STATSHARE: A Statistical Model for Managing Cache Sharing via Decay

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

[1]University of Patras, Greece

[2]Uppsala University, Sweden

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 becomes a necessity. In this paper we examine the management of on-chip shared caches. Our paper offers two major contributions. First, we propose a new statistical model of a shared cache that can be fed with run-time information: reuse-distance information for thread accesses. Our model, called StatShare, accurately describes the behavior of the sharing threads, helps us understand which threads can be "compressed" into less space without perceptible damage, and how effectively each thread uses its space. Second, we propose a mechanism to manage the cache at a very fine level, at the cache-line granularity. Our mechanism is inspired by cache decay, but with some important differences. Decayed cache-lines are not turned-off to save leakage but rather they are "available for replacement." Decay modifies the underlying replacement policy (random, LRU), to enforce our high-level policy decisions but in a very flexible and non-strict way. The statistical model allows us to assess a thread's cache behavior under decay. Using this information we can then apply high-level policies such as policies that try to minimize the global miss-rate, or maximize the "usefulness" of the cache real estate, or even custom space-allocation policies according to external QoS needs. To evaluate our approach we have implemented StatShare in a CMP simulator. Our results show that: i) managing sharing via decay outperforms coarse-grain partitioning schemes, ii) StatShare can yield run-time information to allow high-level policies to control decay.

1. INTRODUCTION

Even though shared caches have been widely used in many shared-memory multiprocessors to date, the effects of constructive and destructive cache sharing has not received much attention until the recent focus on chip-multiprocessor architectures (CMP). CMPs introduce a trade-off between the number of cores and the amount of cache space on a processor chip. This may lead to a decrease in the fraction of the chip space devoted to caches. Furthermore, the number of threads sharing the scarce cache resource can be quite high in a CMP. An example of this is the Sun T1 processor (code-named Niagara) [9] where 32 threads share a 3MB L2 cache. It will become 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 applications' interplay in shared caches. Our cache model, named StatShare, is derived from the StatCache statistical cache model [7], 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 on-line analysis feeding a dynamic resource scheduler. Examples of such are thread migration management and management of the cache sharing for optimal performance. StatShare can accurately predict miss rate and spacetime cost as a function of the active cache ratio used by an application. The paper demonstrates the accuracy between results derived from the StatShare model by comparing them to simulation results.

We also demonstrate how online StatShare results can be used as inputs to a resource scheduler. We model and evaluate a cache resource sharing strategy based on Cache Decay, originally proposed for leakage reduction [8]. Our proposal introduces important differences to decay. In our case 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 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 cachelines. 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. Thus, an on-line management policy can select appropriate decay intervals for co-scheduled applications to achieve its goals.

We demonstrate StatShare and decay-managed sharing using simulation. Our results show that: i) decay outperforms a previously proposed dynamic partitioning scheme [2], and ii) StatShare's output allows us to reason about the effects of decay and provide —at run-time— the necessary information for high-level policies to control decay.

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 high-level cache-sharing management policies and Section 7 implementations. Section 8 presents our results and Section 9 our conclusions.

2. STATCACHE AND RELATED WORK

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 [7]. 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

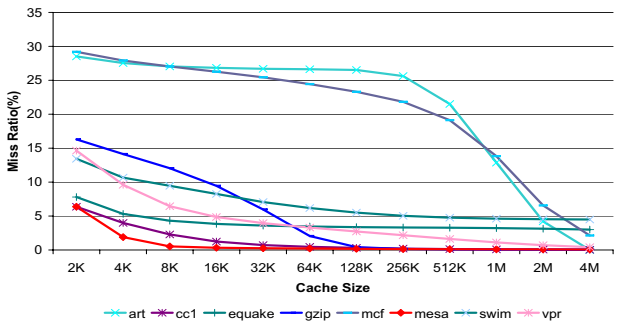

Figure 1. StatCache results for selected SPEC2000.

be captured using functionality supported in today's hardware and operating systems.

The reuse-distances of an application's memory accesses is 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 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 [7]. 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 (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 this figure 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 the benefit of other programs.

In addition to StatCache, three other recent papers are quite relevant to our approach [1][2][3]. Kim et al. [2] 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 [13] 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], Chandra 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.

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. In practice, 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 [9][10][11]. There are two aspects for why it is a good approximation: first in terms of conflict misses an associativity of 4 or 8 in a relatively large cache is not very far from full associativity. Second, a realization in the course of this work changed our naive perception of Random replacement which is close to LRU in highly associative caches (e.g., 8-way) [12]. Random replacement is not as "random" as we tend to think: the longer an item remains in the cache, the greater is the chance of it having been replaced. In many cases items that are in the cache for a long time are in their dead time and simply await eviction [8]. Thus, there is a probabilistic skew in Random for replacing the same items as LRU. Due to lack of space, we will not further quantify this phenomenon, suffice to say that for the rest of the paper we use the model with great success to describe the realistic caches (8-way set-associative) 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 [7] 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, as we will show in the next section, the expected lifetime of an 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: $h(i)$, $i = 0,\infty$. Figure 2 shows the histograms for two SPEC 2000 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. We will use this 2-thread "didactic" example throughout the presentation of our theory.

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 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

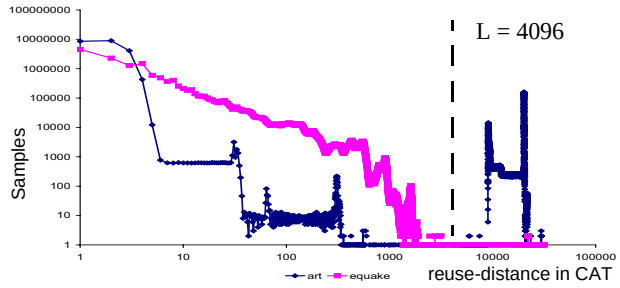

Figure 2. CAT reuse-distance histograms, *h(i)*, for art and equake (both axes log scale)

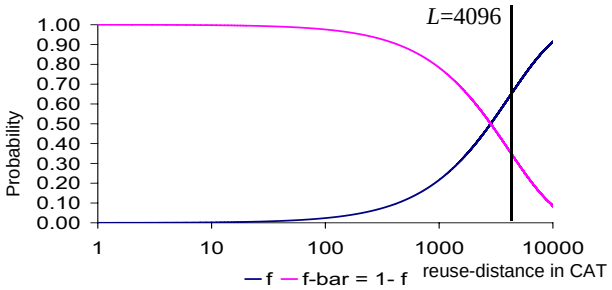

Figure 3. *f* and *f-bar* for Random replacement in a FA cache

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 *f-bar* functions (Figure 3). 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. 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.

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. If an item has a CAT reuse-distance of *i*, then after *i* misses, 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 define the miss probability function *f*, measuring the probability of a second access to a sample, of a CAT reuse-distance *i*, being a miss, as:

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

The hit probability is similarly denoted by the function *f-bar*:

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

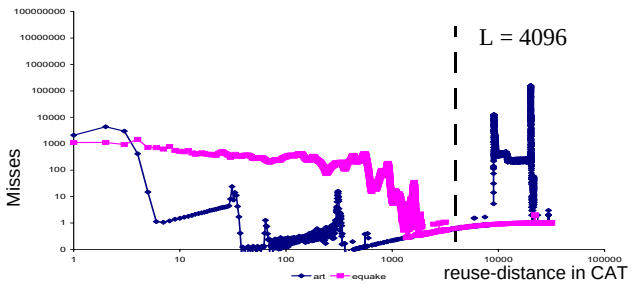

Figure 4. Multiplying *f(i)* with *h(i)* gives us misses (log-log graph)

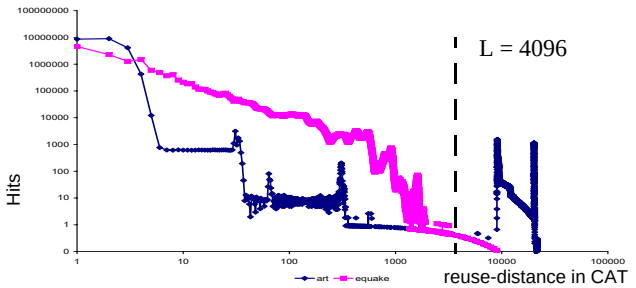

Figure 5. Multiplying *f-bar(i)* with *h(i)* gives us hits (log-log graph)

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 *f-bar* 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 [6].

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 threads' histograms and *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 and whether we can improve

on it by managing sharing. In our example, art hogs the cache with an extremely unfavorable useful-to-wasted spacetime ratio. In contrast equake, pressured by art, shows a good useful-to-wasted spacetime ratio. In the rest of this section we show how we compute spacetime from the threads' histograms, and how decay affects it.

4.1 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 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 because of decay).

4.2 Spacetime Associated with Misses

Spacetime for misses is more complex to calculate. This is because the information we have for a miss with a reuse-distance of r is that it lived less than r. Thus, we need to estimate the expected lifetime of a miss given that it has a reuse-distance of r.

The probability of an item being replaced at the i^{th} CAT is equivalent to a set of Bernoulli trials where we count the number of trials until (and including) the first "success:"

$$p_Z(i) = p \times q^{i-1} = \frac{1}{L} \times \left(1 - \frac{1}{L}\right)^{i-1}$$

(p_z *is the geometric pmf*, i is said to have geometric distribution [5]) where p is the probability of "success" (replacement) equal to the inverse of the size of the cache, $1/L$, and q is the probability of "failure" (non-replacement), equal to $1–p = 1–1/L$. Since we are interested in the spacetime occupied just before the replacement we use the modified geometric pmf p_x [5] (which is the number of trials —non-replacements— that precede a success, i.e., a replacement):

$$p_X(i) = \frac{1}{L} \times \left(1 - \frac{1}{L}\right)^{i}$$

The expected lifetime of an item with a modified geometric pmf in the cache is:

$$E = \sum_{k=0}^{\infty} k \times p_X(k) = \sum_{k=0}^{\infty} k \times \frac{1}{L} \times \left(1 - \frac{1}{L}\right)^{k} = (L-1)$$

In other words, the expected lifetime (in CAT) of any item in the cache is L*-1*. But we are interested in the expected lifetime of a miss with a reuse-distance of i. This restricts the lifetime of any miss to an upper limit of i replacements (otherwise it would not have been a miss). Thus, the life expectancy of a miss, $l(i)$, and its spacetime, S_{miss}, are:

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

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

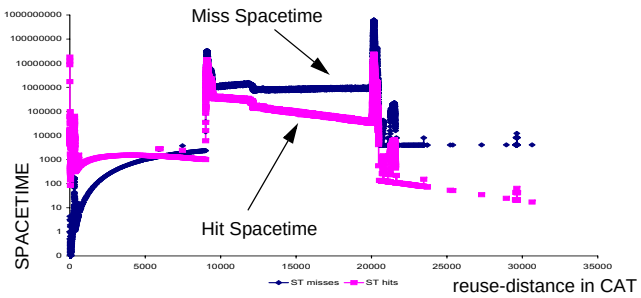

Figure 6. Spacetime associated with hits and misses for art (x-axis is not in logscale (y-axis in logscale).

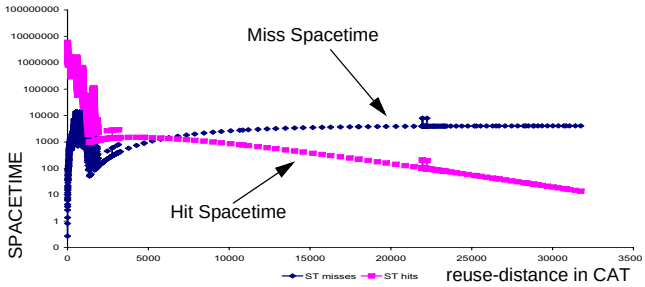

Figure 7. Spacetime associated with hits and misses for equake. (x-axis not in logscale but y-axis is)

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 art and equake respectively. In both graphs the x-axis is plotted normally (not in logarithmic 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 form the graphs that art has significantly more wasted space (miss spacetime) than useful space and this is due entirely to large reuse-distances. equake also has wasted space, however, its overall spacetime is much less than art's.

4.3 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 could 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 to guide the management of the cache using decay. The theoretical model gives us the necessary information on which application we can "compress" to release "wasted" space for the benefit of system as a whole. Decay changes the 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* and *f-bar* functions

To understand the effect of decay we divide replacements to "Murphy" and managed replacements. Murphy replacements (after the notorious Murphy laws) 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 StatShare.

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, and with the above discussion in mind, 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 *f-bar* function which we denote as f_{nd}*-bar(i)*. The f_{nd} function is simply $(1 - f_{nd}$*-bar)*.

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

The f_{nd}*-bar* 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 f_{nd}*-bar* is simply a **scaled version of their original** $\overline{f}$*-bar*. It is as if the non-decayed application(s) operate in a much larger cache ($1/\mu$ larger to be precise).

To understand the behavior of the decayed application's new *f-bar* function —denoted f_d*-bar*— let us consider the two extreme cases: i) when there are always decayed items available (μ=0), and ii) when there are none (μ=1). In the first case, any item with a reuse-distance less than D, cannot be replaced but decayed items have a probability close to one of being replaced (assuming there are just enough to drive the Murphy misses to 0). In this case, the f_d*-bar* is a step function, f_{step}*-bar*, with a value of 1 for reuse-distances up to the decay interval, and 0 afterwards. In the second case, where we never find a decayed item, the f_d*-bar* function is, naturally, the original, no-decay, *f-bar* function.

Our approximation for the f_d*-bar* for μ ranging between 0 and 1 is simple: its is a superposition of the step function and the original *f-bar* function, scaled by the probability of Murphy misses:

$$\overline{f_d(i)} = (1-\mu) \times \overline{f_{step}(i)} + \mu \times \overline{f(i)}$$

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 quickly consumed by replacements, and this is reflected in the sharp fall of its f_d*-bar* at D.

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-associativity instead of FA). The approximation is not very accurate only when we have a significant supply of decayed items. In this case the assumption of the step function is not accurate for reuse-distances greater than D, assuming that we allow hits on decayed items.

Figure 8 and Figure 9 shows how well StatShare's calculated *f-bar* functions agree with *measured f-bar* functions. We have used simulation to gather reuse-distance histograms and hit histograms. Dividing these histograms gives us the measured *f-*

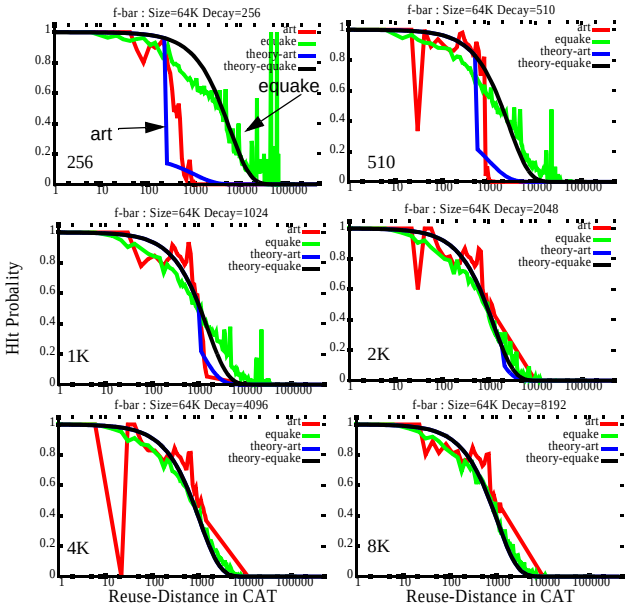

Figure 8. Measured and computed *f-bar* functions for art and equake in a high-pressure 64K cache. Art is decayed for various decay intervals (256-8K CAT). The *f-bar* function for art approaches a step function with small decay intervals but returns to its original form with very large decay intervals. For equake, with enough decayed lines, its *f-bar* function is scaled by the Murphy misses but also returns to its original state as decay weakens.

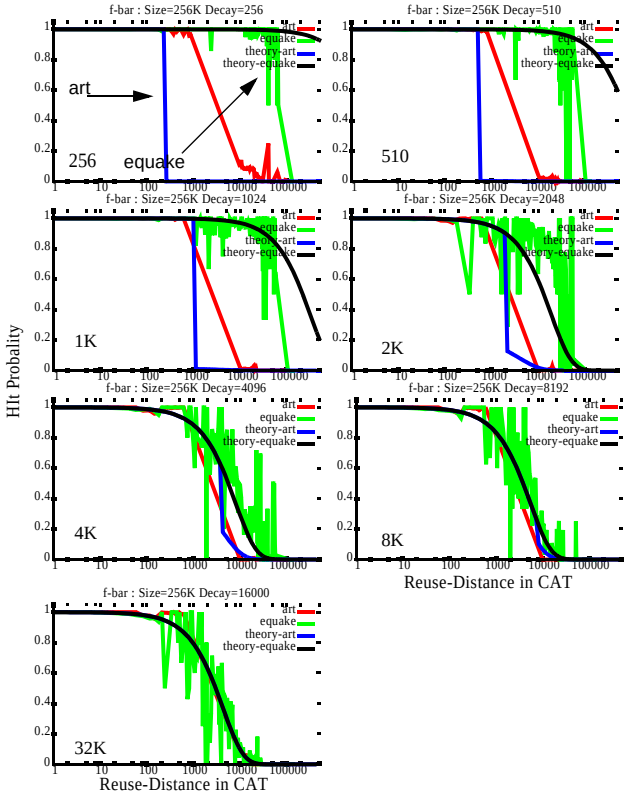

Figure 9. Measured and computed f-bar functions for art and equake in a medium-pressure 256K cache. Art is decayed for various decay intervals (256-32K CAT).

bar functions. Figure 8 shows art and equake on a very small, high-pressure, 64K cache and Figure 9 for a medium-pressure 256K cache. art is decayed and causes both applications to assume new *f*-functions. In the graphs we plot the measured *f-bar* functions; the smooth, dark, lines are StatShare's *f-bar* 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 4.1 and Section 4.2 the spacetime of the applications in the presence of decay becomes:

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

The expected lifetime of a miss is derived exactly the same as in Section 4.2 but using as the probability p of "success" — replacement— the term μ/L instead of $1/L$:

$$l(i) = \sum_{k=0}^{i} k \times \frac{\mu}{L} \times \left(1 - \frac{\mu}{L}\right)^k$$

However, the above expression for the expected lifetime only holds for items that *cannot* decay. For items that can potentially decay we must account for the increased probability of the item being replaced *after* it decays. The expected lifetime of an item that can potentially decay is:

$$l_d(i) = \sum_{k=0}^{i} k \times p(k)$$

Where $p(k)$ is the probability that the item is replaced at the k^{th} replacement. This probability changes after the item is decayed so the sum consists of two terms: the first term is the contribution to the lifetime until we reach the decay interval and is the same as the formula for the non-decayed items; the second term is the contribution to the lifetime after the item has decayed.

According to our reasoning for the f_d functions, we assume that the probability of a decayed item to be selected for replacement is (1–μ), i.e., the probability of a managed replacement. Its probability of not being selected is equal to the probability μ of the Murphy miss. Thus, the probability of a decayed item surviving a number of replacements is the probability of reaching the decay interval alive, $(1-\mu/L)^D$, and then *not* being selected for an additional $i–D$ CAT, μ^{i-D}. This yields the following expression for the expected lifetime of the misses of the decayed application:

$$l_d(i) = \sum_{k=0}^{D} k \times \frac{\mu}{L} \times \left(1 - \frac{\mu}{L}\right)^k + \sum_{k=D+1}^{i} k \times (1-\mu) \times \left(1 - \frac{\mu}{L}\right)^D \times \mu^{k-D}$$

Although, the above expression is a simple approximation, in practice we found it to be quite useful. Better approximations are possible but require the estimation of the amount of decayed items.

6. MANAGEMENT POLICIES

The value of the StatShare model in relation to decay is that it allows us to reason about decay in the cache and make informed decisions for selecting decay intervals. StatShare gives us all the elements required to construct high-level cache management policies which set the decay intervals. In this section we describe some of the possibilities for high-level policies.

Minimizing total misses. A policy whose goal is to maximize throughput by minimizing total misses for all applications, starts by determining what is the effect of decaying each application using a set of decay intervals. Murphy misses are estimated for each case and the new f_d-functions are computed. The resulting miss-ratios for the decayed and non-decayed applications are then computed from the f_d-functions and the application histograms. In the general case, with many co-scheduled applications, this is a difficult problem so additional criteria must be used to drive the selection of decay intervals.

Maximizing useful space. One such criterion is the maximization of the useful ratio of the cache. Decay intervals are selected so as to increase the total useful spacetime. The applications selected for decay must release more wasted space than useful space and the non-decayed applications must be able to exploit the wasted space by converting (some of it) into useful space.

Setting specific active ratios. Another policy that relates to QoS requirements is to give each application specific space in the cache. This is achieved by restricting applications that exceed their active ratio limit with decay and allowing others that have less than their fair share of the cache to expand. The selection of the decay intervals in this case is a repetitive process. Starting from the worst offender we select a decay interval that will bring it inside its active ratio limits according to StatShare's spacetime calculations. Once this is achieved, we move on to the next thread that exceeds its active ratio limit. Note that we cannot do much for a thread that does not fill its designated active ratio.

7. PRACTICAL IMPLEMENTATIONS

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

7.1 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 was also used in StatCache [6][7]. 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. We use a small set of watchpoint registers where we insert (cacheline) 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 1:1024, 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.

7.2 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 replacement algorithm. With random replacement as the underlying replacement algorithm, our scheme is very simple to implement.

Our implementation of decay, also differs markedly from [8]. We do not use local and global decay counters (as in [8]), since we are not interested in decaying an item at the earliest possible time. Instead, we are only interested in knowing which items are decayed at the time of a replacement. Furthermore, we need to be able to selectively decay cachelines belonging to different threads with the appropriate decay intervals. A set of registers hold the decay intervals of the threads. Non-decayed threads have an "infinite" decay interval corresponding to the largest value of these registers. Cachelines are tagged with two pieces of information:

1. with the thread ID of the thread that brought them into the cache, tagged at the time of the respective miss, and,
2. with the CAT clock, *tagged on every subsequent access* (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 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.

7.3 Policy Enforcement at the OS Level

Once we obtain a sampled, quantized, histogram we can compute at run-time, hits, misses, their spacetime, and the effect of decay on the *f*-functions. This requires a run-time system capable of performing the relevant computations. Although, it is possible to propose simple management heuristics that can be implemented in hardware —and we aim to do so in future work— for this paper we propose as the appropriate place for using StatShare, the operating system and

in particular the thread scheduler. There are also additional reasons that support such decision:

- A sampling period is required at the end of which a management decision can be made. Managing the cache must be performed periodically, rather than once, 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 on the order of 25M-50M instructions.
- Although the computational overhead of StatShare is low, management decisions must not be made too often as this will eventually eat into performance gains. Such overhead, however, is negligible at the level of the OS scheduler.
- Another consideration is that cache management must take into account thread priorities. This information is readily available at the OS level which can use it in relation with the output of StatShare to take appropriate decisions.
- 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.

8. EVALUATION

8.1 Experimental Setup

For our simulations we have modified an SMT simulator to model a CMP architecture with 2 cores. All shared structures have been replicated appropriately and the memory hierarchy modeled for independent accesses from the cores. Each core is a modest 2-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 understanding cache sharing and 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 form 64KB to 1MB. The memory latency is 300 cycles. Our intention is to isolate and study only the data accesses 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 and showcase our cache management.

We use a small subset of memory-intensive SPEC2000 benchmarks: art, gzip, equake, 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.

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 the caches for 50M instructions. Management decisions are taken every 45M instructions.

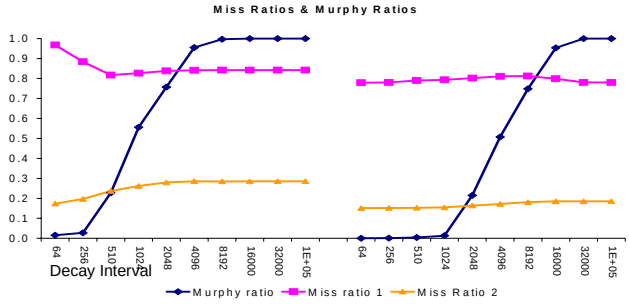

Figure 10. art vs. equake: miss rates and murphy misses

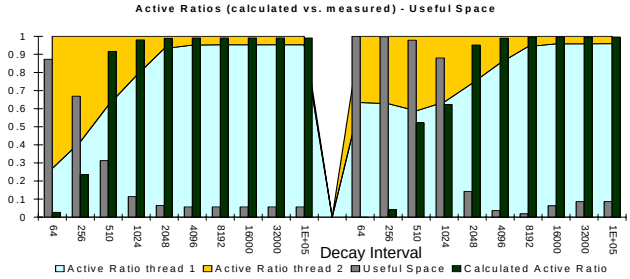

Figure 11. art vs. equake: active ratios and useful ratio

8.2 Results

To understand the behavior of decay in managing sharing, in relation with StatShare's outputs, we have simulated pairs of co-scheduled applications where one of them is decayed. Although our methodology allows any decay interval to be chosen to manage a thread, for simplicity we have constrained decay intervals to a set of 10 (from 64 to 100K CAT). The 100K CAT decay interval corresponds to the *unmanaged* mode, since nothing decays. In addition, we simulate two different cache sizes 64KB and 256KB, which were chosen to induce very high pressure (64K) and medium-pressure (256KB), with the benchmarks we examine. Even larger caches were also examined but do not provide interesting behavior for our study since pairs of co-scheduled SPEC2000 benchmarks fit entirely in such sizes. Subsequent sections discuss results for four representative cases.

art vs. equake. In the example used to illustrate StatShare throughout the paper, art shares the cache with equake. Figure 11 shows the active ratios of the two threads for the two cache sizes, and for the 10 decay intervals for each cache size. For the 100K decay intervals (effectively the unmanaged cases) art's active ratio is 90% for both 64K and 256K caches—clearly "hogging" the cache. Once it is decayed, it releases space for the benefit of equake. However, equake *cannot* exploit its increased space. This is evident from StatShare: equake's reuse-distance histogram (Figure 2) is such that replacing the normal *f*-functions with the f_d-functions (scaled by the Murphy miss ratio shown in Figure 10) produces few additional hits! In short equake does not have the right kind of histogram to benefit from a new scaled L_{eff}. Figure 10 depicts the change in the miss ratios and the Murphy miss ratio for the various decay intervals.

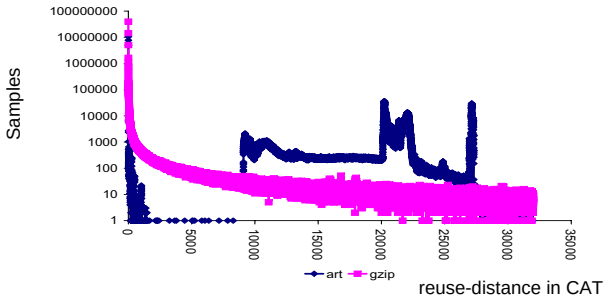

Figure 12. art vs. gzip reuse-distance histograms. gzip can benefit from scaled *f*-functions.

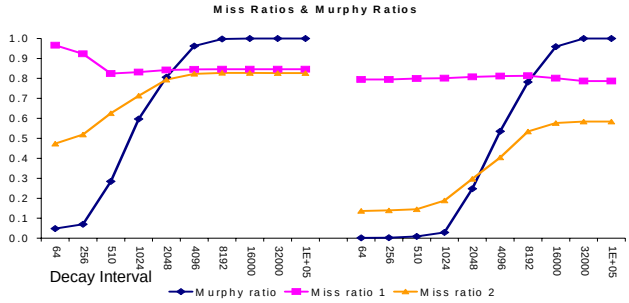

Figure 13. art vs. gzip: miss rates and Murphy misses

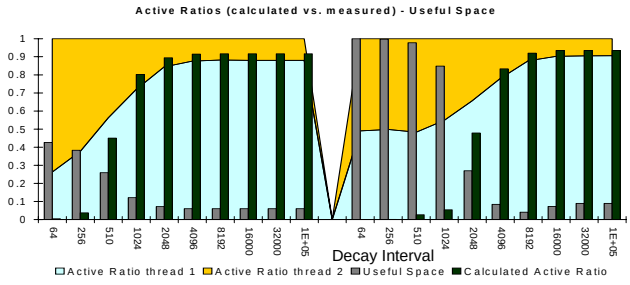

Figure 14. art vs. gzip: active ratios and useful ratio

Figure 11 also shows along the measured active ratios (shaded areas) the computed active ratio for the decaying thread (dark bars) and the useful ratio, i.e., the total useful spacetime divided by the total spacetime (light bars). art's active ratio is computed using the spacetime formulae of Section 5.2. It tracks the measured active ratio but our model breaks for very low numbers of Murphy misses (especially in the 256K cache).The (calculated) useful ratio shows a significant increase and tracks well the improvements in miss ratios.

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

Figure 13 shows how the miss ratios and the Murphy ratio change with the decay intervals for the different caches. In 64K there is a significant improvement for gzip (up to 43% reduction in misses) but decay intervals below 510 CAT hurt art. The situation is much better for the 256K cache, where we get significant improvements for gzip (up to 76% reduction in misses). Figure 14 shows the corresponding active ratios and the useful ratio of the cache. Computed active ratios track the

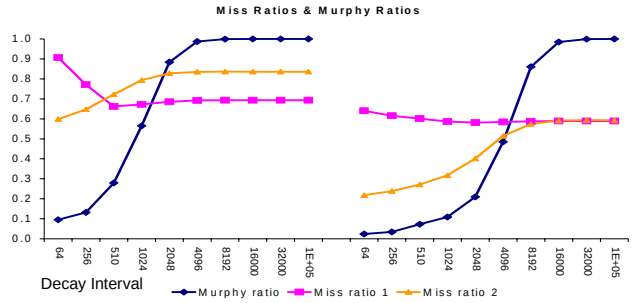

Figure 15. mcf vs. parser: miss rates and Murphy misses

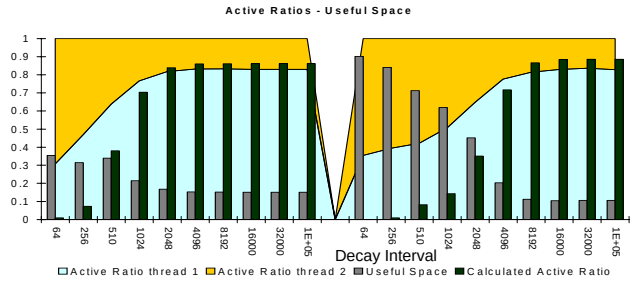

Figure 16. mcf vs. parser: active ratios and useful ratio

actual active ratios for the cases where we have Murphy misses but diverge when Murphy misses fall very low. Here, we see a significant improvement in the useful space for both 64K and 256K caches (3.75x and 4.75x respectively).

mcf vs. parser. Our third example is mcf co-scheduled with parser. mcf is quite similar to art in its behavior. This is also evident from the StatCache miss rate curves (Figure 1). mcf is chosen for decay since it decays better than parser and occupies the most space in the cache (Figure 16). mcf's decay benefits parser with up to a maximum reduction of miss ratio of 28.5% for the 64K and 63% for the 256K cache (Figure 15). Decay leads to improvements in the caches' useful space (Figure 16). In terms of active ratios, the same conclusions hold as in the previous case.

art vs. mcf. Finally, we give a counter example where we are unable to drive down the number of Murphy misses by decaying a single application. This is when two threads such as art and mcf are co-scheduled. art decays slightly better for 64K and 256K caches (and this also shows in Figure 1 where art appears "flatter" for smaller cache sizes). In this example, decay affects little the useful space (Figure 18) and as result we see very little change in the miss ratios of mcf. The large number of Murphy misses lead to better agreement of the computed and actual active ratios (Figure 18).

8.3 Comparison

Finally, we compare decay with coarse-grain partitioning used in [2]. Kim et al, introduce a set of fairness metrics and drive their partitioning scheme accordingly to minimize such metrics [2]. As in [2], the partition unit is equal to one cache way (of the 8-way 256K cache). Since we want to show that decay outperforms coarse-grain partitioning of the cache, *regardless of fairness constraints*, we exhaustively examine all (seven) possible combinations of coarse-grain partitioning between two threads. Figure 19 shows the results and contrasts them with the decay results for 256KB caches for the four co-scheduled application pairs. The decay results comprise of all

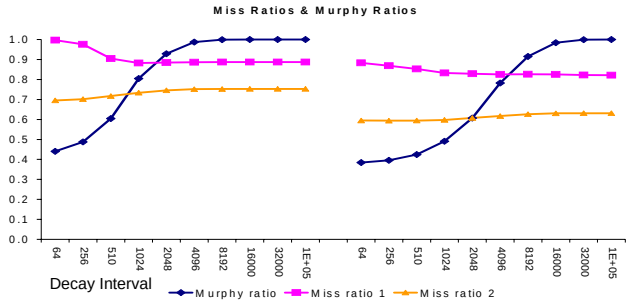

Figure 17. art vs. mcf: miss rates and Murphy misses

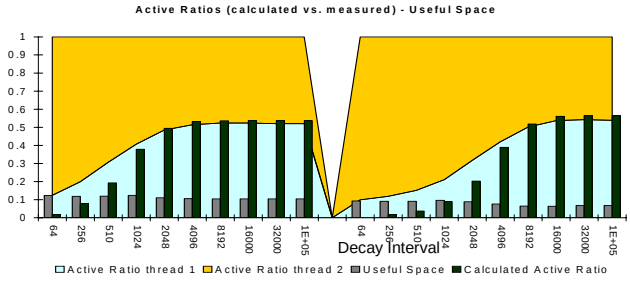

Figure 18. art vs. mcf: active ratios and useful ratio

10 decay intervals. The coarse-grained results comprise of seven combinations, 1-7...7-1, indicating the number of partition units allocated to each thread. For art/equake and mcf/parser, the best decay result is 3.6% better than the best coarse-grain partitioning, while for art/gzip 2%. In most cases with decay intervals less than 4096 (the size of the cache) decay performs very well. In the case of art/mcf, our inability to reduce Murphy misses, results in decay falling behind the coarse-grain partitioning scheme. Furthermore, our policy of allowing the decayed application to reclaim its decayed space, in this case, works against us.

9. CONCLUSIONS

This paper presents and evaluates a statistical cache model for shared caches and a fine-grained way to manage such shared caches on a chip multiprocessor. The input to the statistical model is sparsely sampled data collected during runtime (on-line) in the shared environment. From this data the model is capable of estimating the miss rate and the space time for applications both when they are unmanaged and when they are managed in different 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 such as: maximizing throughput or making sure that all applications get an equal amount of L2 cache.

Our cache control logic is based on decay, but with some important differences. Instead off 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 in a very precise way.

The statistical model and the decay management are verified with a detailed CMP model running memory intensive SPEC

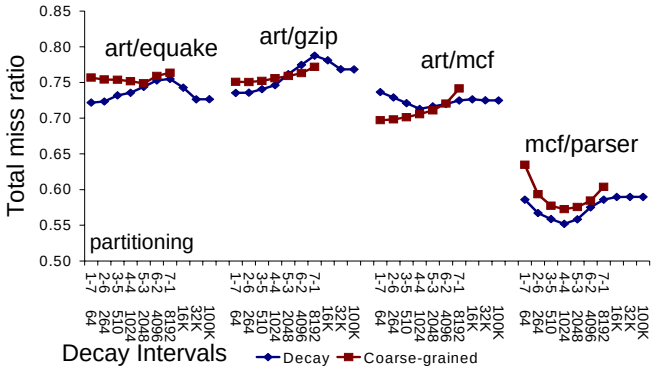

Figure 19. Comparison of decay and coarse-grain partitioning

applications in a SPEC rate fashion. We find the model to be very accurate in predicting both the miss rates and the space occupied by the different applications. Moreover, the modified replacement algorithm makes it possible for system software to maximize performance metrics or to provide custom Quality-of-Service guarantees.

10. 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, 2002.

[2] S. Kim, D. Chandra and Y. Solihin. “Fair cache sharing and partitioning in a chip multiprocessor architecture” Parallel Architectures and Compilation Techniques, 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, 2005.

[4] M.Karlsson and E. Hagersten. “Timestamp-Based Selective Cache Allocation.” In High Performance Memory Systems, edited by H. Hadimiouglu, D. Kaeli, J. Kuskin, A. Nanda and J. Torrellas, Springer-Verlag, 2003.

[5] K. S. Trivedi. “Probability & Statistics with Reliability, Queuing, and Computer Scienece Applications” Prentice-Hall, ISBN 0-13-711564-4.

[6] E. Berg, H. Zeffer and E. Hagersten. “A Statistical Multiprocessor Cache Model” IEEE International Symposium on Performance Analysis of Systems and Software, 2006.

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

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

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

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

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

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

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