Profile-Guided Object-Level Cache Partitioning

Qingda Lu[1], Jiang Lin[2], Xiaoning Ding[1], Zhao Zhang[2], Xiaodong Zhang[1] and P. Sadayappan[1]

[1]Dept. of Computer Science and Engineering

The Ohio State University

[2]Dept. of Electrical and Computer Engineering

Iowa State University

Technical Report

OSU-CISRC-6/08-TR30

Profile-Guided Object-Level Cache Partitioning

Qingda Lu[1], Jiang Lin[2], Xiaoning Ding[1], Zhao Zhang[2], Xiaodong Zhang[1] and P. Sadayappan[1]

[1] Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{luq,dingxn,zhang,saday}@cse.ohio-state.edu

[2]Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
{linj,zzhang}@iastate.edu

Abstract

Efficient on-chip cache utilization is critical to achieve high performance for many memory-intensive applications. The shared cache structure of multi-core processors has made this a more challenging issue due to the increasingly intensive inter-thread contention for shared cache resources. Exsiting hardware and software solutions address this problem by adptively allocating space in the shared cache to multiple threads aiming at minimizing the cache contetion. However, in many applications that cache contention can also be caused by data accesses to several commonly defined data objects in private/shared caches due to the lack of object-level locality knowledge. This observation holds for collaborative threads and/or within a single thread, with a concequence that data of an object with high utility are evicted by accesses to a low-utility object. To address this problem, we present a software framework for object-level cache partitioning. We first collect object-relative stack histograms and inter-object interference histograms via memory trace sampling. With several low-cost training runs, we are able to distinguish data objects of three significant types with different locality patterns: (1) objects with significant temporal resue, (2) object with little temporal reuse but with a large cache footprint, and (3) objects without clear locality patterns. Our cache partitioning policies segregate data objects by locality types and properly allocate cache spaces to data objects, aiming at maximizing cache usage. We have implemented object-level cache partitioning support in Linux kernel, and tested our framework on a commodity multi-core processor. Experimental results show the effectiveness of our system framework with single- and multi-threaded programs from the SPEC CPU2000 benchmark suite and NAS benchmarks. In comparison to uncontrolled LRU caching, our framework provides up to 1.31 speedups and up to 62.5% L2 cache miss reductions in our

experiments.

1 Introduction

The performance gap between the processor and DRAM has been increasing exponentially for over two decades. This "memory wall" problem is likely to persist due to the limited off-chip bandwidth [3]. By providing fast data and instruction buffers to on-chip computing resources, caching has been used to bridge the performance gap between the processor and DRAM. Reducing cache misses and therefore avoiding off-chip accesses is a key to achieve high performance on modern architectures. The technological trend of having chip multiprocessors (i.e. multi-core processors) due to power and heat constraints only makes efficient utilization of limited on-chip cache resources a more challenging problem. Because most proposals choose to use a shared last level cache (LLC), such as L2 or L3 cache to simplify coherence protocols and reduce capacity misses, threads running on different cores may have significant interference with each other in the shared cache. Problems such as performance degradation and unfairness often arise because of such cache contention.

Previous studies [25, 13, 18, 6, 9] have shown the limitations of unconstrained cache sharing and proposed approaches to solving this problem. Despite their differences in metrics and designs, the solutions follow two directions: (1) Partitioning the cache space between threads with additional hardware support. For example, a set-associative cache can be partitioned into ways and different ways can be allocated to different threads. In addition to the basic cache partitioning mechanism, special hardware support is often needed to detect programs' cache utility functions and decide cache partitions at run time. (2) Selecting threads with non-conflicting cache access patterns as co-runners to share the cache. While this approach does not need any hardware support, its usage is limited by the requirement of a large job pool.

The above studies share one common limitation: they target independent workloads in a multiprogramming environment where programs do not share data. However, data-sharing workloads and parallel programs, such as OpenMP programs are becoming increasingly important with chip multiprocessors (CMPs). To address this limitation, we propose object-level cache partitioning to reduce cache misses of sequential and data-sharing parallel programs, an approach orthogonal to inter-thread cache partitioning or optimizing co-running jobs by scheduling. Our approach is motivated by an observation in prior work such as [18]: *LRU caching is demand-driven instead of utility-driven*. With inter-thread cache sharing, a program with higher cache demands obtains more cache resources but is often unable to translate them to higher performance. This observation also holds at finer granularity levels such as cache sharing between data objects and even instructions. To address this problem and maximize the utility of cache resources, we propose to segregate objects that have conflicting access patterns in the shared cache, such as the L2 cache.

In this paper we focus on partitioning the L2 cache space among large global and heap objects for high-performance applications. For a given program, our proposed framework first generates profiles for frequently accessed large objects using training inputs. Based on the profiles we then detect the pattern of the profiles. When the program is scheduled to run with an actual input, we predict its locality profile with the detected access patterns and the actual cache configuration parameters, and then make an object-level partition decision. We have implemented our cache partitioning framework in Linux kernel running on a commodity CMP, and shown its effectiveness.

The contributions of the paper are as follows. First, to the best of our knowledge, this paper is the first work that uses object-level cache partitioning to reduce cache misses for both sequential and OpenMP-style data-sharing parallel programs. In comparison, previous related studies [2, 5, 27] either focus on reducing conflict misses or depend on additional hardware support and modified instruction sets. Second, our approach works across program inputs and cache configurations. The proposed framework is also independent of compiler implementations by working on binary executables. Third, our framework has been implemented and evaluated in commodity systems instead of simulation environments, therefore it can be directly used in practice to improve application performance.

The rest of the paper is organized as follows. We first discuss a motivating example for our work in Sec. sec:motivation. We then present an overview of the proposed object-level cache partitioning framework in Sec. 3. In Secs. 4, 5 and 6, we describe how we generate program profiles, analyze generated profiles and make partition decisions based on the analysis results with a given cache configuration respectively. We evaluate the effectiveness of our approach in Sec. 7 on a commodity CMP using several programs from SPEC CPU2000 and NAS benchmarks. We discuss related work in Sec. 8 and present our conclusions in Sec. 9.

2 A Motivating Example

Here we use the conjugate gradient (CG) program in NAS benchmarks as a motivating example to illustrate the problem. As shown in Fig. 1, CG spends most of its running time on a sparse matrix-vector multiplication $w = a \cdot p$, where a is a sparse matrix, $rowstr$ and $colidx$ are row and column index arrays and w and p are dense vectors. In CG, the majority of accesses are on arrays a, p and $colidx$. Although vector p has high temporal reuse in the matrix-vector multiplication code, depending on its size, its elements may get constantly evicted from cache before their reuses due to the streaming accesses on arrays a and $colidx$. As the result of this thrashing effect from accessing arrays a and $colidx$, CG reveals a streaming cache access pattern in cache. Without special code/data treatment based on domain knowledge, general compiler optimizations, such as tiling, cannot be applied in this case because of the irregular nature of this program — there is indirection in most

array accesses.

```
for (i = 0; i < niters; i++) {
   ... ... // other code, with accesses to arrays not shown
   for (j = 1; j <= lastrow-firstrow+1; j++) {
       sum = 0.0;
       for (k = rowstr[j]; k < rowstr[j+1]; k++) {
           sum = sum + a[k]*p[colidx[k]];
       }
       w[j] = sum;
   }
   ... ... // other code, with accesses to arrays not shown
}
```

Figure 1. An outline of NAS-CG code.

If we allow the cache space to be partitioned between objects, there are different ways to reduce and even completely eliminate capacity misses on array p without increasing the misses on other objects. One approach is that we can protect p in an exclusive cache space and leave the remaining cache capacity to the rest of the objects. Alternatively, we can divide the cache such that the minimum cache quota is given to arrays $colidx$ and a. This optimization is not limited to single-thread performance. When the code is augmented with OpenMP directives, with a shared cache the above partition decisions can also reduce capacity misses. If we take the approach to keeping the minimum cache quota for arrays $colidx$ and a and co-schedule CG with other programs, since it does not reveal a streaming access pattern that significantly interferes with its co-runners, high throughput can be achieved with judicious inter-thread cache partitioning. In this paper, we focus on object-level cache partitioning and leave the combination of inter-object and inter-thread cache partitioning to our future work.

To quantify the improvement from object-level cache partitioning, let us assume that CG runs on processors with 64-byte L2 cache lines and 2MB L2 cache capacity. For simplicity of our discussion, we also assume non-zero elements in a and $colidx$ are distributed in such a way that exactly one of every two consecutive elements in p is used in the innermost loop in Fig. 1. Assuming p has 160000 elements (1250KB) and there are 4×10^6 total accesses on array p in CG, there are 4×10^6 accesses on arrays a and $colidx$ respectively. As floating-point elements in a and p are 8 bytes and elements in $colidx$ are 4 bytes, while accesses on p touch 20000 cache lines in the innermost loop of Fig. 1, accesses on p and $colidx$ only read 10000 and 5000 cache lines respectively. With uncontrolled cache sharing, because 35000 distinct cache lines are referenced between a data reuse on p while the L2 cache only has 32768 cache lines, CG's cache miss rate is 100%. If we can apply either one of the discussed object-level cache partitioning schemes, we can eliminate all the misses on p, which reduces the miss rate to 42.9%. We keep using the assumed cache parameters and program inputs here in the next several sections to

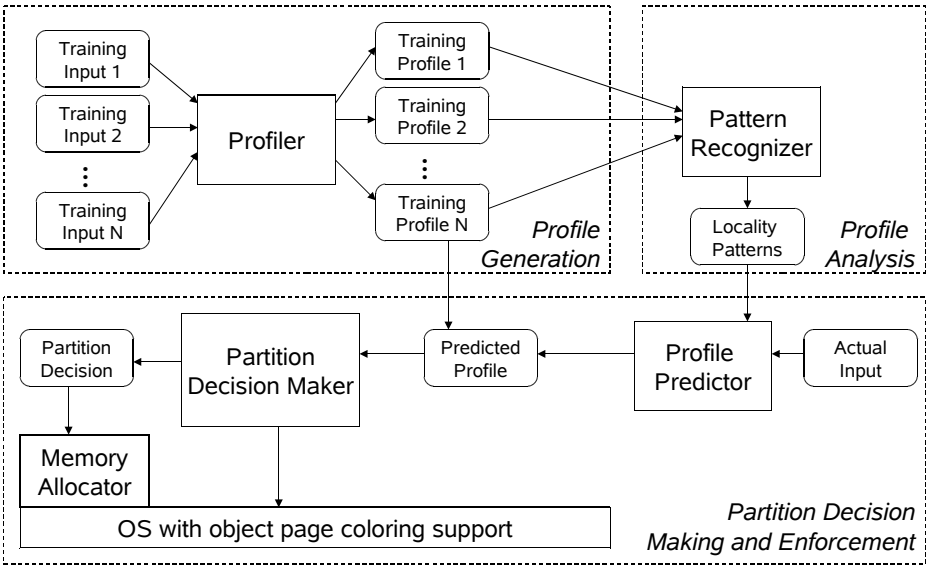

Figure 2. Overall structure of the object-level cache partitioning framework.

illustrate our cache partitioning framework.

3 Overview of the Approach

The CG example in Fig. 1 demonstrates the benefits of partitioning cache space at the object level. In order to do so, we need to solve the following important problems: (1) How can we identify important objects as partitioning candidates and capture the interference among the objects that share and compete for cache space? (2) In which way can we capture data reuse patterns at the object level, across cache configurations and program inputs? (3) How can we make quick inter-object partition decisions with a new program input? (4) What is the minimum hardware support we need to enforce cache partition decisions? Without such a mechanism available, can we still evaluate our approach in an efficient way and even still make our approach useful in practice?

To solve the above problems, we propose a framework to detect programs' data reuse patterns at the object level through memory trace profiling and pattern recognition and enforce partition decisions at run time with operating system support. This proposed framework consists of the following steps and is summarized in Fig. 2.

1. *Profile Generation.* For a given program and its training inputs, we capture memory traces in an object-relative form through binary instrumentation. Instead of keeping raw traces, we obtain object-relative stack distance histograms and inter-object interference histograms for large or frequently accessed objects. These histograms are program profiles that represent the program's data locality patterns.

2. *Profile Analysis.* Based on program profiles from training runs, we detect the program's object-relative data locality

patterns, using a modified version of the data locality pattern recognition algorithm by Zhong et al. [29].

3. *Cache Partition Decision Making and Enforcement.* When the program is scheduled to run with an actual input, we predict its object-relative stack histograms. with the detected access patterns. In combination with inter-object interference information, we obtain the the locality profile for the input. Then using this locality profile and the cache configuration information of the underlying system, we make an object-level partition decision. Our partition decisions are finally enforced on commodity CMPs based on an operating system technique called *page coloring* [14].

In this paper, we focus on global and heap objects larger than a threshold T_{obj}. If an object's size is less than T_{obj}, it is merged into a special object group obj_0. There are two reasons for this decision: First, we focus on reduction of L2 cache misses for scientific programs. In these programs, data localities on small objects are often exploited in L1 instead of L2 cache. Second, it is simply not feasible to include all objects due to the complexity and large memory requirement.

4 Profile Generation

In order to make cache partition decisions, we need to understand locality patterns of a program at the object level. We make several important decisions with respect to reference locality modeling.

- The classical reference locality model categories data localities into temporal and spatial localities. In this work, we only focus on temporal locality at the cache line granularity because spatial locality is taken care of by viewing a complete cache as the basic unit. While this approach may appear to affect the proposed framework's generality, it is not a problem because our framework aims at detecting the data locality patterns of a given program binary that works on processors in the same processor family. While cache capacities and degrees of associativity often vary, processors in a modern processor family are unlikely to use different line sizes at the same cache level. For example, Intel X86 processors with NetBurst and Core microarchitectures all use 64-byte L2 cache lines.
- An important design choice is to model reference locality statically or dynamically. While there has been prior work such as [7, 4] that models programs' cache behaviors statically and symbolically, these approaches can hardly be used in practice. They are limited to regular loop code and do not work across compilers that apply different compiler optimizations to the same program. We choose to follow a dynamic approach through binary instrumentation.

4.1 Modeling Object-Relative Temporal Locality

We model temporal locality using *stack distance* (i.e. reuse distance) [29, 4], defined as the number of distinct data references between two references to the same data. Since we decide to model data locality at the cache line granularity, stack dis-

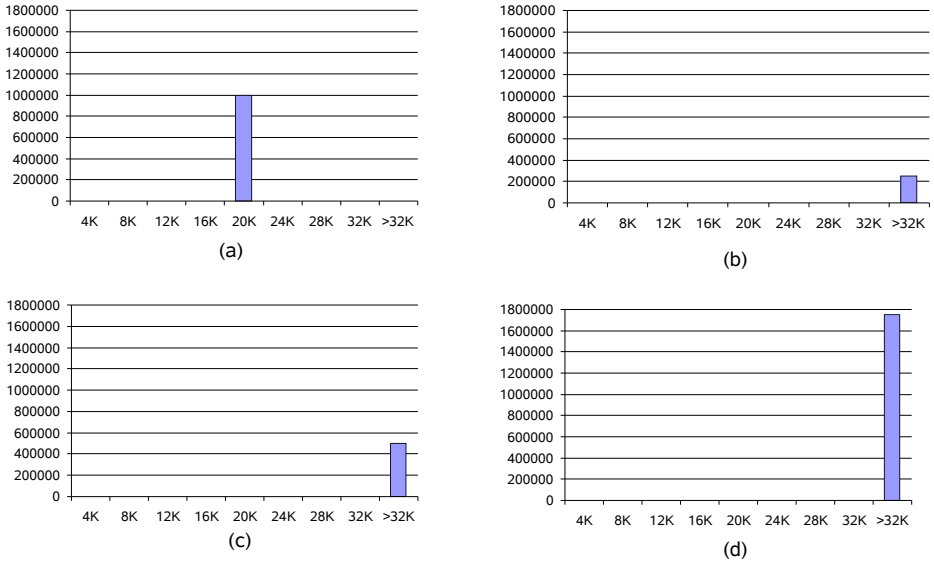

Figure 3. Object-relative stack distance histograms for frequently accessed objects in CG and the whole-program stack distance histogram for CG. (a) Object-relative stack distance histogram for p. (b) Object-relative stack distance histogram for $colidx$. (c) Object-relative stack distance histogram for a. (d) Whole-program stack distance histogram for CG.

tance refers to the number of distinct cache lines accessed between two accesses to the same cache line. As it is not feasible to record the stack distance between each data reuse, a histogram is used to summarize the temporal locality. In a stack distance histogram, the distance space is divided into N consecutive data ranges $(0, R_1]$, $(R_1, R_2]$,...,$(R_{N-2}, R_{max}]$,$(R_{max}, +\infty)$ and the value of each range represents the percentage or the absolute number of temporal reuses whose stack distances falling in to this range. R_{max} is the largest cache capacity we consider in terms of cache lines. Our approach differs from prior work on reference locality modeling in several aspects: (1) Stack distances in prior work are at the whole program level but in this paper we model temporal locality by keeping a stack distance histogram d_A for accesses within each object A. In such a way we are able to identify individual objects' locality patterns and treat objects differently in later stages of the framework. (2) For each object-relative stack distance histogram, instead of using a relative metric, we keep absolute reuse counts. This is because with different inputs an object's accesses may have a varying weight the whole-program accesses. (3) We ignore reuses with zero stack distances because such reuses are mostly handled by L1 cache while we optimize L2 cache accesses in this paper.

Assuming we run CG with the cache configurations and the program input in Sec. 2, we have object-relative stack distance histograms for objects p , $colidx$ and a as shown in Fig. 3(a), Fig. 3(b) and Fig. 3(c) respectively. In comparison, a whole-program stack distance histogram for CG is shown in Fig. 3(d), where all accesses have stack distances larger than the cache capacity.

4.2 Modeling Inter-Object Interference

With object-relative temporal reuse information alone we cannot decide if a partition decision is better than another. We need to predict the overall cache miss rate by composing the data locality profiles of individual objects. In order to do so, we model reference interference between different objects. *Inter-object interference* $I_{A,B}$ is defined as the average number of distinct data references to object B per distinct reference to object A. In our framework, similar to temporal locality modeling, we extend the above definition and model inter-object interference at the cache line level. Note that inter-object interference I is not symmetric, that is, $I_{A,B}$ and $I_{B,A}$ may not be identical. For a simple regular program, $I_{A,B}$ can be a constant. However, for complex programs with multiple phases, $I_{A,B}$ may vary with phase changes, often with changes on stack distances of objects A and B. Therefore we use a histogram to summarize inter-object interference $I_{A,B}$ whose ranges correspond to those in stack distance histograms for d_A and d_B and heights represent inter-object interference values.

As an example, Fig. 4 shows the interference histograms for $I_{a,p}$ and $I_{p,a}$ in CG. Fig. 4(a) shows that interference from object p to object a's locality is almost 0 since vector p is accessed repeatedly between any temporal reuse of a. In contrast, Fig. 4(b) shows that $I_{a,p} = 2$ because there are two distinct cache lines of p accessed between a temporal reuse on a cache line of a. We can see object a significantly interferes with object p's temporal reuses.

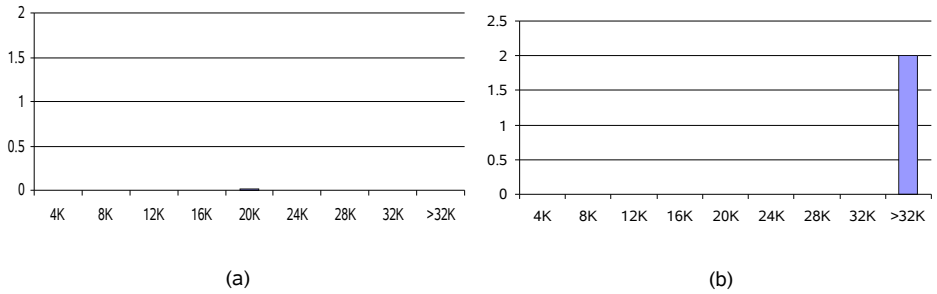

Figure 4. Examples of inter-object interference histograms for CG. (a)Inter-object interference $I_{p,a}$. (b)Inter-object interference $I_{a,p}$.

4.3 Profile Generator

Fig. 5 illustrates how a program profile, which consists of object-relative stack distance histograms and inter-object interference histograms, is generated with a training input. There are three important components used in profile generation: *object table*, *custom memory allocator* and *memory profiler*.

The object table maintains the basic information of every profiled object. As the hub of the profiling process, it is updated and queried by both the custom memory allocator and the memory profiler. Object information stored in the object table

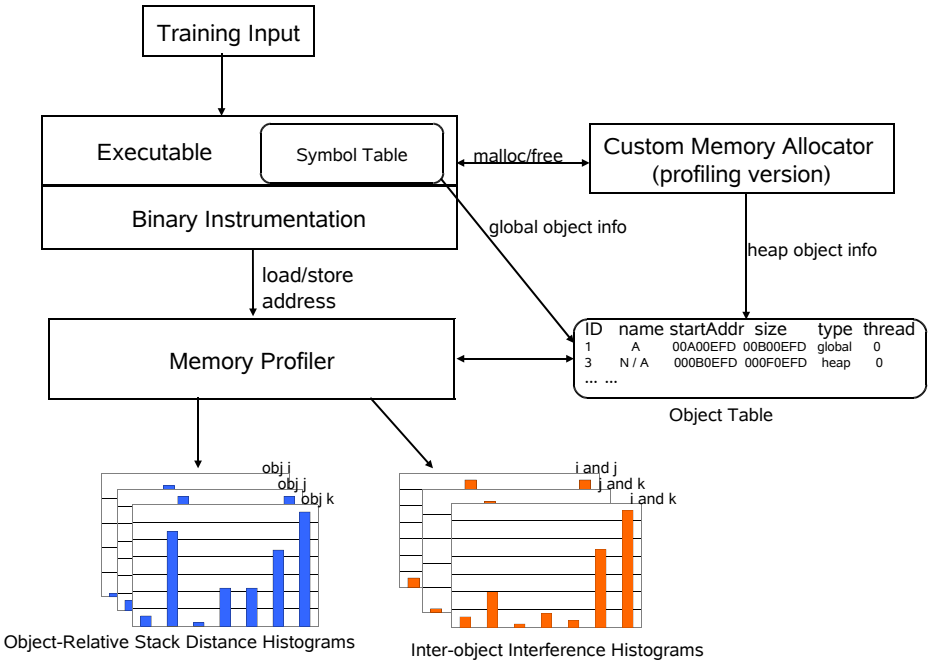

Figure 5. Program profile generation with a training input.

includes object identifier, name, starting address, size, type, and thread number. In this paper we focus on global and heap objects therefore an object's type is either *heap* or *global*. An object's identifier is used to facilitate fast query and retrieval. A global object's identifier is decided by its order in the symbol table of the binary executable. A heap object's identifier is calculated by a hash function that takes its allocation site, allocation order and the total number of global objects as parameters. Identifier 0 is reserved for the special object group obj_0. Similar to identifiers, object names are retrieved from the symbol table for global objects and decided by a function mangling allocation sites and allocation order for heap objects. The thread number field is used to record the allocator of a heap object in order to identify thread-private data. However, in this paper we focus on shared objects and in our experiments all heap objects are allocated by thread 0. Therefore we ignore this aspect in the following discussion. Because heap objects may have overlapping address ranges due to their non-overlapping life cycles, a raw address can be found within multiple objects. To avoid this problem, when a heap object's memory space is released, we move its information to an area dedicated to freed objects in the object table.

The custom memory allocator is used to capture each heap object's creation and deletion. We replace standard memory management functions such as malloc(), calloc(), free(), and realloc() with our implementations. In this way during profiling runs these memory management requests are redirected and recorded.

The memory profiler controls the profiling process and starts a profiling run by updating the object table with global object information read from the program's symbol table. It relies on binary instrumentation to obtain the raw address stream of a given program. Our current profiler implementation is written as a tool based on PIN [16] that inserts instruction

and object probes before every instruction accessing the memory. The core components in the profiler are a set of *stack distance profilers* and an *inter-object interference counter table*. A stack distance profiler is used to track the stack distances of one object and implemented with a hash table and a Splay tree following Sugumar and Abraham [24]. The inter-object interference counter table is two-dimensional and used to sample inter-object interference between temporal reuses. The memory profiling algorithm used is summarized in Algorithm 1. By sampling 10% of memory references and employing several optimizations, our current implementation of the memory profiler has a reasonable slowdown of 50 to 80 times while also maintaining a high accuracy.

Algorithm 1 The memory profiling algorithm.

$tracedAddr[0..objNum-1] \leftarrow 0$
for each memory reference with raw address $addr$ **do**
  Search the object table for such a live object obj that $obj.startAddr \leq addr$ and $obj.startAddr + obj.size \geq addr$
  $addr \leftarrow addr/CacheLineSize$
  if obj exists **then**
    $(objID, offset) \leftarrow (obj.ID, addr - obj.startAddr)$
  else
    $(objID, offset) \leftarrow (0, addr)$
  end if
  $stackDist \leftarrow stackDistProfiler[objID].trace(offset)$
  $objStackDistHistogram[objID].sample(stackDist, 1)$
  if $tracedAddr[objID] = 0$ **then**
    $tracedAddr[objID] \leftarrow addr$
    continue to process next memory access
  end if
  if $tracedAddr[objID] = addr$ and $stackDist \neq 0$ **then**
    $tracedAddr[objID] \leftarrow 0$
    for each i where $i \neq objID$ **do**
      if $interferenceCounter[objID][i] \neq 0$ **then**
        $interferenceCountHistogram[obj][i].sample(stackDist, interferenceCounter[objID][i]/stackDist)$
        $sampleCounterHistgram[obj][i].sample(stackDist, 1)$
        $interferenceCounter[objID][i] \leftarrow 0$
      end if
    end for
  end if
  for each j where $j \neq objID$ **do**
    if $stackDist > interferenceCounter[j][objID]$ **then**
      $interferenceCounter[j][objID] \leftarrow interferenceCounter[j][objID] + 1$
    end if
  end for
  for each obj i **do**
    for each obj where $i \neq j$ **do**
      for each range k **do**
        $inteferenceHistogram[i][j][k] \leftarrow \frac{interferenceCountHistogram[i][j][k]}{sampleCounterHistgram[i][j][k]}$
      end for
    end for
  end for
end for

5 Profile Analysis

After at least two program profiles with different training inputs are obtained from the profiling process, we analyze these profiles with the three goals: (1) categorizing objects based on their locality types, (2) searching object cliques and (3) detecting the patterns of object-relative locality, object sizes and data access volumes.

Object Categorization. With program profiles, we categorize objects into four types based on their data locality patterns:

1. *Cold objects* refer to the objects that have few memory accesses while still traced in the profiling process because of their large sizes. We set a threshold T_{cold} to detect cold objects. For an object obj, if $\frac{\#accesses\_obj}{\#totalAccesses} < T_{cold}$, it is categorized as an cold object. $\#accesses\_obj$ is the number of accesses to the object obj, and $\#totalAccesses$ is the total number of accesses to all the objects. As an exception, we never try to identity obj_0 as a cold object.
2. *Hog objects* refer to the objects that have high memory demands but reveal little or no temporal locality. With a threshold T_{hog}, If an object obj is not a cold object and $\frac{\#accesses_{obj}(stackDist \leq R_{max})}{\#accesses_{obj}} < T_{hog}$, then it is a hog object. $\#accesses_{obj}(stackDist \leq R_{max})$ refers to the number of references to object obj with reuse distances less than R_{max}.
3. *Hot objects* are the objects with high temporal locality. For an object obj, if we have $\frac{\#accesses_{obj}(stackDist \leq R_{max})}{\#accesses_{obj}} > T\_hot$ and if obj is not a cold object, then it is a hot object.
4. *Other objects* are the objects that do not belong in any above category.

For example, after object categorization, profiled objects in CG are categorized into groups, as shown in Table 5. Because the majority of the objects are categorized as cold objects, it significantly simplifies further analysis and partition decision making procedure.

Cold Objects	Hog Objects	Hot Objects	Other Objects
iv,v,$acol$,$arow$,$rowstr$,x,w,r,q,z,$aelt$	a, $colidx$	p	obj_0

Table 1. Categorizing objects in CG's profiles.

Object Clique Search. We view a set of objects and their interference relationships as a graph with vertices representing objects and unweighted edges representing interferences between objects. Based on the graph representation, we enumerate the cliques inside hog objects. As we will show in the next section, the cumulative object size of the maximum clique decides the memory requirement of all hog objects and thus decides their cache allocation. While the clique enumeration problem has exponential space and time complexities, it does not bring much overhead in our particular problem because the number

of hog object cliques is always fairly small in real programs. For example, CG has only one hog object clique that includes p and $colidx$.

Pattern Recognition We detect object-relative data locality patterns following the approach by Zhong et al. [29]. Each object-relative stack distance histogram is divided into n small groups. With two profiles p_1 and p_2 having different input sizes, we find a function f_k to fit each group formed for the two profiles, $g_{1,k}$ and $g_{2,k}$ for k from 1 to n. f_k matches stack distances $d_{1,k}, d_{2,k}$ such that $f_k(inputSize_1) = d_{1,k}$ and $f_k(inputSize_2) = d_{2,k}$. What differs ours from the approach by Zhong et al. is that f_k in our approach is a polynomial function instead of functions such as square root. This is because we use one program argument as the input parameter instead of object sizes. As we keep absolute numbers instead of percentages in histograms, we detect the data access volume and data size patterns of each object as well. The approach to detecting these patterns is similar to temporal locality pattern recognition presented above. As an example, after pattern recognition, we find that in CG p's object-relative stack distances grow linearly with program parameter na. In contrast, $colidx$ and p's stack distances are simply predicted as larger than the cache size across input sizes.

6 Cache Partition Decision Making and Enforcement

Fig. 6 illustrates how our framework makes a partition decision with an actual input and how the partition decision is enforced on a commodity CMP through software. When the program with training profiles is scheduled to run with a large input, we first predict the object-relative stack distances, object sizes and data access volumes with patterns recognized during profile analysis. The partition decision maker then selects objects to be isolated from the rest of the data space and decide their cache quotas. Finally this decision is enforced by OS via virtual-physical address mapping. This phase of the framework shares several components with the profile generation process such as the object table and the custom memory allocator but with a few changes. For example, in the object table field $colors$ is added to denote the cache space allocated to an object as the total cache capacity is divided into a set of *colors*(regions). Unlike the custom memory allocator of the profiling process, the memory allocator used in this stage reads the object table to check and enforce cache partition decisions but never updates the table. The rest of this section discusses in detail the partitioning mechanism and the partition decision making algorithm.

6.1 Partition Enforcement

The most straightforward way to enforce cache partition decisions is through hardware support. Cache partitioning can be at different granularity levels such as cache lines, cache ways or pages. Because hardware-based partitioning support is not readily available, here we provide a software mechanism that essentially emulates page-level hardware cache partitioning

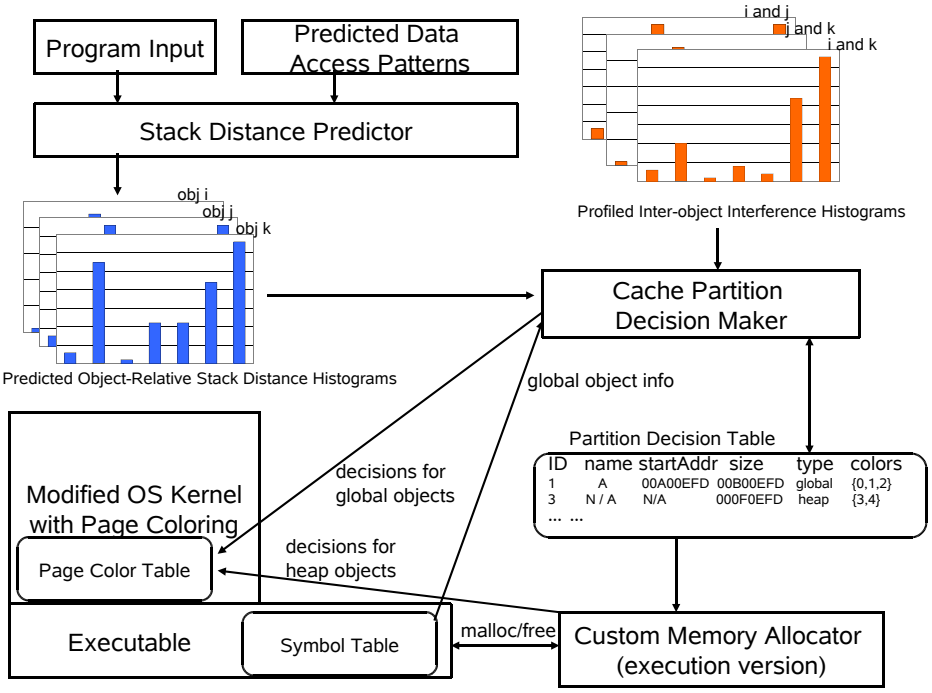

Figure 6. Cache partitioning decision making and enforcement with an actual input.

based on a well accepted OS technique called *page coloring* [26]. It works as follows. A physical address contains several common bits between the cache index and the physical page number. These bits are referred to as *page color*. A physically addressed cache is therefore divided into non-overlapping regions by page color, and pages in the same color are mapped to the same cache region. We assign different page colors to different objects and/or threads, thus, cache space is partitioned among objects. By limiting the physical memory pages of an object within a subset of colors, the OS can limit the cache used by the object to the corresponding cache regions. In our experiments, the Intel Xeon processor used has 4MB, 16-way set associative L2 caches, each shared by two cores. The page size is set to 4KB. Therefore, We can break the L2 cache to 64 colors (cache size / page size / cache associativity).

Our implementation is based on the Linux kernel. We maintain a *page color table* for threads sharing the same virtual memory space to guide the mapping between virtual and physical pages. Each entry in the table specifies a set of colors that the virtual page can be mapped to. Each thread has a pointer in its task structure pointing to the page color table. We also modify the buddy system in the memory management module of the Linux kernel, which is in charge of mapping virtual pages to physical pages, to follow the guidance specified in the page color table. We add a set of system calls to update the page color table at user level. These system calls are used by the partition decision maker and the memory allocator to enforce cache partition decisions for global and heap objects.

Because all the threads in a thread family share the same virtual memory space, they share the same page color table. In

our implementation, each table entry occupies one byte and therefore the page color table incurs no more than 1MB space overhead for each process family in 32-bit systems. Thus our implementation has a small space overhead. Because all the decisions are made at user level, there is no run-time overhead in the kernel.

6.2 Partition Decision Making

The partition decision making algorithm consists of three major steps. (1) To simplify late parts of the algorithm, we first merge cold objects' stack distance histograms and inter-object interference histograms with those of the special object group obj_0. (2) Although in theory hog objects do not need any cache space, we still need to allocate enough cache capacity to them because the L2 cache and physical memory are co-partitioned by the page color-based partition enforcement mechanism. (3) Finding the optimal cache partition is NP-hard because the decision problem of integer linear programming can be reduced to this problem. Since it is not feasible to search the best partition decision in a brute-force way, we employ a heuristic to maximize the benefit-cost ratio at every cache allocation step until there is no further benefit of cache partitioning. The complete partition decision making algorithm is summarized in Algorithm 2.

The key operation in the partition decision making algorithm is estimation of shared cache misses on a set of objects with a given cache size. This essentially relies on mergence of object-relative stack distance histograms and inter-object interference histograms of a set of objects. Once we have the combined stack distance histogram, all the accesses with stack distances larger than the given cache capacity are concluded as misses. The combination process is as follows.

- For an reference on object i, its stack distance in the combination object set S is computed as:

$$dist_S = dist_i + \sum_{j \in S - \{i\}} interference((i,j), dist_i)$$

- For an object set S, its combined inter-object interference with an object k not in this group is computed as:

$$interference((k,S), dist_k) = \sum_{j \in S} interference((k,j), dist_k)$$

7 Experimental Results

In this section we present the experimental results using the object-level cache partitioning framework.

Experimental Environment We conducted experiments on a Dell PowerEdge 1900 workstation that has two quad-core 2.66GHz Xeon X5355 Processors and 16GB physical memory with eight 2GB dual-ranked Fully Buffered DIMMs (FB-DIMM). Each X5355 processor has two pairs of cores and cores in each pair share a 4MB, 16-way set associative L2 cache.

Algorithm 2 The partition decision making algorithm.

Require: Predicted object-relative stack distance histograms and inter-object interference histograms

STEP 1. (Merge cold objects)
All cold objects are merged into obj_0
STEP 2. (Find the minimum cache space for hog objects)
Find the clique in hog objects with the largest memory requirement mem_{hogs}
$hogCacheColors \leftarrow \lceil mem_{hogs}/(totalMemory/\#pageColors) \rceil$
$cacheColors \leftarrow totalCacheColors - hogCacheColors$
STEP 3. (Heuristic-based cache partitioning for hot objects)
$partitionedObjs \leftarrow \phi$, $objsLeft \leftarrow hotObjs$
while $objsLeft \neq \phi$ **do**
  $bestBenefitCost \leftarrow 0$
  for each object obj in $objsLeft$ **do**
    for $colors = 0$ to $cacheColors$ **do**
      Try to find non-conflicting colors in assigned colors to $partitionedObjs$
      $cost \leftarrow (colors - nonConflictingColors)$
      $benefit \leftarrow misses(cacheColors, objsLeft) - (misses(cacheColors - cost, objsLeft) + misses(colors, obj))$
      if $benefit/cost > bestBenefitCost$ **then**
        $bestBenefitCost \leftarrow benefit/cost$
        $(obj_{best}, colors_{best}) \leftarrow (obj, colors)$
      end if
    end for
  end for
  if $(obj_{best}, colors_{best})$ is not empty **then**
    $partitionedObjs \leftarrow partitionedObj \cup \{obj_{best}\}$
    $objsLeft \leftarrow objsLeft - \{obj_{best}\}$
    $cacheColors \leftarrow cacheColors - colors_{best}$
  else
    Break from the while loop
  end if
end while

Each core has a private 32KB instruction cache and a private 32KB data cache. Both adjacent line prefetch and stride prefetch are enabled on this machine. Because we target shared cache performance, in our experiments we used at most a pair of cores via function *sched_setaffinity* that sets process/core affinity. Our cache partitioning mechanism was implemented in Linux kernel 2.6.20.3. While there are 64 page colors in the shared L2 cache, we only used 5 least significant color bits in a physical address. Therefore we have 32 colors and each color corresponds to 128KB cache space. Without incurring page swapping, the maximum physical memory mapped to a page color is 512MB on this machine.

Benchmark Selection We selected a set of memory-intensive programs from NAS benchmarks [1] and Spec CPU2000 benchmark set [23]. These benchmarks and their characteristics are summarized in Table 7. In some cases, to fully test our framework, we had to make source code changes in the selected benchmarks due to two limiting factors. First, some Fortran programs use common blocks which makes global objects in a common block indistinguishable in the symbol table. Due to this we modified $swim$'s source code such that every global object was only in one common block. We also chose to use a C implementation of NAS benchmarks instead of the original Fortran programs to avoid this complexity. Second, some C programs use a programming idiom that creates a multi-dimensional heap array from many dynamically allocated sub-arrays. We can modify such code by allocating memory to the array at once. Such a change is needed for art from Spec CPU2000. Note that there is no difference in performance or memory requirement before and after such simple changes.

Table 7 also shows the training inputs and actual inputs we used for each benchmark. For each benchmark, we first had two training runs to obtain training profiles through binary instrumentation. We then analyzed the profiles by categorizing objects and different program patterns. Finally for each (actual input, thread number) combination in Table 7 a cache partition decision was made and then enforced during an actual run, to examine if there was an observable performance improvement (i.e. over 2% running time reduction) over unconstrained cache sharing.

Table 7 shows that using object-level cache partitioning we achieved observable performance improvements on NAS-CG, NAS-LU and Spec-art with at least one (actual input, thread number) combination. For these three benchmarks, speedups observed are up to 1.29 (Spec-art with $objects = 40$) and cache miss reduction is up to 34%(NAS-LU with 4M cache $problem\_size = 30$). For NAS-BT and Spec-swim, there was no observable performance improvement with any given input sizes and thread numbers and there is also no observable performance degradation (i.e. over 2% running time increase). Further investigation showed that for NAS-BT the partition decision maker did not choose to segregate any object with actual inputs under a 4MB cache. For Spec-swim, there were 6 hog objects identified and segregated. However, the resulted cache miss reduction is negligible because accesses on these hog objects only account for a very small part of the total accesses. To further understand the effect of object-level cache partitioning, we focused on the three benchmarks with observable

performance improvements. For each benchmark, extensive experiments were conducted by varying thread number (1 or 2), input size and the effective cache capacity (2M or 4M through page coloring).

Benchmark	Language	Code modified?	Num. of objects profiled	Input parameter
NAS-CG	C	No	15	na
NAS-LU	C	No	9	$problem\_size$
NAS-BT	C	No	13	$problem\_size$
Spec-art	C	Yes	7	$objects$
Spec-swim	Fortran	Yes	15	$N = N1 = N2$
Benchmark	**Training input**	**Actual input**	**Num. of Threads**	**Improvement observed?**
NAS-CG	7000 (Class W), 14000 (Class A)	75000 (Class B), 150000 (Class C)	1,2	Yes
NAS-LU	12 (Class S), 33 (Class W)	64 (Class A), 102 (Class B)	1,2	Yes
NAS-BT	12 (Class S), 24 (Class W)	64 (Class A), 102 (Class B)	1,2	No
Spec-art	10, 20	30,40	1	Yes
Spec-swim	257,513	1061, 1335	1	No.

Table 2. Characteristics of selected benchmarks and experimental designs.

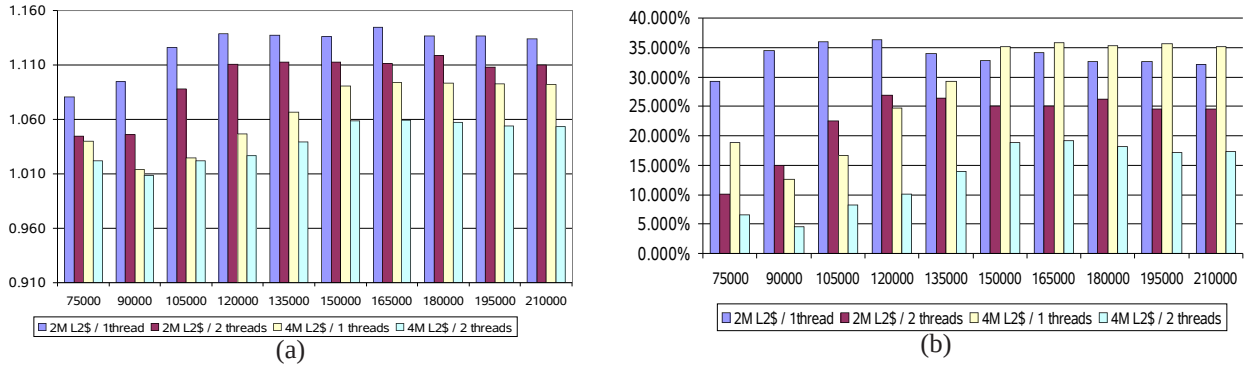

Figure 7. NAS-CG with object-level cache partitioning. (a) Speedups and (b) Cache miss reductions in comparison to uncontrolled LRU caching.

Fig. 7 shows the speedups and cache miss reduction numbers on NAS-CG in comparison to an uncontrolled shared cache. For NAS-CG, with different thread numbers, input sizes and cache sizes, the partition decision is unchanged. It always segregates hog objects $colidx$ and a and lets the rest of the objects share the remaining cache capacity. There is performance improvement and miss rate reduction across cache configurations and input sizes. When cache size is small (2MB), the effect of cache partitioning is more obvious than with a larger cache (4MB). This is as expected since p's capacity misses increase when the cache capacity is reduced while a and $colidx$'s number of misses being close to a constant. When the thread number

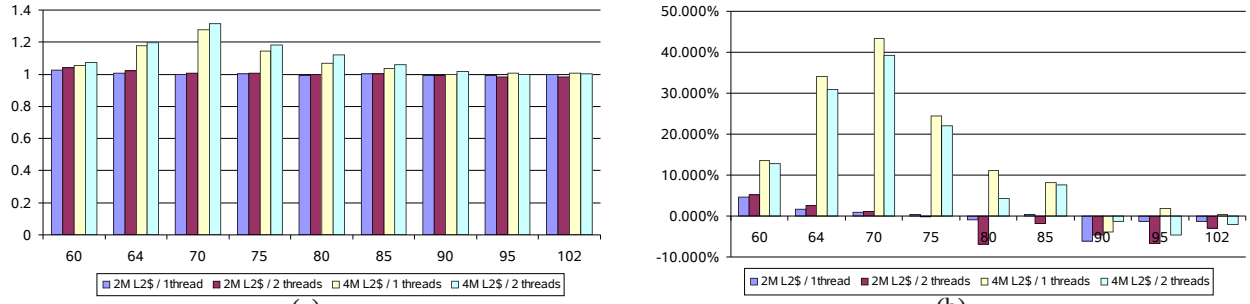

Figure 8. NAS-LU with object-level cache partitioning. (a) Speedups and (b) Cache miss reductions in comparison to uncontrolled LRU cachi

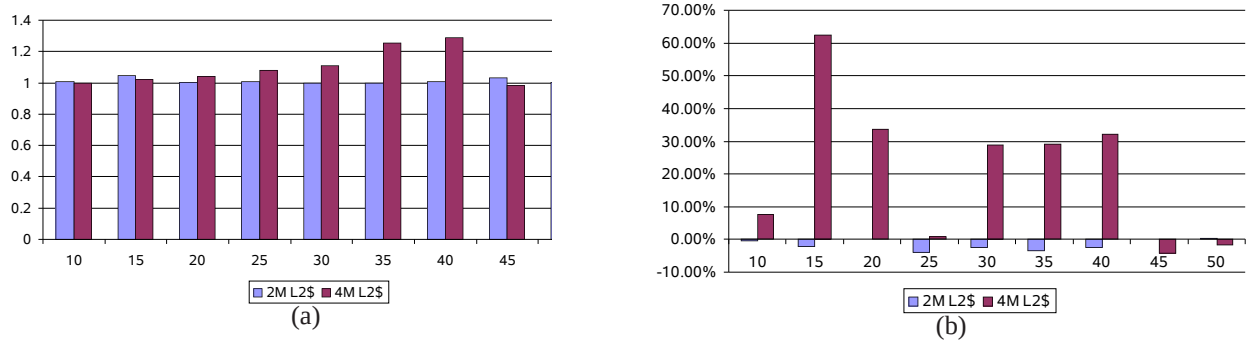

Figure 9. Spec-art with object-level cache partitioning. (a) Speedups and (b) Cache miss reductions in comparison to uncontrolled LRU caching.

is increased from one to two, with NAS-CG the benefit of segregating hog objects is reduced. For example, when two threads are used, with 2MB cache size the average performance improvement with object-level cache partitioning is reduced from 12.7% to 9.6%. Experimental data show that NAS-CG's total L2 misses increase from one thread to two threads while a and $colidx$'s cache miss rates do not change. The reduction of performance improvement is likely due to increasing intra-object cache contention on p between two threads.

Figs. 8 and 9 show the speedups and cache miss reduction numbers on NAS-LU and Spec-art, which are different from those on NAS-CG. Object-level cache partitioning offers very high performance improvements to NAS-LU and Spec-art with certain input sizes. For instance, when the input size is 70, NAS-LU achieves an improvement of 31.7% relative to uncontrolled cache sharing. However, unlike NAS-CG, with a range of different input sizes object-level cache partitioning are not always useful. For example, when the cache size is 2MB, with the range of our input sizes, cache partitioning is not found profitable for NAS-LU and thus the default shared cache is used by the partition decision maker. The above differences are because the majority of NAS-LU and Spec-art's data references are to hot objects, not to any hog object as with NAS-CG. With a given configuration, the thrashing effect from accessing multiple hot objects is significant with a certain range of input

sizes but does not persist across inputs. As the result, in NAS-LU partition decisions vary with given input sizes and cache sizes but always only involve hot objects from arrays u, rsd, a, b, c and d. Similarly, partition decisions with Spec-art also only use three hot heap objects in our experiments.

Results in Figs. 8 and 9 also show that in some cases we may conduct object-level cache partitioning that actually increases the cache miss rate by up to 6%. However, performance degradation observed is never over 2%. We believe this comes from inaccuracies in modeling shared cache misses. In our framework, we implicitly assume an in-order processor model and do not model the hardware prefetching effect. While this issue deserves further investigation, as a practical solution, we can modify the cache partitioning algorithm such that unconstrained cache sharing is chosen when the predicted L2 miss reduction is below a threshold (e.g. 10%).

8 Related Work

Many approaches have been proposed to overcome the cache contention problem by partitioning the shared cache at the thread or process level. Most proposed approaches added cache partitioning support at the micro-architecture level [15, 25, 18]. Several studies highlighted the issues of QoS and fairness [12, 17, 13, 11, 6]. There have been several studies on OS-based cache partitioning policies and their interaction with the micro-architecture support [19, 8]. Our work differs from these studies in that we focus on object-level cache partitioning and do not require any new architectural support. There has been prior work using OS scheduling to reduce the effect of inter-thread contention. Fedorova et al. [9, 10] improved CMPs' throughput by judiciously selecting co-runners and time-schedules in the operating system. In comparison, in this paper we focus on improving the performance of sequential and parallel collaborative workloads that may not have candidate co-runners to choose. Because our proposed object-level cache partitioning framework works orthogonally to the above thread-level techniques including inter-thread cache partitioning and job pairing, our framework is complementary to these techniques when there are a large number of cores sharing the cache.

Understanding data locality is critical to performance optimization and is mostly through offline analysis of memory traces. Wu et al. [28] proposed a data trace representation in a object-relative form and demonstrated its application in computing memory dependence frequencies and stride patterns. Our work is related to this work though we use object-relative traces in a different way and do not attempt to store or compress data traces. Zhong et al. [29] studied whole-program data locality pattern recognition. We adapt their technique in our framework to detect locality patterns at the object level.

There have been several studies that improve data locality through using compiler and OS interaction [2, 22, 21, 5]. These studies focus more on avoidance of conflict misses which is not a significant problem to L2/L3 caches nowadays due to their high degrees of associativity. In comparison, we work on reduction of capacity misses of the shared cache. The problem is

irrelevant to cache's degree of associativity.

In the field of embedded systems, cache partitioning has been studied to reduce cache misses and power consumption. In particular, a recent study by Ravindran et al. [20] share many similarities with our work. Our work differs from their study and other related work in embedded systems in several aspects. First, our work is at the object level requiring no hardware support while their work is at the instruction level based on way partitioning. Second, we aim at reducing L2 cache capacity misses instead of L1 conflict misses, therefore complex techniques are used to detect locality patterns. Third, our framework does not employ static analysis, therefore no source level information is needed. Fourth, our goal is performance improvement instead of power reduction.

9 Conclusion and Future Work

We have designed and implemented a framework that partitions the cache at the object level, in order to improve program performance for both single-thread and parallel data-sharing programs. The framework consists of several major steps including profile generation, profile analysis, cache partitioning decision making and enforcement. Experimental results with a benchmark set from SPEC CPU2000 and OpenMP NAS benchmarks demonstrate the effectiveness of our system framework.

Ongoing and future work is planned along the following two directions. First, we will include small objects into consideration since small objects may reveal a collective locality behavior in many programs. A custom memory allocator supporting co-allocation is necessary for this purpose. Second, we plan to carefully study the potential of combining inter-object cache partitioning and inter-thread techniques such as job pairing.

References

[1] *NAS Parallel Benchmarks in OpenMP*. `http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html`.

[2] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *ASPLOS '96: Proceedings of the seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255, 1996.

[3] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 78–89, 1996.

[4] C. Caβcaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 150–159, 2003.

[5] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 139–149, 1998.

[6] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, 2007.

[7] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. *SIGPLAN Not.*, 36(5):286–297, 2001.

[8] S. Cho and L. Jin. Managing distributed, shared L2 caches through os-level page allocation. In *MICRO '06: Proceedings of the 39th International Symposium on Microarchitecture*, pages 455–468, 2006.

[9] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 26–26, 2005.

[10] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, 2007.

[11] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–22, 2006.

[12] R. Iyer. CQoS: a framework for enabling qos in shared caches of cmp platforms. In *ICS '04: Proceedings of the 18th annual International Conference on Supercomputing*, pages 257–266, 2004.

[13] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[14] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multi-core cache partitioning: Bridging the gap between simulation and real systems. In *HPCA '08: Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

[15] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 176, 2004.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[17] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. volume 35, pages 57–68, 2007.

[18] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO '06: Proceedings of the 39th International Symposium on Microarchitecture*, pages 423–432, 2006.

[19] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 2–12, 2006.

[20] R. Ravindran, M. Chu, and S. Mahlke. Compiler-managed partitioned data caches for low power. In *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 237–247, 2007.

[21] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 38–49, 1998.

[22] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, pages 155–164, 1999.

[23] Standard Performance Evaluation Corporation. *SPEC CPU2000*. `http://www.spec.org`.

[24] R. Sugumar and S. Abraham. Efficient simulation of multiple cache configurations using binomial trees. Technical report.

[25] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[26] G. Taylor, P. Davies, and M. Farmwald. The TLB slice–a low-cost high-speed address translation mechanism. In *Proc. ISCA'90*, pages 355–363, 1990.

[27] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, page 199, 2002.

[28] Q. Wu, A. Pyatakov, A. Spiridonov, E. Raman, D. W. Clark, and D. I. August. Exposing memory access regularities using object-relative memory profiling. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 315, 2004.

[29] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Trans. Computers*, 56(3):328–343, 2007.