Modeling Cache Sharing on Chip Multiprocessor Architectures

Pavlos Petoumenos,[1] Georgios Keramidas,[1] Håkan Zeffer,[2] Stefanos Kaxiras,[1] Erik Hagersten[2]

[1] *Department of Electrical and Computer Engineering, University of Patras, Greece*
{ppetoumenos, keramidas, kaxiras}@ee.upatras.gr

[2] *Department of Information Technology, Uppsala University, Sweden*
{zeffer, eh}@it.uu.se

Abstract — As CMPs are emerging as the dominant architecture for a wide range of platforms (from embedded systems and game consoles, to PCs, and to servers) the need to manage on-chip resources, such as shared caches, becomes a necessity. In this paper we propose a new statistical model of a CMP shared cache which not only describes cache sharing but also its management via a novel fine-grain mechanism. Our model, called StatShare, accurately describes the behavior of the sharing threads using run-time information (reuse-distance information for memory accesses) and helps us understand how effectively each thread uses its space. The mechanism to manage the cache at the cache-line granularity is inspired by Cache Decay, but contains important differences. Decayed cache-lines are not turned-off to save leakage but are rather "available for replacement." Decay modifies the underlying replacement policy (random, LRU) to control sharing but in a very flexible and non-strict way which makes it superior to strict cache partitioning schemes (both fine and coarse grained). The statistical model allows us to assess a thread's cache behavior under decay. Detailed CMP simulations show that: i) StatShare accurately predicts the thread behavior in a shared cache, ii) managing sharing via decay (in combination with the StatShare run time information) can be used to enforce external QoS requirements or various high-level fairness policies.

1. Introduction

Processor designers are fast moving towards multiple cores on a chip to achieve new levels of performance. Most newly released CPUs are chip multiprocessors and all processor vendors offer at least one CPU model of this design. The goal is to hide long memory latencies as much as possible and maximize performance from multiple threads under strict power budgets. CMPs are becoming the dominant architecture for many server class machines [8,9]. For reasons of efficiency and economy, sharing of some chip resources is a necessity. Shared resources in CMPs typically include Level 2 caches and this creates a need for skilful management policies, since L2 caches are a critical element in the performance of all modern computers. It is essential for the future management of cache resources, as well thread migration strategies, to fully understand how threads sharing a common cache interact with each other.

To model and understand cache sharing we have built a new theoretical framework that accurately and concisely describes the application 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 from a single set of reuse-distance measurements. 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 (Cache Allocation Ticks, CAT [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 online analysis feeding, for example, a dynamic resource scheduler —in contrast, StatCache is an off-line model. This paper shows, with detailed CMP simulation and co-scheduled applications, that StatShare accurately predicts both miss ratios and cache footprints online.

We also demonstrate how StatShare can be used to manage a shared cache. We model and evaluate a control mechanism based on Cache Decay, initially proposed for leakage reduction in uniprocessor caches [7]. The original Cache Decay uses cycle timers in each cache line to turn power off to the cache line after a period of inactivity (called "decay interval"). By tuning this decay interval, one can restrict the "active ratio" of an application (i.e., its "live" lines) to a small percentage of the cache without significantly impacting performance. Decay discards dead lines that are unlikely to be accessed in the future.

Similarly, in a shared cache we use decay to control the active ratio of applications so we can enforce high-level policies, for example QoS policies [13], cache fairness policies [2,3,14], or simply optimizations for performance [1,15]. However, our proposed mechanism introduces important differences to decay. A decayed cacheline is simply *available for replacement* rather than turned-off for leakage. Thus, hits on decayed lines are allowed (since they are still in the cache). Secondly, the decay interval is measured not in cycles but in CAT time. This gives *CAT Decay* some interesting properties that can be used in conjunction with our model to determine the number of decay-induced misses and the space that is released by decayed applications. An important aspect of our work is that decay is integrated into the StatShare model allowing us to predict and control the effects of decay.

Decay has important advantages used as a cache-sharing management mechanism. It allows complete freedom over which cache lines an application can have in the cache. It does not restrict sharing in either cache ways or sets —it only controls an application's active ratio (the number of its live lines). Furthermore, decay only controls the *average* active ratio, allowing its instantaneous variations to track miss-frequency fluctuations. These properties make decay superior to strict cache partitioning schemes (either fine-grained that allocate a specific number of cache lines to an application or coarse-grained that divide the cache into larger chunks, per application). However, a comparison of cache management schemes is *not* the focus of this paper and we will not expand further into this due to lack of space. Instead, here we focus on the (online) statistical model which characterizes the sharing behavior of applications and *drives* the managing mechanism. We note, that our modelling methodology can easily be adapted to describe less sophisticated managing schemes (e.g., cache partitioning schemes).

Structure of this paper. Section 2 reviews the StatCache model and related work. Section 3 presents the StatShare model, while Section 4 the notion of Spacetime, and Section 5 decay in StatShare. Section 6 discusses implementations and Section 7 presents our results. Section 8 offers our conclusions

2. StatCache and Related Work

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 cacheline. 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 OS [6].

The reuse-distances of an application's memory accesses are most easily represented as a histogram *h(i)*, where *h(*0*)* is the number of references to the same cacheline 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 be approximated cheaply by randomly picking every N^{th} access and measuring its reuse-distance. Sampling every 10^7*th* accesses is sufficient for long-running applications [6]. StatCache uses an application's histogram together with a simple statistical model of a cache and a numerical solver to derive the miss rate of the application as a function of cache size.

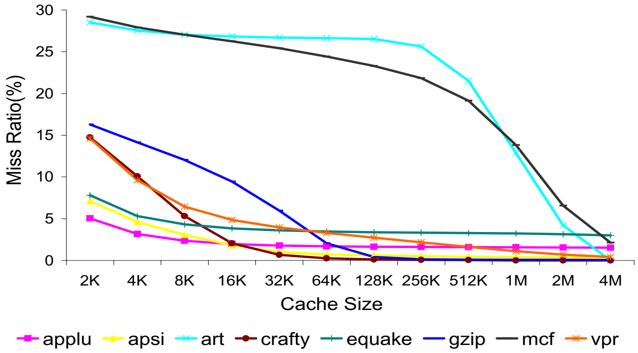

Figure 1. StatCache results for selected SPEC2000.

Figure 1 shows StatCache results for a number of SPEC2000 benchmarks for various cache sizes (reuse-distance histograms were collected from runs on the Intel/Linux platform). This figure provides our motivation for managing the cache. 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 that can benefit from more cache space.

Cache Managing Schemes and Fair Sharing. The issue of cache fairness has been initially proposed 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 (initialization, rollback and re-partitioning) for shared-cache partitioning. Their algorithms are based on stack-distance counters [12] but do not restrict the cache replacement algorithm to LRU. Their partitioning mechanism 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.

Chandra et al. [3] extend this work with three performance models that predict the impact of cache sharing on co-scheduled threads. The input to the models are the L2 stack distances of isolated applications and the output is the number of extra L2 misses for each thread due to cache sharing. Lie et al. evaluate the trade-offs between private and unified L2s and address interleaved shared L2 designs, noting their individual benefits and drawbacks [14]. Yeh and Reiman tried to address the problem of fairness in a physically distributed NUCA L2 cache design [13]. 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 in [2], but their focus is in performance and not fairness. Finally, Snavely et.al. showed that the performance of a SMT processor is heavily dependent on the applications that are running on it [15]. The authors propose alternative thread scheduling schemes based on the behavior of the corresponding application mix.

Previous modeling work examines the behavior of applications in isolation and predicts the result of having more than one application share the same cache. In contrast, we provide a low-overhead, online, theoretical model that captures the sharing behavior of the threads while they are actively sharing the cache. Our model can be used to provide informed management decisions to drive a flexible mechanism (for example, CAT Decay) in order to enforce external high-level policies stemming for either QoS guarantees, policies to increase fairness, or simply performance optimizations.

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 a useful model is to assume a fully-associative cache with random replacement. StatShare directly describes capacity misses and can estimate cold misses but fails to take into account conflict misses. It is a good approximation for relatively large caches (greater than 64KB) of moderate to high associativity (greater or equal to 4), such as the likely L2 or L3 caches in CMPs [8,9]. This is because in terms of conflict misses an associativity of 4 or 8 in a relatively large cache is not very far from full associativity [10]. For the rest of the paper we use the fully-associative model with great success to describe 8-way set-associative caches in our simulations.

3.1 CAT Time

The reuse-distance of a (cacheline) address is measured as the number of intervening events —a notion of time— between two consecutive accesses to this address. In StatCache [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 importance of CAT time stems from its ability to relate time (events) to space (cache size). For example, using our theory we can show that the expected lifetime of any item in the cache is *L* CAT ticks where *L* is the cache size in cachelines.

3.2 CAT Reuse-Distance Histograms

The reuse-distance histogram of a program measured in CAT time is denoted as: *h(i)*, *i* = 0, ∞. Figure 2 shows the histograms for two SPEC 2000 programs, art and equake sharing a 256K 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.

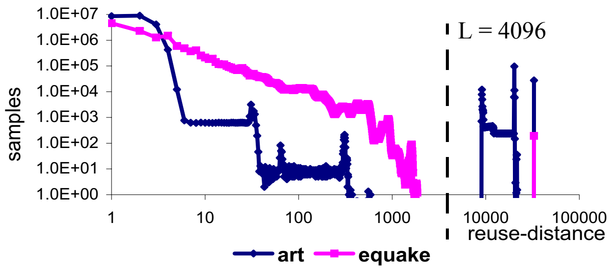

Figure 2. CAT reuse-distance histograms, *h(i)*, for art and equake (both axes log scale)

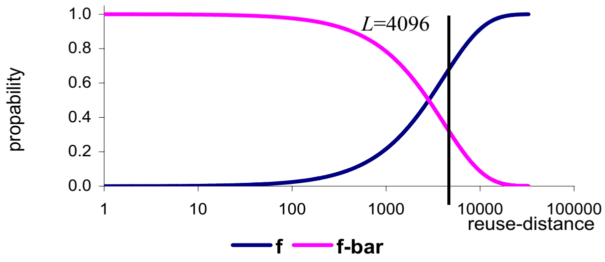

Figure 3. *f* and $\overline{f}$ for Random Replacement in FA cache

As we see in Figure 2 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 that 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 *could* benefit from expansion to a larger footprint. In general many programs behave either like art or like equake. art-like programs are prime candidates for management either via decay or by other means.

3.3 Basic Probability Functions

The centerpiece of our theory are the *f* and $\overline{f}$ functions (Figure 3). These functions give the probability of a miss (*f*) or a hit ($\overline{f}$) 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. In this paper we will concentrate on random replacement since it is memoryless thus allowing us to derive analytical expressions for the *f*-functions. We have also examined LRU *f*-functions but since LRU replacement depends on the cache state, analytical expressions are too complex to introduce in this paper. However, we note that given the appropriate LRU f-functions our methodology is exactly the same as with random replacement.

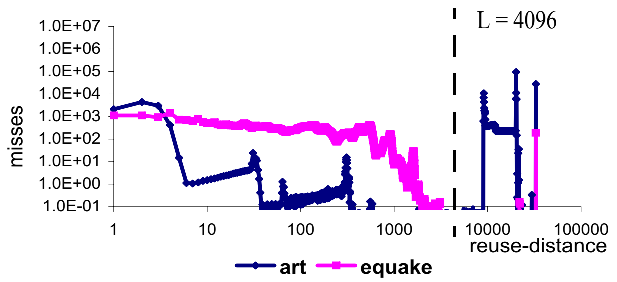

Figure 4. Multiplying *f(i)* with *h(i)* gives us misses

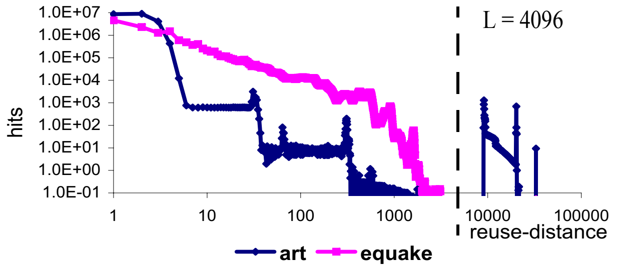

Figure 5. Multiplying $\overline{f(i)}$ with *h(i)* gives us hits

Any item in a fully-associative, random-replacement 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. Thus, after *i* misses, an item 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 define the miss probability function *f* and the hit probability function $\overline{f}$, measuring the probability of a second access to a sample, of a CAT reuse-distance *i*, being a miss or a hit respectively, as:

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

3.4 Hits and Misses

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 $\overline{f}$ and *f* functions respectively (Figure 4 and Figure 5):

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

The results of these formulae 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. Spacetime

Spacetime is the space occupied by a cacheline multiplied by the duration of the cacheline's lifetime in the cache (from the miss that brought it in to the miss that evicts it). Of course time is measured in CAT in our case. We are considering the contents of the cache not at any particular cycle but over a period of time. We define *active ratio* as the thread's average cache occupancy over a period of time. The active ratio can also be expressed as the spacetime of the thread divided by the total spacetime (cache size × total CAT time). Consequently, the ratio of two threads' active ratios is the ratio of the threads' spacetimes.

Spacetime is derived from the thread histograms and the *f*-functions. Moreover, our methodology allows us to calculate the spacetime that is associated with hits and misses. Spacetime associated with hits is *useful space*; spacetime associated with misses is *wasted space*. The ratio of a thread's useful to wasted space, its ***useful ratio***, is a metric of how well a thread exploits its cache space. In our example, art hogs the cache with an extremely unfavorable useful ratio. In contrast equake, pressured by art, shows a good useful-to-wasted ratio. In the rest of this section we show how we compute spacetime from the thread histograms and how decay affects it.

4.1 Spacetime of a Thread

The lifetime expectancy of a sample, which has a reuse distance *i*, is equal to the sum of the probabilities of being alive during each of the *i* CAT ticks that will occur until it will be requested again. Since the probability of being alive during the k^{th} tick is the probability of not being replaced during the previous *k-1* replacements, which is $\overline{f(k-1)}$, the lifetime expectancy of the sample is:

$$l(i) = \sum_{k=1}^{i} \overline{f(k-1)} = \sum_{k=1}^{i} \left(1 - \frac{1}{L}\right)^{k-1} = L \times f(i)$$

Thus, the spacetime occupied by a thread is equal to the *number of misses produced by a thread multiplied by the size of the cache in cache lines:*

$$S = \sum_{i=0}^{\infty} l(i) \times h(i) = \sum_{i=0}^{\infty} L \times f(i) \times h(i) = L \times misses$$

Consequently, the spacetime occupied by all the threads is equal to the cache size (measured in cachelines) multiplied by the total number of the misses. This definition is consistent with the previous definition of the total spacetime, since the number of misses is equal to the total CAT ticks (if we do not take into account cold misses). More importantly, the above equation directly associates the cache footprint of a thread with its misses. *The space that is occupied by any thread in a shared cache can be calculated if we divide the thread misses with the total cache misses.*

4.2 Spacetime Associated with Hits

A hit means that the corresponding thread occupies the cache for the full reuse-distance of this hit. Thus the spacetime for hits is:

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

Hit spacetime is *useful*: we pay the spacetime cost but we are benefiting by a hit. However, when the cost-to-benefit ratio

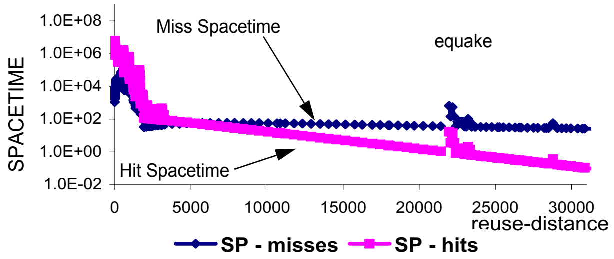

Figure 6. Spacetime associated with hits and misses for equake (x-axis is not in logscale, y-axis in logscale).

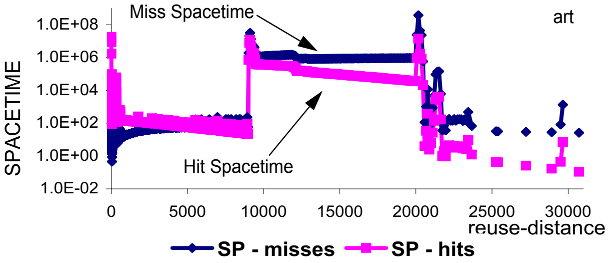

Figure 7. Spacetime associated with hits and misses for art (x-axis is not in logscale, y-axis in logscale).

is excessive it might be beneficial for the system as a whole to let this hit go and reclaim its spacetime. This is what decay achieves with "decay-induced misses" (hits that are converted to misses due to decay).

4.3 Spacetime Associated with Misses

The spacetime associated with misses can be easily calculated as the difference between the spacetime of the thread minus the spacetime associated with hits:[1]

$$S_{misses} = \sum_{i=0}^{\infty} L \times f(i) \times h(i) - \sum_{i=0}^{\infty} h(i) \times \overline{f(i)} \Rightarrow$$

$$S_{misses} = \sum_{i=0}^{\infty} [(L+i) \times f(i) - i] \times h(i)$$

Spacetime associated with misses is ***wasted*** spacetime: this spacetime is occupied for no benefit. Unfortunately, it is a necessary "evil" for the existence of useful spacetime.

To validate our spacetime theory we computed spacetime according to the above formulae using thread histograms collected in simulations. The results give us an accurate estimate for the ratios of the threads' footprints (as reported by our simulations). Figure 6 and Figure 7 compare the spacetime for hits and misses for equake and art respectively. In both graphs the x-axis is plotted normally (not in log scale) to clearly show the contribution of various reuse-distances to hit and miss spacetime. The total spacetime for the application is the area under each curve. It is obvious from the graphs that art has significantly more wasted space (miss spacetime) than useful space and this is due to large reuse-distances. equake also has wasted space, but its total spacetime is much less than art's.

4.4 Effects of Decay on Spacetime

Assume now that we decay a thread with a decay interval of D CAT. Decay "releases" some spacetime from the thread in the form of decayed lines. Such lines are available for replacement by other threads. Decaying one thread with a decay interval D and assuming that all decayed lines are misses (pessimistically since decayed cachelines can be reclaimed by the owning thread), the hits beyond the decay interval become decay-induced misses (DIM):

$$DIM = \sum_{i=D}^{\infty} h(i) \times \overline{f(i)}$$

One method to control decay can then be based on the ratio of decay induced misses to the total misses of the application:

$$DIMratio = \left(\sum_{i=D}^{\infty} h(i) \times \overline{f(i)}\right) \Big/ \left(\sum_{i=0}^{\infty} h(i) \times f(i)\right)$$

Applying the above formulae to our example with art and equake sharing a cache, we conclude that:

- art can be decayed in a 256K cache without consequences since its *DIMratio* is close to 0. Decay intervals down to a few CAT do not change art's miss ratio, in accordance to Figure 1. However, in larger caches, art starts to fit in the cache. In such cases it becomes increasingly difficult to decay art.
- equake cannot be decayed in small caches since any decay interval (less than L) immediately results in an increase in its misses. This means that equake is already at a compressed state and can only benefit from expansion.

Both miss and hit spacetime are released by decay. The former is wasted space that can be potentially exploited by other threads to increase their useful space. The latter, though, *was* useful space for the decayed thread and its release can be harmful for the whole system if other threads fail to convert it again to useful space. In our evaluation we show how cache management affects the overall useful space in the cache.

5. Integrating Decay in the Model

The goal of this work is to construct a theoretical model in order to provide a practical framework for the management of a shared cache using decay. The theoretical model gives us the necessary information about the "wasted" and the "useful" space of each thread sharing the cache. Decay changes the

[1] We can derive the exact same expression by computing the expected lifetime $l_{miss}(i)$ of an item *conditionally on being a miss*. This is derived from the expected value of a random variable following a modified geometric *pmf*—a set of Bernoulli trials where success is replacement, *1/L*, and failure is not being selected, *1-1/L*):

$$l_{miss}(i) = \sum_{k=0}^{i} k \times \frac{1}{L} \times \left(1 - \frac{1}{L}\right)^{k-1} = (L+i) \times f(i) - i$$

$$S_{misses} = \sum_{i=0}^{\infty} l_{miss}(i) \times h(i) = \sum_{i=0}^{\infty} [(L+i) \times f(i) - i] \times h(i)$$

replacement policy so that decayed cachelines take precedence for eviction. However, once we apply decay in the cache, the cornerstone assumption of StatShare, the random replacement (and the probability of $1/L$ of an item being chosen for replacement) is no longer valid. In this section we integrate decay into StatShare by showing how it modifies the *f*-functions of both the decayed and the non-decayed applications.

5.1 Decayed *f*-functions

To understand the effect of decay we divide replacements into "Murphy" and managed replacements. Murphy replacements (after the notorious Murphy law) escape our efforts to manage them. A more rigorous definition is that a Murphy replacement happens if there is *no decayed cacheline available for eviction*. In contrast, a managed replacement is when we select a decayed cacheline for replacement. Our evaluation shows, that even if the average number of decayed lines is significant, the number of available decayed lines at any single point in time varies considerably, many times being 0. Even when we apply strong decay in the cache, we can reduce Murphy replacements to very few but rarely we can manage to bring them to zero. Of course, this phenomenon is more pronounced in real set-associative caches than in the FA caches of our model.

The probability of a Murphy replacement, which we call μ, varies considerably over time depending on the availability of decayed items. However, for simplicity we use an average value of μ within a time window. The reason for using this average is that we can readily measure it as the number of Murphy replacements over all replacements:

$$\mu = \frac{MurphyReplacements}{misses}$$

For the *non-decayed* application, the probability of an item surviving a particular replacement is the probability of finding a decayed line to replace —which is $(1-\mu)$— plus the probability of *not* finding any decayed line to replace —which is μ— *and at the same time* not being selected for replacement —which is $(1-1/L)$. Note that, when we cannot find a decayed line to replace, the *f*-functions behave as the normal *f*-functions before decay. The probability of surviving i replacements is, therefore, the new $\overline{f}$ function which we denote as $\overline{f_{nd}(i)}$. The f_{nd} function is simply $(1 - \overline{f_{nd}(i)})$:

$$\overline{f_{nd}(i)} = \left((1-\mu)+\mu\times\left(1-\frac{1}{L}\right)\right)^i = \left(1-\frac{\mu}{L}\right)^i$$

The $\overline{f_{nd}(i)}$ function *holds only for live lines* since the probability $(1-1/L)$ of not being selected for replacement is for live lines. For the non-decayed applications $\overline{f_{nd}(i)}$ is simply a **scaled version of their original** $\overline{f}$. It is as if the non-decayed application(s) operate in a larger cache ($1/\mu$ larger to be precise).

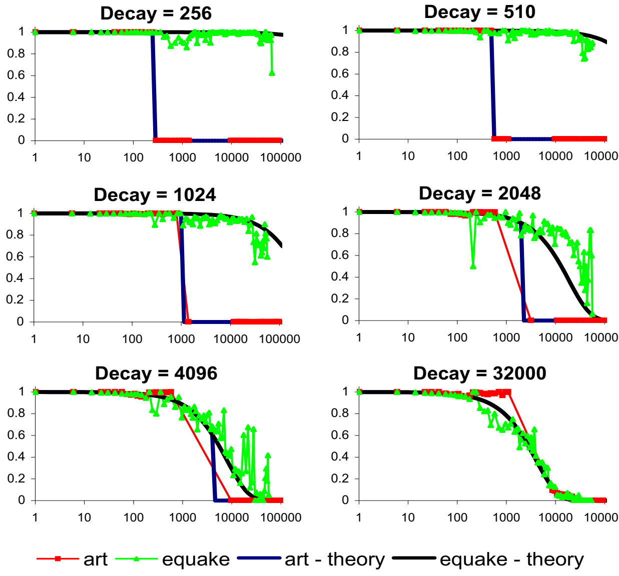

Figure 8. Measured and computed $\overline{f}$ functions for art and equake in a medium-pressured 256K cache. x-axis is reuse-distance, y-axis is hit probability. Art is decayed for various decay intervals (256-8K CAT). The $\overline{f}$ function for art approaches a step function with small decay intervals but returns to its original form with very large decay intervals. For equake, with enough decayed lines, its $\overline{f}$ function is scaled by the Murphy misses but also returns to its original state as decay weakens.

The behavior of the *decayed* application's new $\overline{f}$ function —denoted $\overline{f_d(i)}$ — is similar. Lines with reuse distances less than or equal to D behave identically to the live lines of the non-decayed thread (μ/L probability of being replaced at each CAT tick), but lines with reuse distances greater than D, either can be replaced before D ticks have occurred, or are decayed after D replacements, in which case we treat them as dead. Therefore, we conclude that the $\overline{f_d(i)}$ is equal to the $\overline{f_{nd}(i)}$ for reuse distances less than or equal to D and zero for reuse distances greater than D:

$$\overline{f_d(i)} = \begin{cases} \left(1-\frac{\mu}{L}\right)^i, i \le D \\ 0, i > D \end{cases}$$

The decayed application, similarly to the non-decayed applications, "feels" that it is operating in a much larger cache, but this only concerns its live cachelines (reuse-distance less than D). In contrast, decayed cachelines are considered dead and all accesses to these lines are treated as misses (albeit *zero-latency misses* in practice).

Our approximations for the f_d-functions agree to a great extent with experimental data given the assumptions we have made (e.g., measurement of Murphy replacements within a time window) and the differences of the model and our simulations (e.g., 8-way set-associative instead of fully-associative). The approximations fail only when we have a significant supply of decayed items. In this case the assumption of the step to 0 (at *D*) is not accurate.

We use simulation to gather reuse-distance histograms and hit histograms. Dividing these histograms gives us the measured $\overline{f}$ functions. Figure 8 shows how well StatShare's calculated $\overline{f}$ functions agree with measured $\overline{f}$ functions for art and equake sharing a "medium-pressure" 256K cache. art is decayed and causes both applications to assume new f-functions. In the graphs we plot the measured $\overline{f}$ functions; the smooth-dark lines are StatShare's $\overline{f}$ functions. They are calculated based on a probability of Murphy misses of μ, which is derived from the measured Murphy misses in the simulations. Again, inaccuracies are due to differences between the model and simulation and because very small numbers of Murphy misses, which imply large numbers of decayed items, lead to larger errors.

5.2 Spacetime with decay

Using the same reasoning as in Section 4.2 and Section 4.3 the spacetime of the non-decayed applications in the presence of decay becomes (where *L'* equals *L*/μ):

$$S = \sum_{i=0}^{\infty} L' \times h(i) \times f_d(i)$$

$$S_{hits} = \sum_{i=0}^{\infty} i \times h(i) \times \overline{f_d(i)}$$

$$S_{misses} = \sum_{i=0}^{\infty} [(L' + i) \times f_d(i) - i] \times h(i)$$

The above expressions do not hold for decayed applications. In this case, any line that has survived *D* replacements is decayed and leads to a miss as far as our model is concerned. So the lifetime expectancy for those lines becomes:

$$l(i) = \sum_{k=1}^{i} \left(1 - \frac{\mu}{L}\right)^{k-1} = L' \times f_d(i), \quad i \le D$$

$$l(i) = \sum_{k=1}^{D} \left(1 - \frac{\mu}{L}\right)^{k-1} + \sum_{k=D+1}^{\infty} 0 = l(D), \quad i > D$$

and the corresponding spacetimes are:

$$S = \sum_{i=0}^{D} L' \times h(i) \times f_d(i) + \sum_{i=D+1}^{\infty} L' \times h(i) \times f_d(D)$$

$$S_{hits} = \sum_{i=0}^{D} i \times h(i) \times \overline{f_d(i)}$$

$$S_{misses} = \sum_{i=0}^{D} [(L' + i) \times f_d(i) - i] \times h(i) + \sum_{i=D+1}^{\infty} L' \times h(i) \times f_d(D)$$

Although, the above expressions are simple approximations, in practice we found them to be quite useful. Better approximations are possible but require the estimation of the amount of decayed items.

6. Practical Implementations

In this section we discuss necessary modifications for realistic run-time implementations of StatShare.

6.1 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 is also used in StatCache [5]. Instead of collecting reuse-distances for all accesses, we select a few accesses at random, and only trace these 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. We use a small set of watchpoint registers where we insert addresses for tracing. Sampling itself allows us to keep only a small number of watchpoint registers and still be able to measure very large reuse-distances. Each register is tagged with the current CAT clock. Whenever an access to the cache matches a watchpoint register, the difference of the CAT clock to the CAT tag of the register is the reuse distance of this particular sample. In our evaluation our sampling ratio is $1{:}2^{10}$, i.e., we select randomly one out of 2^{10} accesses.

The second fundamental technique that allows a practical implementation of StatShare is *quantization* of the reuse-distance histograms. 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.

6.2 Decay Implementation 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 none— we use the underlying replacement algorithm. With random replacement as the underlying replacement algorithm, our scheme is very simple to implement.

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 timestamp for the point in time they were last touched. CAT tags can be made just a few bits long [4]. Upon a replacement, the CAT tag of each cacheline is subtracted from the 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. In our methodology, the only decision we make is which decay interval to use for each thread.

6.3 Policy Enforcement at the OS Level

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 [16]. 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 example, in our evaluation the sampling window is 45M instructions. Finally, QoS 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, the scheduler can satisfy such requirements by adjusting the decay intervals.

7. Evaluation

7.1 Experimental Setup

For our simulations we have modified an SMT simulator [11] to model a CMP architecture with 2 to 4 cores. Each core is a modest 4-way out-of-order superscalar. In this paper, we concentrate on understanding the effects of cache management in terms of cache metrics. We will not expand into processor performance metrics, such as IPC, since they can have a damping or an amplification effect on cache performance (depending on processor cores) and are irrelevant for validating StatShare. The memory hierarchy consists of private L1-instruction (1MB, 4-way, set-associative, 64B-line, 3-cycle) and data caches (16KB, 4-way set-associative, 64B-line, 3-cycle), and a shared, 8-way set-associative, 64B-line, L2 whose size ranges from 64KB to 1MB. The memory latency is 300 cycles. Our intention is to isolate and study only the data access behavior of applications, hence we use a relatively large instruction L1 to preclude instruction misses from polluting the L2.

Even though we use the most memory intensive SPEC2000 benchmarks, their cache requirements are still quite low. A 2MB cache easily fits the most aggressive SPECint benchmarks as shown in Figure 1. Thus, L2 caches are scaled to small sizes (64K to 1MB) to enhance the effects of sharing. We use a subset of memory-intensive SPEC2000 benchmarks: art, gzip, equake, vpr, mcf and parser. We have also examined other benchmarks but compute-intensive benchmarks with low cache requirements do not exhibit interesting behavior in our case. Workload mixes consisting of such benchmarks cannot benefit much from management since in most cases they fit nicely in the cache. This is also pointed out in related work [1,2,3,13].

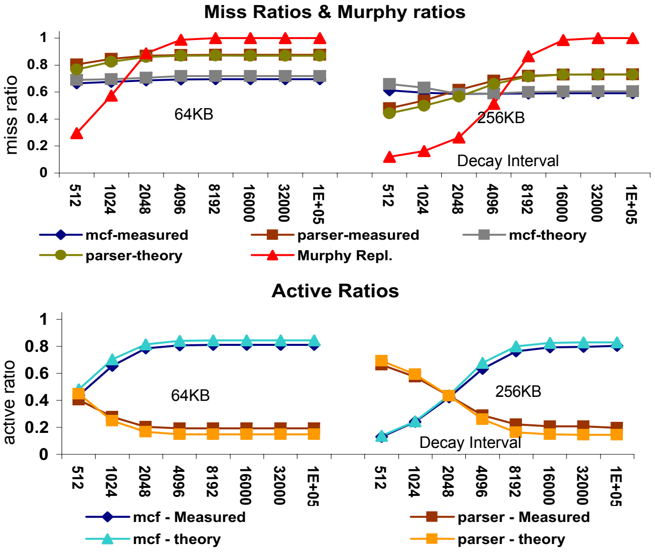

Figure 9. mcf–parser: miss rates, Murphy misses, and active ratios (simulation vs. theory)

For all of our experiments we skip the initialization part of each benchmark. Specifically, we skip 1B instructions for art and gzip, 2B for mcf, parser, and vpr, and 3B for equake. Afterwards, we warm the caches for 50M instructions and start collecting data for 200M instructions of detailed cycle-accurate simulation. StatShare-based management decisions are taken every 45M instructions.

7.2 Results

The purpose of this section is twofold. First, to show the effectiveness of the StatShare model in estimating the sharing behavior of the threads (including CAT decay effects). Second, to show that StatShare-generated information can be used to make informed decisions and to construct high-level policies (e.g., for QoS, fair sharing, etc.). Using the StatShare output, we are able to understand the cache behavior of co-scheduled threads and drive accordingly the underlying replacement policy of the cache by selecting appropriate decay intervals.

To show these aspects we selected a set of five application mixes. Each application mix consists of two or four co-scheduled SPEC2000 benchmarks. Two different cache sizes were simulated for each mix: 64K and 256K for the two-thread mixes, and 512K and 1M for the four-thread mixes. For both cases, these cache sizes were chosen to represent very high pressure (contention) and medium-pressure respectively, with the benchmarks we examine. Subsequent sections discuss the results for the five representative cases.

Model validation: mcf–parser. In this first proof-of-concept example, mcf shares the cache with parser. mcf is one of the two most memory intensive programs of the SPEC2000 suite as it evident from the StatCache miss-rate curves (Figure 1). Figure 9 shows StatShare generated curves when compared to simulation results. The upper part of the figure shows miss ratios while the lower part shows active ratios for

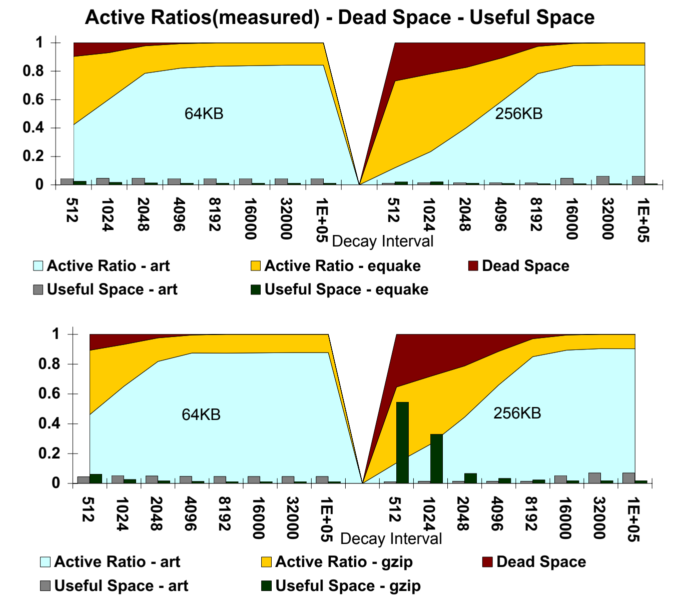

Figure 10. art–equake (up) and art–gzip (down): active ratios (measured), dead space and useful ratio

various decay intervals and for two cache sizes, 64K and 256K. Although our methodology allows any decay interval to be chosen to manage a thread, we have constrained decay intervals to a set of 8 (from 512 to 100K CAT), for simplicity. The 100K CAT decay interval corresponds to the *unmanaged* mode, since practically nothing decays.

In this example, we apply decay to mcf while keeping parser un-decayed. The lower part of Figure 9 shows StatShare's effectiveness in managing the cache space occupied by the two applications. mcf's active ratio is more than 80% for both 64k and 256k caches when no decay is applied. However, once decayed, mcf gives up cache space. With 512 CAT decay interval, mcf ends up with an active ratio of 48% in the 64K cache and an active ratio of 13% in the 256K cache respectively. Moreover, parser is able to use the freed-up cache space and effectively increases its active ratio by a factor 2.1 (4.13) in the 64K (256K) cache.

The upper part of Figure 9 shows how changes in active ratio affect the miss ratios of the two applications sharing the cache. In the 256K cache there is a significant improvement for parser (42% reduction in miss ratio) while decay intervals below 4096 CAT hurt mcf leading to a slight increase of 3% in its miss ratio. parser is the kind of application which can take full advantage of its space expansion. For both cache sizes, an increase in its active ratio automatically results in significant improvements in its miss ratio. On the other hand, mcf is insensitive to decay. This is because mcf is in its "flat" area (Figure 1) hence, can be compressed without significant consequences. However, in larger caches, mcf starts to fit in the cache; it becomes increasingly difficult to decay it, without increasing its miss ratio. Finally, the small increase in mcf's miss ratio attests to the fact that mcf —in contrast to art— does not have a "binary" distribution of reuse-distances. mcf's reuse-distance histogram is not shown because of space limitations, but our next example shows that we can decay art with minimal cost even in small caches. (This can be seen also in Figure 1, where art has a flatter profile than mcf for small cache sizes.)

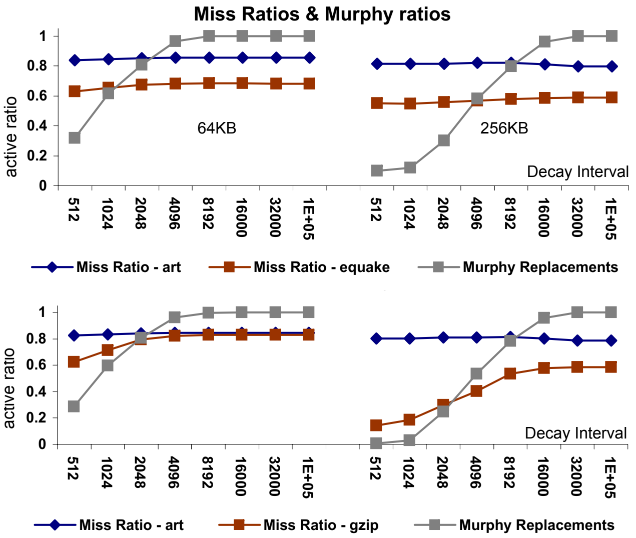

Figure 11. art–equake (up) and art–gzip (down): miss rates (measured) and Murphy misses

StatShare is able to track the simulator results with great success. As can be seen in Figure 9, the maximum divergence between the StatShare predicted active ratios and the active ratios produced by the simulator is 6%. The average difference is 3.1% for mcf and 4% for parser. The difference between the simulator and the StatShare predictions is even smaller in terms of miss ratio, where the average divergence is 2% for mcf and 1.6% for parser (maximum difference is 4.6% and 4.9% respectively).

Reasoning about cache management: art–equake and art–gzip. The previous example shows that StatShare can predict the active ratios and the miss ratios of co-scheduled applications with high accuracy even when decay changes the replacement policy of the shared cache. The purpose of this example is to show that using StatShare we are able to make informed decisions for high level policies such as policies that try to minimize the cache miss ratio by selecting appropriate decay intervals.

The example consists of two cases: art sharing the cache with equake and art sharing the cache with gzip. Figure 10 shows the measured active ratios as well as the dead space (shaded areas) of the cache for the two workload mixes, for the two cache sizes, and for the 8 decay intervals. For the 100K decay intervals (unmanaged cases), art's active ratio is 90% for both workload mixes and for both cache sizes. Once art is decayed, it releases space for the benefit of equake and gzip respectively.

However, equake cannot exploit its increased space. This is evident from equake's reuse-distance histogram shown in

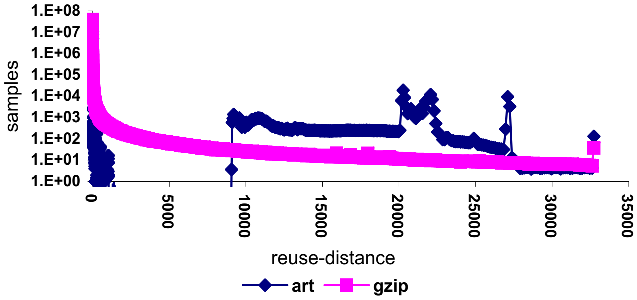

Figure 12. art—gzip reuse-distance histograms. gzip can benefit from scaled *f*-functions.

Figure 2: replacing the normal *f*-functions with the f_d-functions (scaled by the Murphy miss ratio shown in Figure 11's upper graph) produces few additional hits! In short equake does not have the right kind of histogram to benefit from its scaled *f*-function. Figure 10 also depicts the useful ratio, i.e., the total useful spacetime divided by the total spacetime, for art (dark bars) and equake (light bars). The useful ratio of both applications remains almost the same for all the decay intervals which means that: i) art's decay does not influence its useful space, i.e., decay discards useless lines, ii) even though equake increases its active ratio (3.1x for the 64K and 3.87x for the 256K cache), its useful spacetime does not increase. The upper graph of Figure 11 confirms this since we see no improvements in the miss ratios of both programs.

In contrast to equake, gzip has the kind of histogram to benefit from *f*-scaling. Figure 12 shows the reuse-distance histograms collected for art and gzip. Because gzip has a considerable number of samples at large reuse-distances, the more space it gets the more hits it generates (contrast Figure 12 with Figure 2).

The lower graphs in Figure 11 show how the miss ratios and the Murphy ratio change with the decay intervals for the different caches. There is an improvement of up to 25% in the number of misses for gzip in the 64K case. The situation is even better in the 256K case where gzip reduces its misses up to 76%. The lower part of Figure 10 shows the corresponding active ratios and useful ratios. Here, the (calculated) useful ratio shows a significant increase and tracks well the improvements in miss ratios in both the 64K and the 256K caches.

Model validation (4-threads): mcf–parser–equake–vpr. In this example we evaluate our model when the cache is shared among 4 threads —mcf, parser, equake, and vpr. Figure 13 shows measured versus theoretical results for the active ratios (upper graph) and the miss ratios (lower graph) while Figure 14 depicts the errors between simulation and theory in active ratios and miss ratios respectively. All graphs comprise 4 groups of lines corresponding to two different cache sizes (512K and 1M) and two different decay intervals (4K and 6K CAT decay for the 512K cache and 4K and 12K for the 1M cache). The cache sizes and the decay options are noted on top of the groups. Each group comprises 8 lines representing the measured and the theoretical results for the 4 threads of the example. Because of the very good agreement of theory and simulation, the theoretical and simulated lines for each of the applications overlap to a great extent. Each line consists of 5 different points: the first point (tagged with an "A") represents the case where all the threads are compressed with the same decay interval, while the last point (tagged with "N") depicts the unmanaged case (none of the threads is decayed). All intermediate points (tagged with "1" to "4") show results when only one of the threads is decayed with the corresponding decay interval noted on top of the group. Points tagged with "1" are for mcf, points tagged with "2" are for parser and so forth.

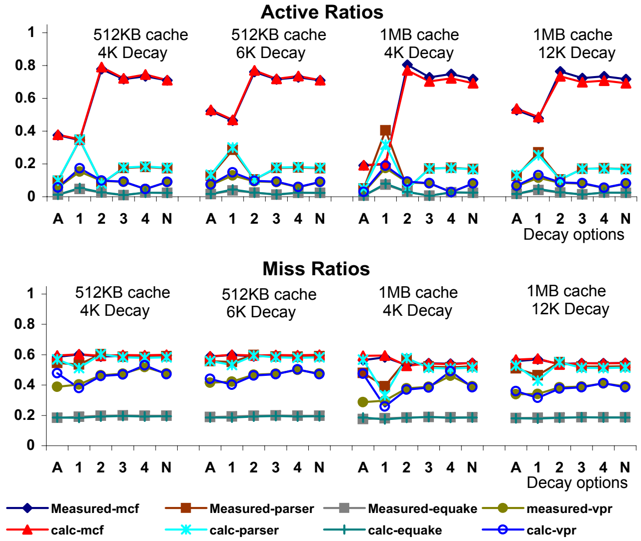

Figure 13. mcf–parser–equake–vpr: active ratios and miss ratios (simulation vs. theory)

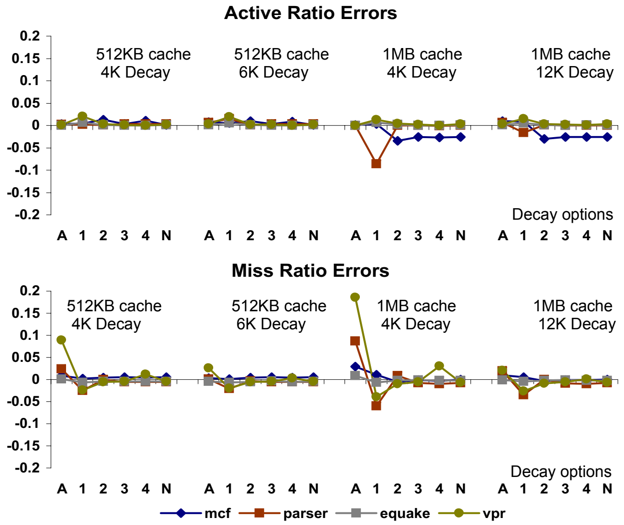

Figure 14. mcf—parser—equake—vpr: active ratio and miss ratio errors

As we see in Figure 13 and Figure 14, StatShare is able to predict the active ratios and the miss ratios of the applications

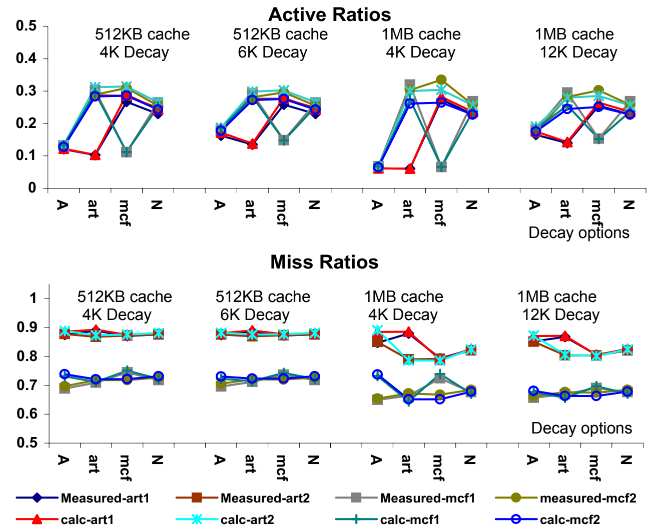

Figure 15. art–art–mcf–mcf: active ratios and miss ratios (simulation vs. theory)

with great success. Regarding the active ratios, the error in most cases is below 2% (especially in the 512K cache), while the maximum error is 8% (when mcf is decayed in the 1M cache). Miss ratio prediction follows the same trend except from the case where decay is applied to all threads. This discrepancy (of up to 18%) attests to the fact that our model considers hits on dead (decayed) cachelines as misses. Thus, when the dead space gets larger, simulation results diverge significantly from the theoretical results. This is why inaccuracies in our model are more pronounced in larger caches where the dead space (for the same CAT decay interval) is comparably more.

Model validation (4-threads, high-pressure): art–art–mcf–mcf. In this example we evaluate our statistical model in a tough scenario: using the most cache "greedy" benchmarks of the SPEC2000 suite. Since art and mcf are by far the most memory intensive benchmarks, we construct a 4-thread application mix consisting of these two benchmarks only (two instances of each). Figure 15 shows the measured versus the theoretical results for the active ratios (upper graph) and the miss ratios (lower graph) for two cache sizes, 512K and 1M. Figure 16 shows the differences between simulation and StatShare.

StatShare manages to track the measured values with small error. In this mix, which was selected to be an aggressive consumer of the shared cache, errors between simulation and theory are quite smaller than the previous example for both the active ratio and the miss ratio. The difference is more pronounced in the miss ratio error. In the all-decay case, the maximum error is 8% (compared to 18% in the previous example), while the average error is less than 1% in the 512K cache and 2.2% in the 1M cache. Smaller errors are due to considerably less dead space in this "high-pressure" case. Thus, the "hits-on-dead-lines" effect —not modelled in StatShare— is not so pronounced as in the previous example.

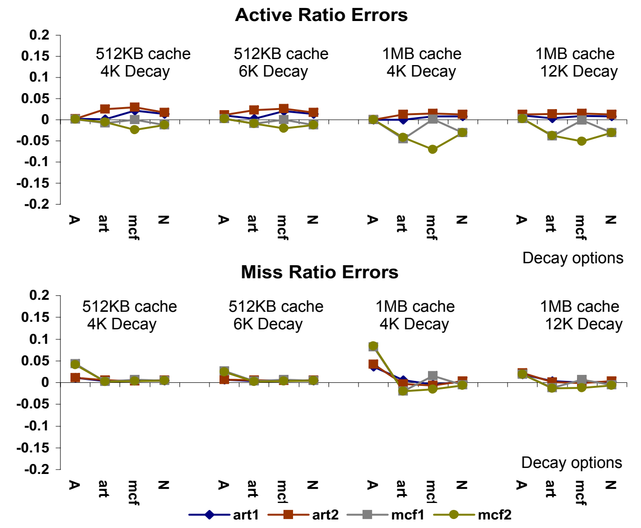

Figure 16. art–art–mcf–mcf: active ratio and miss ratio errors

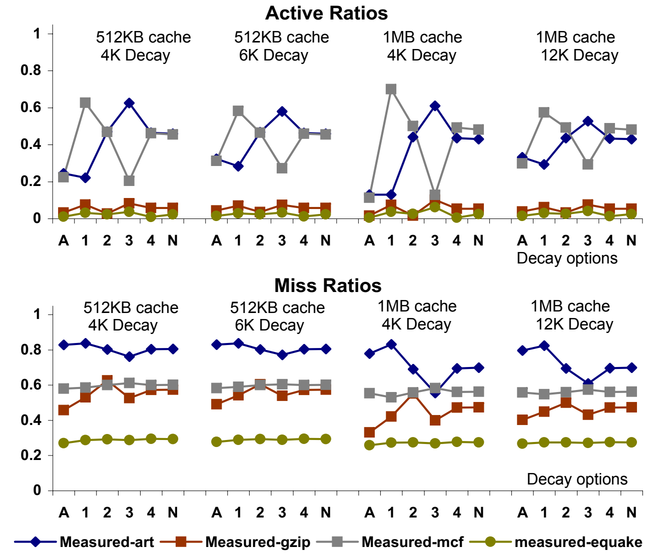

Figure 17. art–gzip–mcf–equake: active ratios and miss ratios (simulation results)

Reasoning about cache management (4-threads): art–gzip–mcf–equake. In our final example, art and mcf are co-scheduled with gzip and equake. Figure 17 shows the active ratios and the miss ratios for 512K and 1M caches while the decay interval is varied. We use 4K and 6K decay intervals for the 512K cache and 4K and 12K decay intervals 1M cache. The cache sizes and the decay options are noted on top of each group of lines. The first point of each line represents the all-decay case, the last point shows the unmanaged case, while the four intermediate points correspond to one decayed application at a time.

As Figure 17 shows, equake's miss ratio remains practically the same irrespectively of the decay intervals applied (or the absence of decay) and the cache size. As we have already explained in the art-equake example, equake cannot exploit increased space. In contrast, while art can be decayed in the 512K cache with almost no performance loss, this is not true for the 1M cache. art starts to fit in large caches, so in such cases it becomes increasingly difficult to decay it. Moreover, art benefits from mcf's decay resulting in a decrease in its miss ratio by 14% compared to the unmanaged case (in the 1M cache). Finally, gzip, in all cases, takes full advantage of the extra space that it gets. The more space it gets the more hits it generates. In the 512K cache, there is a significant improvement for gzip when all the applications are decayed (21% reduction in gzip's miss rate). The situation is much better in the 1M cache, where gzip reduces its miss ratio by 31%.

8. Conclusions

This paper presents and evaluates a statistical cache model for shared caches, called StatShare, and a fine-grained mechanism to manage such caches. The input to the statistical model is sparsely sampled data (reuse distances measured in CAT) collected during runtime, on-line, in the shared cache environment. From these data the model is capable of estimating the miss rates and the cache footprints of applications both when they are unmanaged and when they are managed in various ways. This makes it possible for software such as the OS scheduler and/or dedicated hardware solutions to control the cache based on different policies.

Our control mechanism is based on cache decay, but with some important differences. Instead of turning-off a cacheline we modify the underlying replacement policy to mark these lines available for replacement. The fine-grained management mechanism makes it possible to free-up space in the cache without introducing new misses. It is characterized by its flexibility which makes it preferable to strict cache partitioning schemes.

The statistical model and the decay management are verified with detailed CMP simulation running memory intensive SPEC applications. We find the model to be very accurate in predicting both the miss rates and the space occupied by the different applications making it possible to characterize applications at run-time and apply high-level management decisions accordingly. StatShare provides the necessary information on how to manage an application to reduce or expand its cache footprint in a shared cache, predicting both penalties and benefits associated with such management decisions. Our future work includes using StatShare to implement high-level policies for QoS, fair cache sharing, or performance optimization.

9. References

[1] G. E. Suh, S. Devadas, L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," In Proc. of the *Eighth International Symposium on High-Performance Computer Architecture (HPCA–8)*, 2002.

[2] S. Kim, D. Chandra, Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," *13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, 2004.

[3] D. Chandra, F. Guo, S. Kim, Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," In Proc. of the *11th International Symposium on High-Performance Computer Architecture (HPCA–11)*, 2005.

[4] M. Karlsson, E. Hagersten, "Timestamp-Based Selective Cache Allocation," In *High Performance Memory Systems*, 2003.

[5] E. Berg, H. Zeffer, E. Hagersten, "A Statistical Multiprocessor Cache Model," In Proc. of the *2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2006)*, 2006.

[6] E. Berg, E. Hagersten, "Fast Data-Locality Profiling of Native Execution," In Proc. of the *2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2005.

[7] S. Kaxiras, Z. Hu, M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," *28th Annual International Symposium on Computer Architecture (ISCA–28)*, 2001.

[8] P. Kongetira, K. Aingaran, K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro, vol. 25, no. 2*, Mar/Apr, 2005.

[9] K. Krewell, "Power5 Tops on Bandwidth," In *Microprocessor Report*, 12/22/03-02, 2003.

[10] J. L. Hennessy, D. A. Patterson, "Computer Architecture: a Quantitative Approach," *Morgan-Kaufmann Publishers, Inc.*, 2nd edition, 1996.

[11] R. Goncalves, E. Ayguade, M. Valero, P. Navaux, "A Simulator for SMT Architectures: Evaluating Instruction Cache Topologies," *XII Symposium on Computer Architecture and High Performance*, 2000.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Journal of Research and Development*, 1970

[13] T. Y. Yeh, G. Reinman, "Fast and Fair: Data-Stream Quality of Service," In Proc. of *the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems,* 2005.

[14] C. Liu, A. Sivasubramaniam, M. Kandemir, "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs," In Proc. of the *10th International Symposium on High Performance Computer Architecture (HPCA–10)*, 2004.

[15] A. Snavely, D. M. Tullsen, G. M. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor," In Proc. of the *2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2002.

[16] T. Sherwood, S. Sair, B. Calder, "Phase Tracking and Prediction," *30th Annual International Symposium on Computer Architecture (ISCA–30)*, 2003.