuse the shared cache, but it can be used as a methodology to enforce high-level policies.

The management policy we examine in this paper is as follows:

- We collect reuse-distance histograms using local (per-thread) CAT counters.
- We assess the "threat" that each thread poses based on its reuse-distance histogram. Threads are sorted according to their DoS threat level.

- We assess the performance impact of decaying the most threatening threads using decayed LRU *f*-functions and we choose an appropriate decay interval for each. Decay intervals are restricted to a small set of *L*-fractions (e.g., *L*, *L*/2, *L*/4, etc.).

Finally, we propose as the appropriate place for using StatShare, the operating system and in particular the thread scheduler. This is because a sampling period is required at the end of which a management decision can be made. Managing the cache must be performed periodically, since threads change behavior in different program phases. In addition, threads are created, suspended, or killed dynamically and each change requires a new management decision. The sampling period must be long enough to have the time to collect useful histograms for the threads. For example, in our evaluation the sampling window is 45M instructions. Finally, Quality-of-Service guarantees that must be taken into account can be easily handled at the OS level. For example, if it is desired externally to give specific space to specific threads, this can be taken into account in the scheduler for adjusting decay intervals to satisfy such requirements.

5 Practical Implementations

In this section we show that the abstract theory can be translated into realistic run-time implementations.

Reuse-Distance Histogram Collection. At first sight, the nature of the reuse-distance histograms, which potentially span values from 0 to infinity, seems impractical for run-time collection. There are two techniques that make histogram collection not only practical but even efficient: *sampling* and *quantization*.

Sampling is a technique that was also used in StatCache [6][5]. Instead of collecting reuse distances for all accesses, we select few accesses at random, and only trace those for their reuse distance. The resulting histogram is a scaled version of the original but with the exact same statistical properties. Sampling allows for efficient run-time tracing. In our evaluation our sampling ratio is 1:1024, i.e., we select randomly one out of 1024 accesses.

The second fundamental technique that allows a practical implementation of StatShare is the *quantization* of the reuse distance histogram. Normally, it would be impractical to collect and store a histogram with potentially many thousands of buckets. However, samples with small reuse distances are statistically more significant than the ones with very large reuse distances. We use 20 buckets for quantization. In this way, the histograms can be collected in a set of 20 32-bit registers per thread, that are updated by hardware and are visible to the OS similarly to other "model-specific" registers such as performance counters. We have verified that the outputs of StatShare are practically indistinguishable using either quantized or full histograms.

Decay Implementations and Replacement Policies. Our modified replacement algorithm is very simple: we replace *any* decayed cacheline (randomly) if there is one in the set, or —if there is not— we use the underlying LRU replacement policy.

In order to hold the decay information, we use a set of registers (visible to the OS) to store the decay intervals of each thread. Non-decayed threads have an "infinite"

decay interval corresponding to the largest value of these registers. Cachelines are tagged with the CAT clock which is updated every time a hit or a replacement occurs in the corresponding cacheline. CAT tags can be made just a few bits long [4]. At the time of replacement, the CAT tag of each cacheline is subtracted from the thread CAT clock. If the result is greater than the decay interval of the corresponding thread, the cacheline is decayed and can be chosen for replacement. This check starts at a random place in a set and proceeds until either a decayed line is found or the entire set has been checked. In our methodology, the only decision we make is which decay intervals to use for the various threads.

6 Evaluation

For our simulations we have modified an SMT simulator [12] to model a CMP architecture with 2 to 4 cores. Each core is a modest 2-way out-of-order superscalar. The memory hierarchy consists of private L1s —instruction and data— and a shared, 8-way set-associative, 64B-line L2 cache. The memory latency is 250 cycles. Our intention is to isolate the data accesses behavior of applications, hence we use a relatively large instruction L1 (1MB) to preclude instruction misses from polluting the L2.

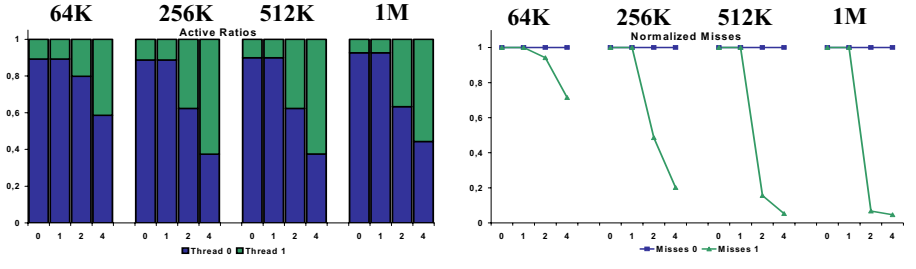

Fig. 4. Tramp vs. Gzip: active ratios and miss ratios for various decay intervals

We use a subset of the most memory intensive SPEC2000 benchmarks for our evaluation: art, gzip, equake, mcf and parser. To emulate the impact of a malicious thread, we write our own malicious program —named tramp— which is designed to be a greedy consumer of the L2. The tramp program scans continuously a very large memory array (bigger than the L2) accessing one byte out of 64 bytes (the size of the L2 block size). In every iteration, a read and a write operation are performed. With this way, the best case miss ratio of the tramp program can be equal to 50%.

To understand the behavior of decay in relation with StatShare's outputs, we have simulated a set of co-scheduled applications where one or two of them are decayed. The workload consists of 2 and 4 threads. In some sets, the tramp program has the role of the greedy application, while in some others the same role is taken by the two most memory intensive benchmarks of the SPEC2000 suite —art and mcf. Although our methodology allows any decay interval to be chosen in order to manage a thread,

we have constrained the choice of decay intervals to be binary fractions of the corresponding cache size.

All our simulations are for 200M instructions per thread. We simulate after skipping 1B instructions for art and gzip, 2B for mcf, parser, and vpr, and 3B for equake. After skip we warm up the caches for 50M instructions. Management decisions are taken every 45M instructions. In the rest of this Section we discuss results for five representative cases.

tramp—gzip. In this example art shares the cache with gzip. Figure 4 shows the active ratios and the miss ratios of the two threads for the four caches we consider, and for four decay intervals (decay is applied to tramp). Every set of bars corresponds to a specific cache size (noted on top of the set). The x-axis shows the decay intervals. The first bar (tagged with a 0 label), for each set, stands for infinite decay interval, while the values 1, 2, and 4 correspond to *L*, *L*/2, and *L*/4 respectively (*L* is the cache size measured in cachelines).

As we can see from Figure 4, our methodology successfully manages to equally divide the cache between the two threads. With an *L*/4 decay interval both applications have almost the 50% of the cache in all cache sizes. The value of our cache management technique can be seen not only for the active ratios, but for the miss ratios too (miss ratios are normalized to the non-decayed case). tramp is already experiencing a high miss ratio, so compressing it will not significantly impact its performance (as it can be seen from the graph). On the other hand, gzip is the kind of application (as it is shown by the statcache curves —Figure 1), that can benefit from its space expansion and reduce its miss ratio. The more space it gets the more hits it generates. In the 64K cache, gzip starts (at non decayed state) with 87% miss ratio and ends up (*L*/4 decay interval) with 62% miss ratio resulting in a normalized reduction of almost 30%. In the 1MB case, the benefit is more pronounced. gzip starts with 21% miss rate and ends up with a miss ratio less than 1%. This corresponds to a normalized reduction of 96%. In all cases, the miss ratio of tramp is always constant at 50%.

tramp—equake. In this case we examine tramp with another SPEC2000 program — equake. Figure 5 shows the active ratios and the miss ratios for the four cache sizes and for the 4 decay intervals (infinite, *L*, *L*/2, *L*/4).

As Figure 5 indicates tramp begins (before decay) by clearly "hogging" the cache having more than 90% of the cache in the non-decayed state (same results as in the previous example). Once it is decayed, it releases space for the benefit of equake. However, in contrast to gzip, equake cannot exploit its increased space except in the case of the 64K cache. This is evident also from the StatCache curves. Giving more space to equake produces very few additional hits.

mcf—parser. Our third example is mcf co-scheduled with parser. mcf is one of the two most memory intensive programs of the SPEC2000 suite (the other is art).

mcf is chosen for decay since it decays better than parser and occupies the most space in the cache (Figure 6). mcf's decay benefits parser with up to a maximum reduction of miss ratio of 23.5% for the 64K, 47% for the 256K, 34% for the 512K, and 25% for the 1M cache. mcf experiences a slightly increase of 3% in the miss ratio only in the case of 1M cache.

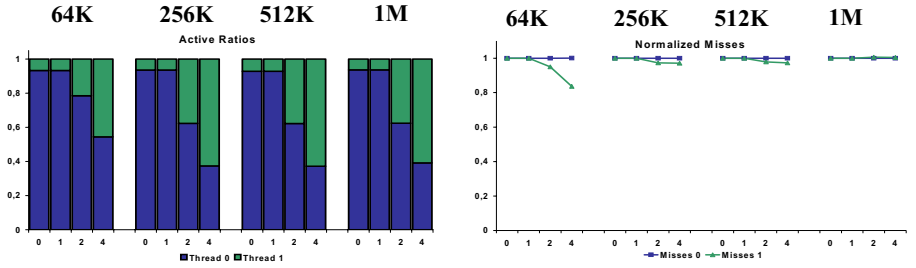

Fig. 5. Tramp vs. equake: active ratios and miss ratios for various decay intervals

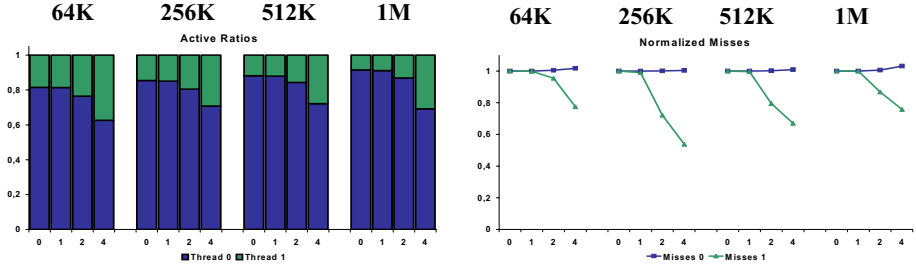

Fig. 6. Mcf vs. parser: active ratios and miss ratios for various decay intervals

tramp—gzip—parser—vpr. In this example, we evaluate our methodology when the L2 cache is shared among 4 threads —tramp, gzip, parser, and vpr. Figure 7 shows the active ratios and the miss ratios in this case.

The interesting observation that can be made from Figure 7 is that tramp must be decayed "harder" in order to see significant changes in its active ratio. Thus, we expand the decay intervals up to *L*/16 (our management algorithm always picks tramp as the decayed application). In the 2M case and for decay interval equal to *L*/16, tramp's miss ratio is increased by 2%, while its cache footprint has been decreased by a factor of 2.9 compared to the non decayed case. The released space by tramp benefits the other 3 applications. gzip increases its space by 1.3x, parser by 1.7x, and vpr by 1.4x. These expansions lead to a decrease in miss ratio of 10% for gzip, 2% for parser and 18% for vpr.

tramp—art—equake—gzip. Finally, we give a 4-thread example where decay is applied to two applications —tramp and art— since they both pose significant threat for DoS and can be significantly compressed. This two-thread decay management decision works very well since, when only tramp is decayed, its released space is occupied directly by art. art's aggressive behavior does not let the other two threads benefit from tramp's compression. On the other hand, even though art increases its cache footprint, its miss ratio does not show considerable improvements. Figure 8 presents the active ratios and miss ratios for this example. The first bar of every set corresponds to the non-decayed case (none of the applications are decayed). In the rest of the bars, tramp has a constant decay interval equal to *L*/16, while art's decay intervals are shown in the x-axis (*L*, *L*/2, *L*/4, *L*/8, *L*/16).

As we can see from Figure 8, equake and gzip benefit from art's and tramp's compression. In the 1M cache, equake increases its space by 4x and gzip by 2.5x. However, equake, in contrast to gzip, cannot exploit its increased space leading to a meagre 2% decrease (improve-ment) in its miss ratio while gzip experiences an impressive 43% decrease. The results are analogous for the other cache sizes with a big difference in art's behavior in the 2M cache. As we can see from Figure 1, art is no longer in its flat area, so if we try to compress it, we will destroy its performance, as it is evident from Figure 8 (2M case). art is not a good candidate for decay in this case.

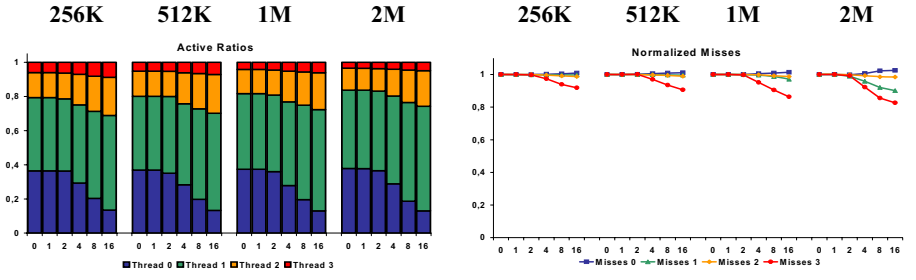

Fig. 7. Tramp-gzip-parser-vpr: active ratios and miss ratios for various decay intervals

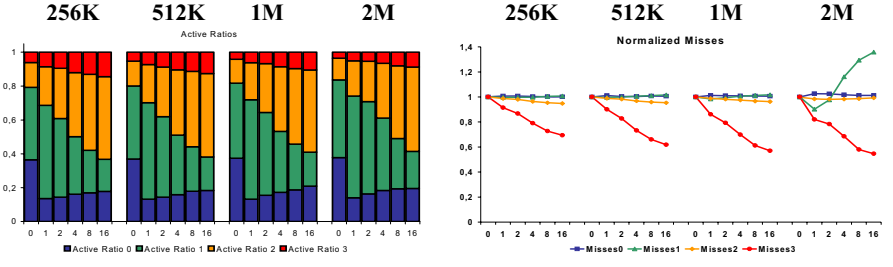

Fig. 8. Tramp-art-equake-gzip: active ratios and miss ratios for various decay intervals

7 Conclusions

In this paper, we demonstrate a new management methodology for shared caches in CMP systems, that utilizes statistical run-time information of the application behavior in order to deal with Denial-of-Service attacks. Our methodology does not need to distinguish between malicious programs and "greedy" but not-by-nature malicious programs, since these two categories behave similarly in terms of reuse distance histograms. This leads us to a more generalized approach, where dealing with DoS attacks is similar to enforcing QoS constraints or sharing the cache in a fair way.

The proposed methodology is evaluated using a detailed CMP simulator running the most memory intensive SPEC2000 applications and a "tramp" program which is designed to be an excellent consumer of the shared cache. Our results indicate that our attack-resistant cache management methodology makes it possible to identify which application (malicious or not) can be "compressed" into less cache space with

negligible damage and modify accordingly —in run-time— the underlying replacement algorithm of the cache using decay. Our results show significant benefits across the board with minimal damage for the managed threads.

References

[1] G. E. Suh, S. Devadas, and L. Rudolph. “A new memory monitoring scheme for memory-aware scheduling and partitioning” High-Performance Computer Architecture HPCA'02, 2002.

[2] S. Kim, D. Chandra and Y. Solihin. “Fair cache sharing and partitioning in a chip multiprocessor architecture” Parallel Architectures and Compilation Techniques, PACT'04, 2004.

[3] D. Chandra, F. Guo, S. Kim and Y. Solihin. “Predicting inter-thread cache contention on a chip multi-processor architecture” High-Performance Computer Architecture HPCA'05, 2005.

[4] M. Karlsson and E. Hagersten. “Timestamp-Based Selective Cache Allocation” In High Performance Memory Systems, edited by H. Hadimiouglu, et al., Springer-Verlag, 2003.

[5] E. Berg, H. Zeffer, and E. Hagersten. “A Statistical Multiprocessor Cache Model” International Symposium on Performance Analysis of Systems and Software (ISPASS-2006), USA, 2006.

[6] E. Berg and E. Hagersten. “Fast Data-Locality Profiling of Native Execution” ACM SIGMETRICS 2005, Canada, 2005.

[7] S. Kaxiras, Z. Hu, M. Martonosi. “Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power” International Symposium on Computer Architecture ISCA’28, 2001.

[8] P. Kongetira, K. Aingaran, and K. Olukutun. “Niagara: A 32-Way Multithreaded SPARC Processor” In IEEE Micro, 2005.

[9] K. Krewell. “Power5 Tops on Bandwidth.” In Microprocessor Report, 2003.

[10] K. Krewell. “Double Your Opterons; Double Your Fun.” In Microprocessor Report, 2004.

[11] J. Hennessy and D. Patterson. “Computer Architecture: a Quantitative Approach.” Morgan-Kaufmann Publishers, Inc., 2nd edition, 1996.

[12] R Goncalves, E Ayguade, M Valero and P Navaux “A Simulator for SMT Architectures: Evaluating Instruction Cache Topologies” 12th Symposium on Computer Architecture and High Performance, (SBAC-PAD 2000), 2000.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L.Traiger. “Evaluation techniques for storage hierarchies” IBM Systems Journal, 1970.

[14] CNN. ‘Immense’ network assault takes down Yahoo, 2000. Available at http://www.cnn.com/2000/TECH/computing/02/08/yahoo.assault.idg/index.html.

[15] Netscape. Leading Web sites under attack, 2000. Available at http://technews.net-scape.com/news/0-1007-200-1545348.html.

[16] D. Grunwald and S. Ghiasi. “Microarchitectural denial of service: insuring microarchitectural fairness” International Symposium on Microarchitecture MICRO-35, 2002.

[17] J. Hasan, A. Jalote, T. N. Vijaykumar, and C. E. Brodley. “Heat Stroke: Power-Density-Based Denial of Service in SMT” High Performance Computer Architecture HPCA'05, 2005.

[18] Techtarget.com. Technology terms: Denial of service. Available at http://http://whatis.techtarget.com /definition/0,289893,sid9 gci213591,00.html.

[19] P. Soderquist and M. Leeser. “Optimizing the Data Cache Performance of a Software MPEG-2 Video Decoder” In ACM Multimedia 97 - Electronic Proceedings, 1997.