Soft-OLP: Improving Hardware Cache Performance Through Software-Controlled Object-Level Partitioning

Qingda Lu[1], Jiang Lin[2], Xiaoning Ding[1], Zhao Zhang[2], Xiaodong Zhang[1], P. Sadayappan[1]
[1] Dept. of Computer Science and Engineering [2]Dept. of Electrical and Computer Engineering
The Ohio State University Iowa State University
{luq,dingxn,zhang,saday}@cse.ohio-state.edu {linj, zzhang}@iastate.edu

***Abstract*—Performance degradation of memory-intensive programs caused by the LRU policy's inability to handle weak-locality data accesses in the last level cache is increasingly serious for two reasons. First, the last-level cache remains in the CPU's critical path, where only simple management mechanisms, such as LRU, can be used, precluding some sophisticated hardware mechanisms to address the problem. Second, the commonly used shared cache structure of multi-core processors has made this critical path even more performance-sensitive due to intensive inter-thread contention for shared cache resources. Researchers have recently made efforts to address the problem with the LRU policy by partitioning the cache using hardware or OS facilities guided by run-time locality information. Such approaches often rely on special hardware support or lack enough accuracy. In contrast, for a large class of programs, the locality information can be accurately predicted if access patterns are recognized through small training runs at the data object level.**

To achieve this goal, we present a system-software framework referred to as Soft-OLP (Software-based Object-Level cache Partitioning). We first collect per-object reuse distance histograms and inter-object interference histograms via memory-trace sampling. With several low-cost training runs, we are able to determine the locality patterns of data objects. For the actual runs, we categorize data objects into different locality types and partition the cache space among data objects with a heuristic algorithm, in order to reduce cache misses through segregation of contending objects. The object-level cache partitioning framework has been implemented with a modified Linux kernel, and tested on a commodity multi-core processor. Experimental results show that in comparison with a standard L2 cache managed by LRU, Soft-OLP significantly reduces the execution time by reducing L2 cache misses across inputs for a set of single- and multi-threaded programs from the SPEC CPU2000 benchmark suite, NAS benchmarks and a computational kernel set.

***Keywords*-Cache Partitioning, Software-Controlled Caching, Reuse Distance, Page Coloring**

I. Introduction

The performance gap between the processor and DRAM has been increasing exponentially for over two decades. With the arrival of multicore processors, this "memory wall" problem is even more severe due to limited off-chip memory bandwidth [1]. Reducing cache misses is a key to achieving high performance on modern processors. In this study, we design and implement effective software methods to address a weakness of LRU-based hardware management of shared last-level caches in modern processors. Most cache designs are based on the LRU replacement policy (its approximations in practice). While the LRU policy offers high performance for workloads with strong data locality, it does not identify weak-locality accesses with long reuse distances and thus often incurs cache misses with memory-intensive workloads due to mis-replacements (weak-locality accesses pollute the cache by evicting strong locality blocks). Previous studies have shown this significant problem with the LRU policy, and proposed a few solutions. Despite their design differences, these hardware proposals follow one of two directions: (1) Hybrid replacement schemes such as [2], [3] that dynamically select from multiple replacement policies based on runtime information, and (2) Cache bypassing approaches [4], [5] that identify weak-locality accesses and place them in a dedicated cache (bypass buffer) to avoid cache pollution. These approaches share one common limitation: they introduce both storage overheads and latency penalties, being difficult to be adopted by processors in practice. Instead of taking transparent hardware solutions, some commercial designs have chosen to partially address the problem by providing special caching instructions such as the *non-temporal store* instruction on Intel architectures [6]. However such hybrid approaches are architecture-specific and limited to certain types of memory accesses such as streaming writes, and it is often not feasible for the programmer or the compiler to produce optimized code versions across different cache configurations and program inputs.

To address the above problem with the LRU policy, we propose a software framework that partitions the cache at the data object level to reduce cache misses for sequential programs and data-sharing parallel programs. Our approach is motivated by the following observation: *many weak-locality accesses at the whole program level may have strong locality within one or a few data objects.* A memory location is said to have weak locality if the reuse distance (number of distinct memory references between successive accesses to the given location) is greater than the cache capacity. By judiciously segregating objects that have interfering access patterns, we can exploit strong locality within objects in their own cache regions using the conventional LRU policy. In this paper we focus on partitioning the last-level cache space among large global and heap objects for high-performance scientific applications. For a given

program, the proposed framework first generates program profiles (*per-object reuse distance histograms* and *inter-object interference histograms*) for global and heap objects using training inputs. Based on the training profiles it then detects the patterns of the program profiles. When the program is scheduled to run with an actual input, the framework predicts its locality profile based on the detected locality patterns. With actual cache configuration parameters an object-level cache partitioning decision is then made to reduce cache misses. This framework is referred to as Soft-OLP (Software-based Object-Level cache Partitioning). We have implemented Soft-OLP with a modified Linux kernel to enforce partitioning decisions by carefully laying out data objects in the physical memory. By running a set of memory-intensive benchmarks on a commodity chip multiprocessor (CMP), we show that Soft-OLP is very effective in handling two common and difficult scenarios in practice. First, when there are *cache hogs*, i.e. objects with unexploitable data locality, Soft-OLP segregates them from the rest of the objects to improve whole-program locality. Second, when contention between multiple strong-locality objects incurs cache thrashing, Soft-OLP alleviates the problem by constraining the effective cache capacity to a subset of the objects.

The contributions of the paper are three-fold. First, to the best of our knowledge, Soft-OLP is the first work that uses object-level cache partitioning to reduce capacity misses for both sequential programs and OpenMP-style parallel programs. In comparison, previous related studies [7], [8], [9] 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. Soft-OLP is also independent of compiler implementations by working with binary executables. Third, Soft-OLP 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 present a motivating example in Section II. We then give an overview of Soft-OLP in Section III. In Section IV, we introduce object-level program locality profiles used in this paper. In Sections V, VI and VII, we describe how we generate program profiles, analyze generated profiles and make partitioning decisions based on the analysis results. We evaluate the effectiveness of our approach in Section VIII on a commodity CMP using programs from a computational kernel set, SPEC CPU2000 benchmarks and NAS benchmarks. We discuss related work in Section IX and present our conclusions in Section X.

II. A Motivating Example

Here we use the conjugate gradient (CG) program from the 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. There are also code pieces not shown in Fig. 1 due to the space limitation for this paper. These code pieces access arrays iv,v,$acol$,$arow$, x,r,q,z,$aelt$ in addition to the arrays shown in Fig. 1. In CG, the majority of the memory 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 repeatedly evicted from cache before their next uses, due to the streaming accesses on arrays a and $colidx$. As the result of this thrashing effect caused by accessing arrays a and $colidx$, CG often reveals a streaming access pattern and has a very high miss rate in cache. Without special code/data treatment based on domain knowledge, general compiler optimizations, such as loop tiling, cannot be applied in this case because of the irregular nature of this program — there is indirection in array accesses.

```
...//other code
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.

The caching inefficiency problem occurs in CG because the conventional LRU cache replacement policy does not distinguish strong- and weak-locality accesses and thus is unable to treat them differently. Since cache replacement decisions are made at the whole-system level, any data reuse with a reuse distance greater than the cache size cannot be hit in the cache. The CG case is an example of variable locality strengths among different data objects, which can not be distinguished and handled properly by LRU. If we allow the cache space to be partitioned between data objects, we would be able to allocate variable cache sizes to different objects based on their locality strengths, well utilizing the limited cache space and minimizing cache misses. With CG, there are different ways to reduce and even completely eliminate capacity misses on strong-locality array p without increasing cache misses on the other objects. One approach is to protect p in an exclusive cache space and leave the remaining cache capacity for the remaining data objects. Alternatively, we can divide the cache such that the minimum cache quota is given to weak-locality arrays $colidx$ and a. This optimization is not limited to single-thread performance. Even when the code is augmented with OpenMP directives, with a shared cache the object-level partitioning decisions should still reduce capacity misses, since memory accesses from different processor cores collectively reveal the same pattern as with sequential execution. If we allocate a very small cache quota for arrays $colidx$ and a and co-schedule CG with other programs, it no longer exhibits a streaming access pattern that significantly interferes with its co-runners, so

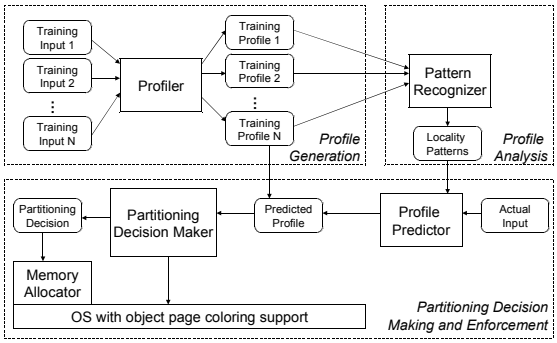


Figure 2. Overall structure of the object-level cache partitioning framework.

that high throughput can be achieved with judicious inter-thread cache partitioning. In this paper, we focus on object-level cache partitioning and defer the combination of inter-object and inter-thread cache partitioning to future work.

III. Overview of the Approach

The CG example in Fig. 1 demonstrates the benefits of partitioning the cache space at the object level. In this paper the term *object* is defined as an allocated region of data storage and used interchangeably with *variable*. Note that this definition is not equivalent to its usage in object-oriented programming. We partition the last-level cache space among global and heap objects for high-performance scientific applications. There are two reasons for this decision. First, high-performance scientific applications often have relatively regular memory access patterns and high data reuse ratios, which makes object-level cache partitioning possible and profitable. Second, in these programs, the majority of the memory accesses and cache misses are on a limited number of global and heap objects. In order to partition the last-level cache space among data objects, we need to answer the following questions: (1) How can we capture data reuse patterns at the object level, across cache configurations and program inputs? (2) How can we capture the interference among the data objects that share and compete for cache space? (3) How can we identify critical objects as partitioning candidates? (4) How can we make quick object-level partitioning decisions with a different program input? (5) What system support is needed to enforce cache partitioning decisions?

To answer the above questions, we propose a system framework called Soft-OLP that detects a program's data reuse patterns at the object level, through memory profiling and pattern recognition, and enforces partitioning 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 several small training inputs, we capture memory accesses in an object-relative form through binary instrumentation. We obtain per-object reuse distance histograms and inter-object interference histograms for data objects. These histograms are program profiles with training inputs that are to be used to predict the program's data access and reuse patterns.
2) *Profile Analysis.* Based on program profiles from training runs, we detect the patterns of the program's per-object data reuse, object sizes and access frequencies as polynomial functions, using a pattern recognition algorithm based on the work in [10].
3) *Cache Partitioning Decision Making and Enforcement.* When the program is scheduled to run with an actual input, we predict its per-object reuse distance histograms and inter-object interference information with detected access patterns. We then categorize data objects as being "hot", "hog", "cold" or "other". Using this classification, we follow a heuristic algorithm to make an object-level cache partitioning decision so that "hog" objects do not prevent us from exploiting the locality of "hot" objects and the contention between "hot" objects is alleviated. Such a partitioning decision is finally enforced on commodity CMPs with an OS kernel that supports *page coloring* [11], [12] at the object level.

IV. Object-Level Program Locality Profile

With a given input, we model a program's data locality at the object level with a locality profile. An object-level program locality profile has two components: an object-relative locality profile consisting of per-object reuse distance histograms and an inter-object interference profile including inter-object interference histograms.

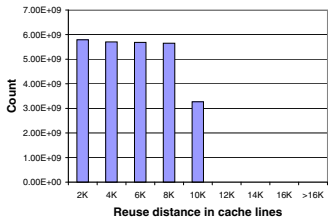
A. Modeling Per-Object Temporal Locality

In this paper we focus on temporal locality at the cache line granularity because spatial locality is automatically modeled by viewing a complete cache line as the basic data unit. While this approach may appear to affect the proposed framework's generality, it is not a problem because Soft-OLP aims at detecting the data locality patterns of a given program binary that works across 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 micro-architectures all use 64-byte L2 cache lines. In this paper we exploit data locality in the last-level on-chip cache. As the last-level cache on a modern processor often has a very high degree of associativity, the impact of conflict misses is not significant and we therefore model the cache as being fully associative.

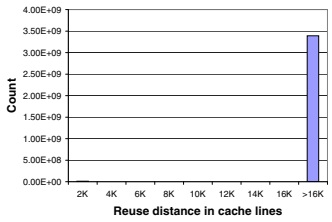
Temporal locality is modeled using *reuse distance* (i.e. stack distance) [13], defined as the number of distinct references between two consecutive references to the same data unit. Since in this paper data locality is modeled with the cache line as the basic unit, reuse distance refers to the number of distinct cache lines accessed between two consecutive references to the same cache line. As it is not feasible to record the reuse distance between each

data reuse, a histogram is used to summarize the temporal locality. In a reuse 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 reuse distances fall into this range. R_{max} is the largest cache capacity considered in terms of cache lines. While reuse distances have primarily been used for analysis at the whole program level, here we model temporal locality via a reuse distance histogram D_A for accesses within each object A. A *per-object reuse distance histogram* maintains absolute reuse counts instead of percentages, because with different inputs, objects have varying weights over the whole-program data space. In one object, consecutive accesses on the same cache line have zero reuse distances. We do not model such data reuses in per-object reuse distance histograms because such reuses are mostly handled by the L1 cache while we optimize last-level cache accesses in this paper.

As an example, Fig. 3 shows simplified per-object reuse distance histograms for objects p and a in CG with input class B, on recent Intel x86 architectures with 64-byte cache lines. While the majority of a's references share a large reuse distance that corresponds to a's size in cache lines, 0.35% of accesses on a have a very small reuse distance, in code not shown in Fig. 1. In comparison, p has relatively short reuse distances across several data ranges resulting from p's random access pattern.



(a) Per-object reuse distance histogram D_p for p.



(b) Per-object reuse distance histogram D_a for a.

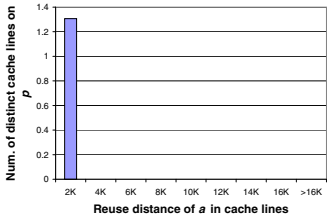
Figure 3. Per-object reuse distance histograms for objects p and a in CG, generated with input class B.

B. Modeling Inter-Object Interference

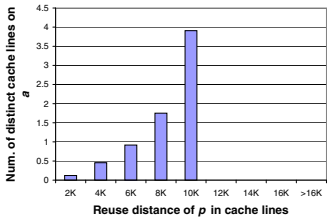
We cannot capture a program's locality behavior with only per-object temporal reuse information. For example, two programs can have the same per-object temporal reuse profiles for arrays A and B, with one accessing A and B in separate program sections and the other interleaving accesses on the two arrays. To compose the temporal locality information of individual objects and examine the whole-program locality behavior, 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. Similar to temporal locality modeling, we extend the above definition to the cache line level. Note that inter-object interference 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 , $I_{A,B}$ often vary with changes on A's reuse distances. Therefore we use a histogram to summarize inter-object interference $I_{A,B}$. In such a histogram, data ranges correspond to those in reuse distance histogram D_A and the value over each range represents the average number of distinct cache lines accessed on object B per distinct cache line accessed on A, between A's reuse with a reuse distance in the range.

For example, Fig. 4 shows the simplified inter-object interference histograms for $I_{a,p}$ and $I_{p,a}$ for CG with input class B. Fig. 4(a) shows interference from object p to object a. $I_{a,p}$ is nearly zero although most accesses on a and p are interleaved. This is due to the fact that between any temporal reuse of a, all the elements in a are accessed only once while elements in the smaller vector p have much higher reuse counts. The exception is with p's accesses having small reuse distances. As mentioned above, this corresponds to an infrequently executed loop. In comparison to $I_{a,p}$, Fig. 4(b) shows that $I_{p,a}$ increases with the reuse distances of p. When p's reuse distances are between 8000 and 10000 cache lines, the interference from a to p is as high as 3.91. This means that between two data accesses on p with reuse distance 9000 there are on average $9000 \times 3.91 = 35190$ distinct cache lines accessed on a. We can see from $I_{p,a}$ and D_p that object a significantly interferes with object p's temporal reuse since a large portion of p's references involve large interference values.



(a) Inter-object interference histogram $I_{a,p}$.



(b) Inter-object interference histogram $I_{p,a}$.

Figure 4. Examples of inter-object interference histograms for CG, generated with input class B.

C. Cache Miss Estimation

As we will show in Section VII, the key operation in the partitioning decision algorithm is the estimation of shared cache misses on a set of objects for a given cache size. This essentially relies on merging of the per-object reuse distance histograms and inter-object interference histograms of a set of objects. Once we have such a combined reuse distance histogram D_S, all the accesses with reuse distances greater than the given cache capacity are predicted as misses. The profile combination process is as follows.

- For a reference to object i, its reuse distance in the combined object set S is computed as:

$$dist_S = dist_i + dist_i \sum_{j \in S-\{i\}} I_{i,j}[dist_i]$$

  Therefore, in an object-level reuse distance histogram, each bar is shifted to the right by a distance determined by its range and interference from the other objects in the group. The resulting reuse distance histogram for the object group is the combination of these individual right-shifted histograms.
- For an object group S, its combined inter-object interference with an object k outside this group is computed as:

$$I_{k,S}[dist_k] = \sum_{j \in S} I_{k,j}[dist_k]$$

For example, based on the per-object reuse distance histograms and inter-object interference histograms shown in Figs. 3 and 4, we combine object-level locality information of a and p and obtain a combined reuse distance histogram $D_{\{a,p\}}$ in Fig. 5. If we assign 1MB (16K cache lines) to the object group consisting of a and p, the total number of cache misses on this object group is estimated to be $\#accesses_{\{a,p\}}(reuseDist > 16384) = 1.23 \times 10^{10}$, where $\#accesses_{obj}(reuseDist > R)$ denotes the number of references to object obj with reuse distances larger than R.

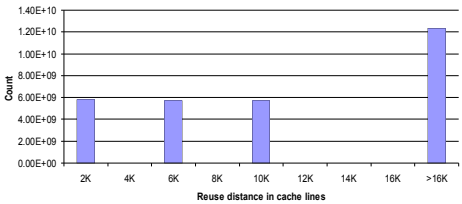


Figure 5. The combined reuse distance histogram for object group $\{a, p\}$ in CG with input class B.

V. Profile Generation

Fig. 6 illustrates how a program profile with a training input is generated. There are three important components used in profile generation: *object table*, *custom memory allocator* and *memory profiler*.

Object Table: 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

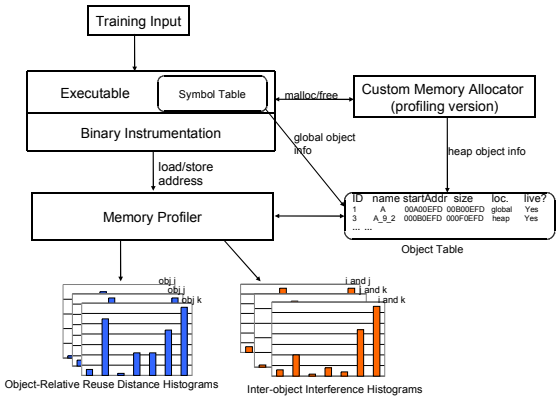


Figure 6. Program profile generation with a training input.

information stored in the object table includes object identifier, name, starting address, size, location and liveness. In this paper we focus on global and heap objects therefore an object's location 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 obj_0 that includes all the data not explicitly profiled. 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. Because heap objects may have overlapping address ranges due to their different life cycles, a raw address can be found within multiple heap objects. We keep track of each object's liveness to avoid this problem. In this way during profile generation each memory access only affects a live object's per-object temporal locality information and inter-object interference information.

Custom Memory Allocator: 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. During profiling runs memory management requests are redirected to the memory profiler and heap objects' life cycles are tracked in the object table.

Memory Profiler: The memory profiler controls the profiling process and starts a training run by updating the object table with global object information from the executable'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 PIN [14] tool that inserts instruction and object probes before every instruction accessing the memory. The core components in the profiler are a set of *reuse distance profilers* and an *inter-object interference counter table*. A reuse distance profiler is used to track the reuse distances of an object and implemented with a hash table and a Splay tree following the approach in [15]. The inter-object interference

counter table is two-dimensional and used to track inter-object interference between temporal reuses. As in high-performance scientific programs the majority of the accesses are on large global and heap objects and it is not feasible to include all objects in cache partitioning decision making, we only track objects larger than a threshold (2KB) in memory profiling. Small objects are merged into a special object obj_0. The memory profiling algorithm used is summarized in Algorithm 1.

Algorithm 1 The memory profiling algorithm.

Require: $objProfiler.trace(addr)$ returns the reuse distance of access $addr$ on object obj and $hist.sample(d, n)$ collects n data with value d into histogram $hist$.

1: $tracedAddr[0..objNum-1] \leftarrow 0$
2: **for** each memory reference with raw address $addr$ **do**
3: Search the object table for such a live object obj that $obj.startAddr \leq addr$ and $obj.startAddr + obj.size \geq addr$
4: **if** obj exists **then**
5: $(objID, offset) \leftarrow (obj.ID, (addr - obj.startAddr)/CacheLineSize)$
6: **else**
7: $(objID, offset) \leftarrow (0, addr/CacheLineSize)$
8: **end if**
9: $reuseDist \leftarrow reuseDistProfiler[objID].trace(offset)$
10: $objReuseDistHistogram[objID].sample(reuseDist, 1)$
11: **if** $tracedAddr[objID] = 0$ **then**
12: $tracedAddr[objID] \leftarrow addr$
13: continue to process next memory access
14: **end if**
15: **if** $tracedAddr[objID] = addr$ and $reuseDist \neq 0$ **then**
16: $tracedAddr[objID] \leftarrow 0$
17: **for** each i where $i \neq objID$ **do**
18: **if** $interferenceCounter[objID][i] \neq 0$ **then**
19: $interferenceCountHistogram[obj][i].sample(reuseDist, interferenceCounter[objID][i]/reuseDist)$
20: $sampleCounterHistogram[obj][i].sample(reuseDist, 1)$
21: $interferenceCounter[objID][i] \leftarrow 0$
22: **end if**
23: **end for**
24: **end if**
25: **for** each j where $j \neq objID$ **do**
26: **if** $reuseDist > interferenceCounter[j][objID]$ **then**
27: $interferenceCounter[j][objID] \leftarrow interferenceCounter[j][objID] + 1$
28: **end if**
29: **end for**
30: **end for**
31: **for** each obj i **do**
32: **for** each obj j where $i \neq j$ **do**
33: **for** each range k **do**
34: $inteferenceHistogram[i][j][k] \leftarrow \frac{interferenceCountHistogram[i][j][k]}{sampleCounterHistogram[i][j][k]\ range[k]}$
35: **end for**
36: **end for**
37: **end for**

As the cost of binary instrumentation is high, for each profiled object, we optimize the memory profiler by collecting reuse distances for a portion of the accesses and then estimate the complete reuse distance histogram based on the sampled profile. Note that every access is still recorded for later reuse distance collection and inter-object interference counting. We have tried different sampling ratios and found in practice 10% as the best trade-off between accuracy and profiling cost. Compared with complete profiling, the error introduced by sampling is less than 2% in our experiments. By sampling 10% of reuse distances and employing several other optimizations, our current implementation of the memory profiler has a slowdown of 50 to 80 times. This profiling cost, while still seemingly high, can be easily amortized over the lifetime of an executable, as the program complexity is often quadratic or cubic and profiling runs are made with much smaller problem sizes than actual runs that is of interest to optimize.

VI. Profile Analysis

After at least two program profiles with different training inputs are obtained from the profiling process, per-object data locality patterns are detected following an algorithm similar to the approach by Zhong et al. [10]. The idea is as follows. Each per-object reuse distance histogram is divided into n small groups. With two histograms p_1 and p_2 corresponding to different input sizes, we need to find a pattern function f_k to fit each formed group in two profiles, $g_{1,k}$ and $g_{2,k}$, for k from 1 to n. f_k matches average reuse distances $d_{1,k}, d_{2,k}$ in group k such that $f_k(x_1) = d_{1,k}$ and $f_k(x_2) = d_{2,k}$, where x is a program parameter. In this paper, for f_k we consider simple polynomial functions in the form of $f_k(x) = a_k + b_k x^m$, where $m = 0, 1, 2, 3, 4$ and a_k,b_k are constants. In order to detect the locality pattern function f_k for each group k, we first pick an exponent m such that x_1^m/x_2^m is the closest to $d_{1,k}/d_{2,k}$. Then we decide a_k and b_k by solving equations $d_{1,k} = a_k + b_k x_1^m$ and $d_{2,k} = a_k + b_k x_2^m$. Following the above process an object's locality pattern is summarized as a list of polynomial functions $f = \{f_1, f_2, ..., f_n\}$. Note that our choice of using polynomial functions in locality pattern recognition differs from the approach by Zhong et al. that takes functions such as square root. This is mainly because we use one program parameter instead of the object size as the variable. We also follow procedures similar to locality pattern recognition to detect the patterns of the data access volume and the data size of each object as well, as absolute counts instead of percentages are kept in histograms. With all the above program patterns detected, given a new input we can construct its per-object reuse distance histograms without tracing the actual execution. For inter-object interference histograms, we detect their patterns similar to object-level reuse pattern recognition. As an example, after pattern recognition using two training profiles, we find that in CG the majority of array p's reuse distances grow linearly with program parameter na while the rest of the reuse distances keep a constant pattern. In contrast, although $colidx$ and a's reuse distances actually grow with the input size, they are simply predicted as large constants as they exceed the maximum cache capacity considered (8MB) even with training inputs. We also predict that inter-object interference $I_{a,p}$ is constant 0 within most ranges while $I_{p,a}$ mostly follows a linear pattern.

VII. Cache Partitioning Decision Making and Enforcement

When the program is scheduled to run with an actual input, we first predict the program profile including per-object reuse distance histograms and inter-object interference histograms with patterns recognized during profile

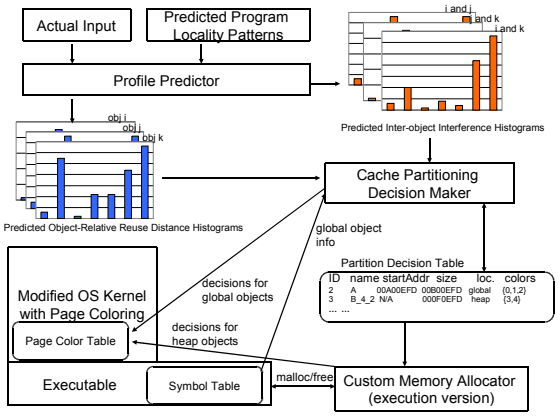


Figure 7. Cache partitioning decision making and enforcement with an actual input.

analysis. The partitioning 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. Fig. 7 illustrates the cache partitioning decision making and enforcement process. This phase of Soft-OLP 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 used in profile generation, the memory allocator used here reads the object table to check and enforce cache partitioning decisions but never updates the table. The rest of this section discusses in detail the cache partitioning mechanism and the partitioning decision-making algorithm.

A. Partitioning Enforcement

The most straightforward way to enforce cache partitioning decisions is through hardware support. Cache can be partitioned at different granularities 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 based on a well accepted OS technique called *page coloring* [11]. 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, thus the 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. As the page size is set to 4KB, we can break the L2 cache to 64 colors (cache size / (page size $\times$ 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 the page color that a range of virtual pages 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 the user level. These system calls are used by the partitioning decision maker and the memory allocator to enforce cache partitioning decisions A special value ($0xFF$) is reserved for page color table entries to indicate the use of the default random page mapping policy of a unmodified Linux kernel.

While we can minimize the storage cost by maintaining a page color table with each entry corresponds to an object of interest, in the 32-bit systems used in the experiments we keep the page color information for each virtual page. The number of table entries is decided by the maximum number of virtual pages, which is 2^{20} in the 32-bit systems. In our implementation, each table entry occupies one byte to represent the color assigned to a virtual page and therefore the page color table incurs no more than 1MB space overhead for each process family in 32-bit systems. Therefore our implementation has a negligible space overhead. As all the decisions are made statically at user level, there is no run-time overhead in the kernel.

B. Partitioning Decision Making

Preprocessing — Object Categorization and Object Clique Search: With the predicted program profile and the actual cache configuration, we put objects into four categories based on their temporal locality profiles:

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 a 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. In our experiments, we set T_{cold} as 1%.
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 $\frac{\#accesses_{obj}(reuseDist \leq cacheSize)}{\#accesses_{obj}} < T_{hog}$, then it is a hog object. Note that $cacheSize$ is in terms of cache lines. $\#accesses_{obj}(reuseDist \leq R)$ refers to the number of references to object obj with reuse distances less than or equal to R and we have $T_{hog} = 2\%$.
3) *Hot objects* are the objects with high temporal locality. For an object obj, if we have

$\frac{\#accesses_{obj}(reuseDist \leq cacheSize)}{\#accesses_{obj}} > T_{hot}$ and if obj is not a cold object, then it is a hot object. We have $T_{hot} = 10\%$.

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 I. Because the majority of the objects are categorized as cold objects, it significantly simplifies the partitioning decision-making process.

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 I
CATEGORIZING OBJECTS IN CG'S PROFILES.

A set of objects and their interference relationships can be viewed as a graph with vertices representing objects and unweighted edges representing interferences between objects. Based on the graph representation, we enumerate the cliques among hog objects. As we will show in the partitioning decision algorithm, 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 small in real programs. For example, CG has only one hog object clique that includes a and $colidx$.

The Partitioning Decision-Making Algorithm: The partitioning decision-making algorithm finds a cache partition among objects with predicted per-object reuse distances, inter-object interferences and cache configuration information. With a data-sharing parallel program, the algorithm uses its sequential counterpart to approximate its data access patterns. The partitioning decision-making algorithm consists of four major steps. (1) To simplify late parts of the algorithm, we first merge the reuse distance histograms and inter-object interference histograms of *cold* and *other* objects with those of the special object 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 a physically addressed cache and the physical memory are co-partitioned by the page color-based partitioning 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 for the best partitioning decision in a brute-force way, a heuristic is employed to maximize the benefit-cost ratio at every cache allocation step until there is no further benefit of cache partitioning. (4) As an important optimization, once there is no additional benefit from segregating complete objects, we attempt to divide a remaining object into two segments and keep one segment in cache. In this way we exploit temporal locality even if object-level reuse distances are greater than cache size,

Algorithm 2 The partitioning decision-making algorithm.

Require: misses(c, S) returns the cache miss number of object group S with cache allocation c, using predicted per-object reuse distance histograms and inter-object interference histograms.
1: STEP 1. (Merge cold and other objects)
2: All cold and other objects are merged into obj_0
3: STEP 2. (Find the minimum cache space for hog objects)
4: Find the clique in hog objects with the largest memory requirement mem_{hogs}
5: $hogCacheColors \leftarrow \lceil mem_{hogs}/(totalMemory/\#pageColors) \rceil$
6: $cacheColors \leftarrow totalCacheColors - hogCacheColors$
7: STEP 3. (Heuristic-based cache partitioning for hot objects)
8: $partitionedObjs \leftarrow \phi$, $objsLeft \leftarrow hotObjs$
9: **while** $objsLeft \neq \phi$ **do**
10: $bestBenefitCost \leftarrow 0$
11: **for** each object obj in $objsLeft$ **do**
12: **for** $colors = \lceil mem_{obj}/(totalMemory/\#pageColors) \rceil$ to $cacheColors$ **do**
13: Try to find non-conflicting colors in assigned colors to $partitionedObjs$. The objects already assigned these colors should have no interference with obj.
14: $cost \leftarrow (colors - nonConflictingColors)$
15: $benefit \leftarrow misses(cacheColors, objsLeft)$ - $(misses(cacheColors - cost, objsLeft - obj) + misses(colors, obj))$
16: **if** $benefit/cost > bestBenefitCost$ **then**
17: $bestBenefitCost \leftarrow benefit/cost$
18: $(obj_{best}, colors_{best}) \leftarrow (obj, colors)$
19: **end if**
20: **end for**
21: **end for**
22: **if** $(obj_{best}, colors_{best})$ is not empty **then**
23: $partitionedObjs \leftarrow partitionedObj \cup \{obj_{best}\}$
24: $objsLeft \leftarrow objsLeft - \{obj_{best}\}$
25: $cacheColors \leftarrow cacheColors - colors_{best}$
26: **else**
27: Break from the while loop
28: **end if**
29: **end while**
30: STEP 4. (Partial-object cache partitioning)
31: $objsLeft \leftarrow$ objects in $objsLeft$ with object-size reuse distances
32: Repeat lines 9-29, at each iteration candidate object obj is tentatively divided into two objects obj_1 and obj_2 where $obj_1.size = (colors \times cacheSize)/\#pageColors, obj_2.size = obj.size - obj_1.size$

a case where both the traditional LRU policy and object-level cache partitioning restricted to complete objects cannot handle. We only apply this optimization to objects whose reuse distances share the same pattern with their object sizes as we notice this pattern is very common in memory-intensive scientific programs. More complex cases can be handled by further extending this approach. For example, an object of n^3 elements can be divided into $2n$ segments if its reuse distance n^2 is greater than the cache size. However, such complicated extensions have not been implemented. The complete partitioning decision making algorithm is summarized in Algorithm 2.

Inaccuracy exists in the data locality model as it is impractical to include architectural complexities such as prefetching and out-of-order execution. Therefore we may mistakenly choose a cache partition that does reduce cache misses in practice. As a practical solution, at each iteration of steps 3 and 4 we accept the partition only if the predicted last-level miss reduction is above a threshold compared to the previous iteration. Otherwise we stop further applying cache partitioning. In practice, we set this threshold to be 5%.

For example, when we apply Algorithm 2 to CG with a given input on a machine with a 4MB L2 cache, the

decision-making algorithm always first segregates hog objects a and $colidx$ with the rest of the objects. Then our algorithm stops trying to partition the remaining cache space as it cannot significantly reduce cache misses with the remaining objects.

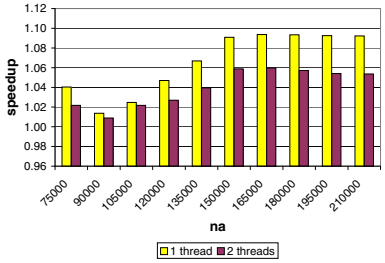
VIII. Experimental Results

We used Soft-OLP to improve the L2 cache performance of a commodity system. In this section we present the experimental results.

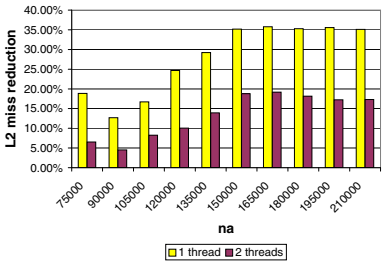
Experimental Environment: We conducted experiments on a Dell PowerEdge 1900 workstation with two quad-core 2.66GHz Xeon X5355 Processors and 16GB physical memory with eight 2GB dual-ranked Fully Buffered DIMMs (FB-DIMM). Each X5355 processor has two pairs of cores and cores in each pair share a 4MB, 16-way set associative L2 cache. Each core has a private 32KB L1 instruction cache and a private 32KB L1 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 of cache space. Without incurring page swapping, the maximum physical memory mapped to a page color is 512MB on this machine. We used *pfmon* [16] to collect performance statistics such as L2 cache misses.

Benchmark Selection: We selected a set of scientific programs from an OpenMP implementation of NAS benchmarks [17], the SPEC CPU2000 benchmark set [18] and a suite of computational kernels [19]. These benchmarks include *jacobi2d*, *stencil3d*, *adi* and *fdtd2d* from the kernel suite, four benchmarks *apsi*,*swim*,*art* and *mgrid* from SPEC CPU2000 that were also used in a related study [20] and five benchmarks *CG*, *LU*, *BT*, *FT* and *SP* from NAS benchmarks version 2.3. We excluded *twolf* and *vpr* that were also used in [20] and *EP* and *IS* in NAS benchmarks, because they are not scientific programs and optimizing these programs is out of the scope of this study. We also did not include *MG* from the NAS benchmarks because it is identical to *mgrid* from SPEC CPU2000. The selected benchmarks were compiled with Intel C/Fortran Compiler 10.1 using the "-fast" flag. With OpenMP problems, the "-openmp" flag is also used.

The efficiency of object-level cache partitioning essentially relies on program information that distinguishes different data objects. In some cases, to fully test our framework, we had to make source code changes in the selected benchmarks due to some limiting factors. First, some Fortran programs use common blocks which makes global objects in a common block indistinguishable. For this reason we modified array declarations in *swim* such that every global object was only in one common block. We also chose to use a C implementation of the NAS benchmarks instead of

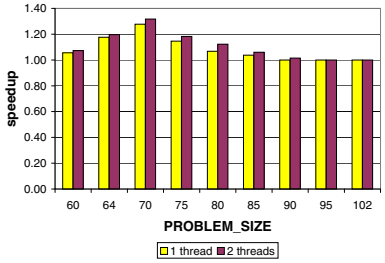


(a) Speedups

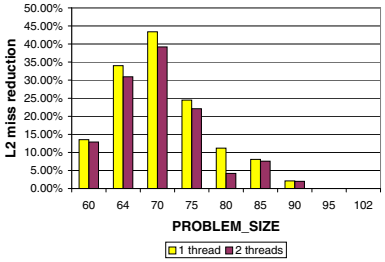


(b) L2 cache miss reduction

Figure 8. *CG* with object-level cache partitioning in comparison to standard LRU caching.



(a) Speedups



(b) L2 cache miss reduction

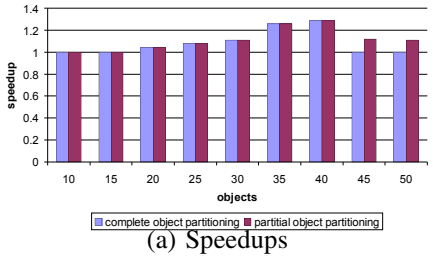
Figure 9. *LU* with object-level cache partitioning in comparison to standard LRU caching.

the original Fortran programs to avoid this complication. 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 and memory requirement before and after such simple changes. Third, some legacy programs use a large fixed-size array as the work array for memory management. *apsi* from SPEC CPU2000 is such an example. With these programs, it is not feasible to distinguish accesses on different objects. Therefore as shown in Table II, we were not able to optimize *apsi*.

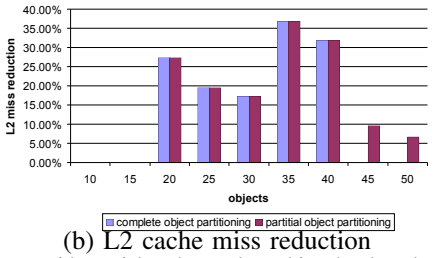
The resulting benchmark suite consists of thirteen programs, including five NAS benchmarks that run both sequen-

Benchmark	Input parameter	Input range	# of training inputs	# of actual inputs	Objects profiled	Object-level reuse distance patterns	# of threads	Max. speedup	Avg. speedup	Max. miss reduction	Avg. miss reduction	# of actual inputs with improvement
jacobi2d	N	[200,1200]	2	9	3	0,1,2	1	1.27	1.13	46.3%	25.2%	7
stencil3d	N	[50,100]	2	9	3	0,1,2,3	1	1.36	1.13	39.3%	19.6%	7
adi	NMAX	[200,1200]	2	9	4	0,1,2	1	1.03	1.01	24.7%	1.0%	8
fdtd2d	N=N1=N2	[200,1200]	2	9	4	0,1,2	1	1.91	1.25	78.0%	29.9%	8
swim	N=N1=N2	[128,1334]	2	7	15	0,1,2	1	1.03	1.01	2.1%	1.0%	2
art	objects	[10,50]	2	7	9	0,1	1	1.29	1.14	36.9%	21.3%	7
CG	na	[75000,210000]	2	8	15	0,1	1	1.09	1.08	35.7%	30.9%	8
							2	1.06	1.05	19.2%	15.4%	8
LU	PROBLEM_SIZE	[60,102]	2	7	9	0,1,2,3	1	1.28	1.08	43.4%	12.5%	5
							2	1.32	1.10	39.2%	10.4%	5
apsi	N=NX=NY=NZ	[64,112]	2	4	2	0,1,2,3	1	1	1	0	0	0
mgrid	2^{LM}	[32,128]	2	1	4	0,1,2,3	1	1	1	0	0	0
BT	PROBLEM_SIZE	[60,102]	2	7	15	0,1,2,3	1,2	1	1	0	0	0
FT	N=NX=NY=NZ	[64,512]	2	2	9	0,1,2,3	1,2	1	1	0	0	0
SP	PROBLEM_SIZE	[60,102]	2	7	15	0,1,2,3	1,2	1	1	0	0	0

Table II
CHARACTERISTICS AND EXPERIMENTAL RESULTS FOR SELECTED BENCHMARKS. FOR BREVITY 0,1,2,3 ARE USED TO REPRESENT CONSTANT, LINEAR, SQUARE AND CUBIC FUNCTIONS RESPECTIVELY.



(a) Speedups



(b) L2 cache miss reduction

Figure 10. *art* with partial and complete object-level cache partitioning in comparison to standard LRU caching.

tially and in parallel. Their characteristics are summarized in Table II. Table II also shows the range of the inputs used for each benchmark. If there is a reference input size for a benchmark, it was always used in our experiments. The two smallest inputs for each benchmark were used in training runs, while the remaining input sets were used for the actual runs. While the profiling time is non-trivial, after two program profiles are generated and analyzed, the time taken for the cache partitioning decisions is negligible – of the order of tens of milliseconds. In the experiments, we did not observe any increase in OS activities from the kernel modification. Table II also shows the number of objects profiled for each benchmark. It can be seen that the number of profiled objects in these scientific applications is quite small – up to 15 in the experiments. This indicates that it is feasible to effectively reason about data locality at the object level. Furthermore, the majority of the data accesses exposed by these objects are very regular and thus can be modeled using simple polynomial functions.

Table II shows performance and cache miss reduction data for our approach compared to that with a standard shared LRU-based L2 cache. For a given input, we ran each benchmark three times with the shared LRU-based L2 cache and three times with object-level cache partitioning. We then reported the speedup and cache miss reduction using the worst performance numbers with object-level cache partitioning and the best numbers with the shared LRU-based L2 cache. If the partitioning decision algorithm did not choose to partition the cache between objects, we simply set speedup to 1 and miss reduction to 0 as the shared LRU-based L2 cache was nevertheless used. In our experiments, we achieved performance improvements on eight benchmarks, with up to 1.91 speedup and up to 78% cache miss reduction (with *fdtd2d* when $N = 500$). In experiments involving these eight benchmarks, we improved 65 out of 80 total cases. We were unable to improve the performance of the five remaining benchmarks. In all the cases where we could not improve performance, the algorithm did not make any bad cache partitioning decision that caused performance degradation.

The inability of Soft-OLP to achieve performance improvement for a few programs is due to two reasons: (1) Indistinguishable data objects in a program can make Soft-OLP lose optimization opportunities. As discussed above, *apsi* is such an example. (2) An object-level cache partitioning decision is a function of the program, the program input, and the cache capacity. Our approach can be effective for a program with a given input only when there is an object with reuse distances larger than the cache capacity with the conventional LRU-based cache and less than the capacity of the assigned cache portion within an object-level cache partition. In most cases where our approach is unable to improve cache performance, reuse distances are either smaller than the cache size at the whole program level (in which case standard LRU cache is very effective) or much larger than the cache size even within one object (in which case object-level partitioning still cannot help). For example, with reference input $2^{LM} = 128$, we could not improve *mgrid*'s performance but the same program's L2 miss rate was reduced in [20]. The reason is that the optimization

opportunity for this program is in the accesses with x^2 locality patterns, whose reuse distances are slightly larger than 512KB in cache lines with conventional LRU caching but less than 512KB in cache lines within individual objects. As our experimental platform has a 4MB L2 cache while the L2 cache capacity on the PowerPC 970FX processor used in [20] was only 512KB, these accesses do not incur cache misses on our experimental platform with conventional LRU caching but could be optimized for PowerPC 970FX. As another example, we also directly applied the partitioning decision-making algorithm with the accurate profiles from training runs. Because training runs often have small working sets that fit in the cache, we only achieved performance improvement in 9 out of 36 training cases.

Case Studies: We analyze the effect of object-level cache partitioning in detail using *CG* and *LU*. These parallel benchmarks are interesting as they are quite complicated and both incur high miss rates across input ranges and thread numbers. Furthermore, they represent two types of applications that can benefit from object-level cache partitioning.

Fig. 8 shows speedups and cache miss reductions with *CG* for training and actual inputs, in comparison to the shared LRU cache. For *CG*, with different thread numbers and input sizes, the partitioning decision is unchanged. It always segregates hog objects $colidx$ and a, and lets the other objects share the remaining cache capacity. Performance improvement and miss rate reduction are also across thread numbers and input sizes, as shown in Fig. 8. When the number of threads is increased from one to two, the relative benefit of cache partitioning decreases. For example, when two threads are used, *CG*'s average speedup is reduced from 1.07 to 1.04. Experimental data show that *CG*'s total L2 misses increase with the number of threads. Since the cache miss rates for a and $colidx$ do not change, this reduction in performance improvement is likely due to increasing intra-object cache contention on p between two threads.

Fig. 9 shows speedups and cache miss reductions on *LU*, a program of over 3000 lines. Results with both training and actual inputs are shown. *LU* achieves very high performance improvements with certain input ranges using Soft-OLP. For instance, when the input size is 70, it achieves a performance improvement of 31.7%, compared to uncontrolled LRU caching. However, unlike *CG*, *LU* does not exhibit this trend of performance improvement across all input sizes. When the problem size is greater than 90, cache partitioning is not beneficial. This difference between *CG* and *LU* results from the fact that the majority of *LU*'s data references are on hot objects u, rsd, a, b, c and d, not on hog objects as in *CG*. Given a cache configuration, the thrashing effect from accessing multiple hot objects is significant only over a range of input sizes. When the input goes beyond a certain value, accesses on a single hot object start to thrash in the cache. As a result, *LU*'s partitioning decisions vary with inputs and cache sizes by involving different numbers of hot objects.

Effect of Partial Object Cache Partitioning: The partitioning decision algorithm considers dividing objects with object-size reuse distances. In the eight benchmarks with performance improvement, six of them have partial object cache partitioning with at least one input. This is because most of the benchmarks used are iterative scientific programs and large objects often have reuse distances as linear functions of its size. For example, Fig. 10 shows the speedup and cache miss reductions on *art* with and without partial object cache partitioning. With *art*, the majority of its memory accesses involve three hot objects $f1\_layer$, tds and bus. When input parameter $objects$ is larger than 40, by only segregating complete objects Soft-OLP cannot improve performance. However with partial object cache partitioning, object bus is divided into two segments and the object's temporal locality is exploited for the segment staying in cache.

IX. Related Work

Many approaches have been proposed to partition the shared cache at the thread or process level. Most of them add cache partitioning support in the micro-architecture [21], [22], [23]. There have been studies on OS-based cache partitioning policies and their interaction with micro-architecture support [24], [25]. Our work differs from these studies in that we focus on object-level cache partitioning and do not require any new architectural support.

Understanding data locality is critical to performance optimization. Authors in [26] proposed a data trace representation in an object-relative form and demonstrated its application in computing memory dependence frequencies and stride patterns. Whole-program data locality pattern recognition has been studied in [10], and we have used the technique to detect locality patterns at the object level.

Several studies have sought to improve data locality through compiler/OS interaction [7], [8], [27], [28], [29]. The approaches in [7], [8], [27] focus more on avoidance of conflict misses, which is not a significant problem with modern L2/L3 caches because of their high degrees of associativity. Also using reuse distance as a tool, the authors in [29] optimized programs for embedded processors with a main cache and a mini-cache. Their framework decides which cache a virtual page is mapped to. The authors in [28] used reuse distance information to manage superpages.

A recent study called ROCS [20] also addresses the weakness of the LRU replacement policy with the last-level cache. It also uses page coloring as the basic support. Their approach are different from ours in that ROCS collects on-line data locality information at the page level using architecture-specific sampled-address data registers (SDAR).

X. Conclusion and Future Work

We have designed and implemented a framework that partitions the last-level 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 and cache partitioning decision making and enforcement. Experimental results with benchmarks from a computational kernel suite, SPEC CPU2000 benchmarks and OpenMP NAS benchmarks demonstrate the effectiveness of our framework.

We have recently looked into the cache conflicting problems for databases running on multicore processors, where OS and database systems collaboratively make efforts to exploit cache locality guided by the database domain knowledge [30]. Other ongoing and future work is planned along the following directions. First, we plan to generalize our framework to take small objects into consideration since small objects may reveal a collective locality behavior. Second, we plan to carefully study the potential for combining inter-object cache partitioning and inter-thread techniques such as job pairing. Third, some emerging architectures such as the Intel Larrabee [31] will provide a rich set of instructions for explicit cache control. When these architectures are available, we plan to study compiler optimization issues with these cache control instructions, based on the program analysis techniques proposed here.

ACKNOWLEDGMENT

This research is supported in part by the U.S. National Science Foundation through awards 0721516, 0834393, 0913050, 0811781, 0509467, 0403342, 0834476 and 0541366. We thank Chengliang Zhang for sharing his insights at the intial stage of the study. The authors would also like to thank the anonymous reviewers for their comments on the earlier version of this paper.

REFERENCES

[1] D. Burger, J. R. Goodman, and A. Kägi, "Memory bandwidth limitations of future microprocessors," in *Proc. ISCA '96*.

[2] R. Subramanian, Y. Smaragdakis, and G. H. Loh, "Adaptive caches: Effective shaping of cache behavior to workloads," in *Proc. MICRO'06*.

[3] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. ISCA '07*.

[4] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proc. ICS '95*.

[5] T. L. Johnson, D. A. Connors, M. C. Merten, and W. mei W. Hwu, "Run-time cache bypassing," *IEEE Trans. Comput.*, vol. 48, no. 12, pp. 1338–1354, 1999.

[6] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corp., http://www.intel.com/products/processor/manuals/index.htm.

[7] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, "Compiler-directed page coloring for multiprocessors," in *Proc. ASPLOS'96*.

[8] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proc. ASPLOS'98*.

[9] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *Proc. PACT '02*.

[10] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding, "Miss rate prediction across program inputs and cache configurations," *IEEE Trans. Computers*, vol. 56, no. 3, pp. 328–343, 2007.

[11] G. Taylor, P. Davies, and M. Farmwald, "The TLB slice–a low-cost high-speed address translation mechanism," in *Proc. ISCA'90*.

[12] 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 *Proc. HPCA'08*.

[13] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM System Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[14] 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 *Proc. PLDI '05*.

[15] R. Sugumar and S. Abraham, "Efficient simulation of multiple cache configurations using binomial trees," CSE Division,Universityof Michigan, Tech. Rep. CSE-TR-111-91, 1991.

[16] *Perfmon project*, HP Corp., http://www.hpl.hp.com/research/linux/perfmon.

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

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

[19] *PLUTO – An automatic parallelizer and locality optimizer for multicores*, example directory in http://www.cse.ohio-state.edu/~bondhugu/pluto/.

[20] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer," in *Proc. MICRO '08*.

[21] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO'06*.

[22] 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 *Proc. PACT'06*.

[23] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *Proc. ICS'07*.

[24] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural support for operating system-driven CMP cache management," in *Proc. PACT'06*.

[25] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *Proc. MICRO'06*.

[26] 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 *Proc. CGO '04*.

[27] R. Ravindran, M. Chu, and S. Mahlke, "Compiler-managed partitioned data caches for low power," in *Proc. LCTES '07*, 2007, pp. 237–247.

[28] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski, "Multiple page size modeling and optimization," in *Proc. PACT'05*.

[29] R. Xu and Z. Li, "A sample-based cache mapping scheme," in *Proc. LCTES '05*.

[30] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, "MCC-DB: minimizing cache conflicts in multi-core processors for databases," in *Proc. VLDB'09*.

[31] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.