Code Layout Optimization for Defensiveness and Politeness in Shared Cache

Pengcheng Li, Hao Luo, Chen Ding
Department of Computer Science, University of Rochester
Rochester, NY, US, 14627
{pli, hluo, cding}@cs.rochester.edu

Ziang Hu, Handong Ye
Futurewei Technologies Inc.
Santa Clara, CA, US
{ziang, hye}@huawei.com

***Abstract*—Code layout optimization seeks to reorganize the instructions of a program to better utilize the cache. On multicore, parallel executions improve the throughput but may significantly increase the cache contention, because the co-run programs share the cache and in the case of hyper-threading, the instruction cache.**

In this paper, we extend the reference affinity model for use in whole-program code layout optimization. We also implement the temporal relation graph (TRG) model used in prior work for comparison. For code reorganization, we have developed both function reordering and inter-procedural basic-block reordering. We implement the two models and the two transformations in the LLVM compiler. Experimental results on a set of benchmarks show frequently 20% to 50% reduction in instruction cache misses. By better utilizing the shared cache, the new techniques magnify the throughput improvement of hyper-threading by 8%.

I. Introduction

As multi-core processors become commonplace and cloud computing gains acceptance, more applications are run sharing the same cache hierarchy. Managing cache sharing is not just for achieving good performance, but also for ensuring stable performance in a dynamic environment; and not just for parallel programs but also for sequential programs co-run with each other.

Program optimization for shared cache is an important problem. In this paper, we explore the subject by examining the effect of traditional cache optimization for sequential code co-running in shared cache. There are two goals when optimizing a program for shared cache, defined below informally using the terms from existing work.

- *Defensiveness*: making a program more robust against peer interference [2].
- *Politeness*: making itself less interfering to others [10], also called *niceness* [26].

Traditional cache optimization reduces the reuse distance of data accesses. The first question is whether it also improves defensiveness and politeness (and what precisely do these terms mean). We will address this question in Section II-A.

Next is the choice of cache optimization. We use code layout optimization. Given a program with F functions, there are $F!$ possible layouts. The goal is to find the one with the best cache performance.

Programs share the instruction cache if they run together using simultaneous multi-threading (SMT). Most high-performance processors today use SMT to turn a single physical core into multiple logical cores. The first implementation in Intel Xeon showed that it adds less than 5% to the chip size and maximum power requirement and provides gains of up to 30% in performance [19]. IBM machines have 4 SMT threads on a Power 7 core and will have 8 threads on Power 8. An extensive study on sequential, parallel and managed workloads found that SMT "delivers substantial energy savings" [7].

In an experiment which we will describe in more detail later, we found that 9 out of 29 SPEC CPU 2006 programs have non-trivial miss ratios in the instruction cache. The next table shows the average miss ratio in solo execution and in hyper-threading co-run with two different peers:

	avg. miss ratio	increase over solo
solo	1.5%	—
co-run 1	2.5%	67%
co-run 2	3.8%	153%

We see that 30% of the benchmark programs will see significantly higher contention in shared instruction cache, which makes it a good target for code layout optimization.

A useful concept is reference affinity. It finds data or code that are often used together in time and places them together in memory. Past work has shown that reference affinity is effective to reorganize structure fields and basic blocks inside a procedure [34], [32], [31]. The number of items to reorder was small, compared to the number of functions or basic blocks in a large program. In this paper, we describe an extension to use reference affinity to reorganize code for the whole program.

Another useful model is called temporal-relation graph (TRG), which was designed especially to optimize code layout [8]. It was later used to optimize for SMT shared cache [13]. In this paper, we study the TRG model as an alternative solution.

We describe two transformations. One is global reordering of functions, and the second is inter-procedural reordering of basic blocks. Previous work targeted either function reordering or intra-procedural basic block reordering. With the two locality models and two transforms, we build and evaluate four code-layout optimizers, three of which have not been studied in the past.

The paper makes several novel contributions:

1) It formally defines the notions of defensiveness and politeness as the goals for program optimization for shared cache.
2) It extends the reference affinity model for global code layout optimization.
3) It describes a novel design for inter-procedural basic-block reordering.
4) These models and transformations are implemented in a real system. The paper evaluates the four optimizers and their effect in shared cache in a multi-core, multi-level memory hierarchy. Previous work used simulated (often sequential) hardware. This paper measures the parallel performance on a real processor and employs both hardware counters and simulation in evaluation.
5) It shows many cases that an optimization does not improve solo-run performance but improve co-run performance.
6) It relates the positive results to a harsh theoretical limit on data layout optimization [23].

Program code accounts for often a very small fraction of run-time memory usage. The end-to-end performance improvement we may expect is small. However, for compiler design, a few percent improvement is useful enough for a single optimization pass and significant enough if a large number of programs can benefit. We will evaluate how broadly typical programs can benefit.

II. Code Layout Optimizations for Shared Cache

This section first defines defensiveness and politeness more formally. Then it describes two adapted locality models: affinity and temporal-relation graph (TRG), followed by two program transformation techniques: function and basic-block reordering. Finally it gives more details about the implementation.

A. Optimization for Cache Defensiveness and Politeness

Shared cache co-run increases conflicts in two ways. First, each program adds to the over demand of cache. The interference in fully associative cache (i.e. capacity misses) can be quantified by two metrics: reuse distance (*RD*) and footprint (*FP*) [28], [25], [5], [29]. The relation can be expressed by the following equation, where the miss probability of each access by a program A depends on whether the sum of its reuse distance and the peer footprint exceeds the cache size C.

$$P(\text{self.miss}) = P(\text{self.RD} + \text{peer.FP} \geq C)$$

The footprint is the average amount of memory accessed during a time period. In the equation, the time period is the reuse time (the time between the last access to the same data block and the current access).

A recent, higher order theory shows that in practice, the reuse distance can be substituted by the footprint [30], where all-window statistical metrics are explored. Li et al. used all-window statistical metrics to study memory behaviors [15], [18], [6], [16]. Thus the equation is changed to:

$$P(\text{self.miss}) = P(\text{self.FP} + \text{peer.FP} \geq C)$$

We can divide the footprint into two parts, instruction and data, and call them *FP.inst* and *FP.data*. Rewriting the last equation, we have

$$P(\text{self.miss}) = P(\text{self.FP.(inst+data)} + \text{peer.FP.(inst+data)} \geq C) \quad (1)$$

An instruction cache is special in that it stores only part of the footprint. From the general Eq 1, we can easily derive the miss probability in the instruction cache of size C':

$$P(\text{self.icache.miss}) = P(\text{self.FP.inst} + \text{peer.FP.inst} \geq C') \quad (2)$$

From the last two equations, we can classify the benefits from reducing the instruction footprint of a program, self.FP, in shared cache.

1) **(Locality)** Conventional benefits in solo-run come from fewer self instruction misses in instruction cache (Eq 2) and fewer self instruction and data misses in unified cache (Eq 1).
2) **(Defensiveness)** Benefits in shared cache, in addition to (1), come from fewer self instruction misses in instruction cache (Eq 2) and fewer self instruction and data misses in unified cache (Eq 1).
3) **(Politeness)** Benefits in shared cache, in addition to (1,2), come from fewer peer instruction misses in instruction cache (Eq 2) and fewer peer instruction and data misses in unified cache (Eq 1).

These benefits are known but with the new, footprint-based formulation, we can classify them precisely, as we have just done.

B. Affinity Analysis

Zhong and her colleagues developed the model of *reference affinity*. It measures how close a group of data are accessed together in a reference trace. It proves that the model gives a hierarchical partition of program data. At the top is the set of all data with the weakest affinity. At the bottom is each data element with the strongest affinity [34].

The technique was developed to model program data and shown effective for data cache. Here we give a new algorithm to analyze reference affinity in program code. First, we give several definitions.

Definition 1: **Trimmed BB or Function Trace** A trimmed basic-block (BB) trace is a sequence of basic blocks where no two consecutive blocks are the same. A trimmed function trace is a sequence of functions where no two consecutive functions are the same.

Definition 2: **Footprint** In a trimmed trace, any two occurrences form a window. The footprint $fp{<}a, b{>}$ is the total amount of code occurred in the window, including a, b.

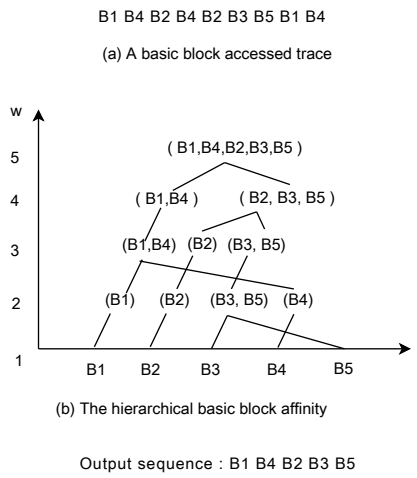


Fig. 1: Hierarchical w-window affinity

The exact size of a basic block or a function depends on compiler code generation. As an approximation, we use the number of distinct basic blocks or functions as the footprint. In the following, we discuss the analysis over a basic-block trace, but it applies to a function trace in the same way.

For example, in a trimmed basic-block trace $T = B_1B_3B_2B_3B_4$, the footprint $fp<B_1, B_2> = 3$.

Definition 3: **w-Window Affinity** For any two code blocks B_x and B_y, if every occurrence of B_x in the trace has a corresponding occurrence of B_y such that $fp<B_x, B_y> <= w$, we say B_x and B_y has w-window affinity.

Definition 4: **Affinity Group and Affinity Partition** Given w, all basic blocks can be divided into a set of groups such that in each group, every pair of blocks have w-window affinity. Each group is a w-window affinity group. The partition is the w-window affinity partition.

Definition 5: **Affinity Hierarchy** As w ranges from 1 to ∞, the affinity partitions form the affinity hierarchy. At the bottom in the 1-window partition, every block is in a separate affinity group. At the top in the $infty$-window partition, all blocks are in one group.

W-window affinity differs from the original definition, which uses the concept of a *link* [34]. In w-window affinity, a w-window marks the locale in which accesses of an affinity group must happen. In link-based affinity, the window size is proportional to the size of an affinity group and not constant. As a result, the partition is unique in link-based affinity but not in w-window affinity. However, the benefit of w-window affinity is faster analysis, a feature we will focus on after showing an example.

Figure 1(a) is an access sequence of 9 executions of 5 basic blocks. With a quick glance, we can see that B_3, B_5 have 2-window affinity. It is more complex when $w = 3$. Two affinity groups are possible, B_3, B_5 as before and B_2, B_3. In the algorithm we will present, the lower-level group takes precedence. When $w = 4$, we have B_1, B_4 and B_2, B_3, B_5 in two groups. When $w = 5$, we have all blocks in one group. The w-window hierarchy is shown in Figure 1(b).

The procedure in Algorithm 1 shows a straightforward solution to compute the w-window affinity hierarchy. For each w from small to large, it incrementally forms all affinity groups.

Algorithm 1: Hierarchical Code Block Locality Affinity

procedure ComputeHierarchicalAffinity

Input: A code block trace

Output: The w-window affinity hierarchy

```
1: for w from 1 to w_max do
2:    initialize an empty set of groups
3:    while there exist code blocks not grouped do
4:       randomly pick one code block A from the remaining
         ungrouped code blocks
5:       for each existed code block group G do
6:          add to G if every code block B in G such that
            fp<A,B> <=w
7:       end for
8:       if code block A not added to any existed groups then
9:          create a new group and add A
10:      end if
11:   end while
12: end for
```

end*ComputeHierarchicalAffinity*

We have developed an efficient solution. For a given w, we run a stack simulation of the trace [20]. At each step, we see all basic blocks that occur in a w-window with the accessed block. We record the frequency of co-occurrences. At the end of the simulation, we put basic blocks in the same group if they always occur together in a w-window. The time complexity is $O(WNB)$, where W is the number of w values, N is the length of the trace, and B is the number of basic blocks. To improve efficiency, we choose w between 2 and 20, and we use test-data-set inputs. We also use a trace pruning method, which we describe in Section II-F. After these optimizations, the increased compilation time is only a couple of times of original compilation time, which is fully acceptable.

In comparison, the original definition of reference affinity is NP-hard to analyze [32]. Zhong et al. gave a heuristic method to analyze affinity among structure fields [34]. It was used to analyze up to 14 fields. For code layout analysis, the number of functions and basic blocks is far more numerous.

Once we have the affinity hierarchy, we generate the optimized code sequence by simply a bottom-up traversal of the hierarchy. For the example in Figure 1, the reordered sequence is $B_1B_4B_2B_3B_5$.

C. TRG Analysis

Gloy and Smith defined a temporal-relation graph to model the potential cache conflicts between program functions [8]. They call their method Temporal Profile Conflict Model(TPCM). The original algorithm has mainly two parts: TRG construction first and then TRG reduction. In TRG reduction, it finds cache-relative alignments for functions greedily to minimize cache conflicts.

In this paper, we modify the process of TRG reduction. Instead of adding space between functions, we find a new order

for functions. We also apply the technique to find the order for basic blocks. We call our adaptation *TRG analysis*.

First, we define TRG as follows. Figure 2(a) shows an example.

Definition 6: **Temporal Relationship Graph** Given a code block trace, a temporal relationship graph is defined as a weighted undirected graph G<V,E>. The node V represents the code block. The edge E has a weight, which is the number of potential conflicts between the two end nodes. The number of conflicts is the times that two successive occurrences of one end node are interleaved with at least one occurrence of the other end node, and vice versa.

To construct TRG, we follow the original algorithm except that we used a hash table plus a link list to make the search part faster ($O(1)$ time).

In TRG reduction, we first assume the same size S for all code blocks. Let cache size be C, associativity be A and cache block size be B. A code block occupies $\lceil (S/(A*B)) \rceil$ cache sets. The total number of cache sets is $C/(A*B)$. Then we have $(C/(A*B))/(\lceil (S/(A*B))) \rceil$ slots to place a code block after we align all code blocks to cache block boundary.

Gloy and Smith used the actual code size, which our compiler cannot do since it has the intermediate not binary code. We assume the same size for every function and basic block. They recommended setting the cache size C to be twice the actual cache size. We abide by this advice.

The constant $2C$ specifies the footprint window which the algorithm examines for co-occurrences. In comparison, affinity analysis consider multiple sizes, not a single size. In terms of co-occurrence information, TRG is equivalent as one layer of the affinity hierarchy. In transformation, it uses the information completely differently. In practice, our affinity analysis considers the range of window sizes far smaller than $2C$.

The Algorithm 2 describes the process of TRG Reduction. In the algorithm, we use a linked list for each slot. When a code block is added to a link list L, they will be combined into one node in TRG. Each link list has one corresponding node in TRG. The algorithm uses L to denote a node, e.g. in step 11.

Algorithm 2: TRG Reduction

procedure ReduceTemprolRelationshipGraph

Input: A TRG G<V, E>, K : the number of code slots

Output: A new reordered sequence of code blocks

1: initialize K link lists for K slots
2: **for** repeatedly choose the heaviest edge <A,B> in the TRG, where A, B $\in$ V **do**
3: **if** A is not in the K link lists **then**
4: initialize a variable **slotMark** with NULL link list, that records which link list A will be added to
5: initialize a variable **conflicts** with ∞
6: **for** each link list L **do**
7: **if** L is an empty link list **then**
8: assign L to slotMark
9: break
10: **else**
11: **if** weight of E<A, L> <conflicts **then**
12: assign L to slotMark
13: assign weight of E<A, L> to conflicts
14: **end if**
15: **end if**Ł
16: **end for**
17: add A to slotMark link list
18: combine A and slotMark node into one node in G, and combine all the connected edges of the two nodes
19: **for** each other code slot L except slotMark slot **do**
20: remove E<A, L> in G
21: **end for**
22: **end if**
23: repeat steps from 4 to 22 for node B
24: **end for**
25: **for** k from 1 to K **do**
26: **if** the $k-$nd link list is not empty **then**
27: output the header node of the link list and remove it from the link list
28: **end if**
29: **end for**

end*ReduceTemprolRelationshipGraph*

In Algorithm 2, steps from 2 to 24 are working on the graph reduction process. After these steps, it forms K non-empty link lists, that represents every code block has chosen its belong-to code slot. In steps from 25 to 29, we alternately choose and output the header code block from K link lists as a sequence. After outputting one, we remove it from link lists. Notice that when two nodes choose the same slot, to show the conflict increase we combine all the edges of the two nodes, done in step 18. When two nodes choose a different slot, there is no conflict between them. We remove the edge between them, done in steps 19 to 21.

Figure 2(b) shows the TRG reduction process of Figure 2(a), where we assume there are three code slots. First, E<A, B> is reduced. The two nodes choose the first and second slots respectively. In the second step, E<E, F> is reduced. E chooses the third slot. F and A are combined into one node. Meanwhile, E<B, F> is removed. In the third step, E and C are combined into one node. Finally we generate the reordered sequence : (A B E F C).

The time complexity of TRG construction is $O(N*Q)$, where Q is the size of stack ($2C$ as recommended). The time complexity of TRG Reduction is $O(N^3)$. Because it needs to traverse all edges in TRG, i.e. $O(N^2)$ in the worst case. For each edge, it costs $O(K)$ time to reduce, where K is the number of slots.

D. Function Reordering

Function reordering is straightforward given the sequence produced by the affinity analysis or the TRG analysis described in the previous two sections. In implementation we use the LLVM compiler [14] by first compiling all program code into a single byte-code file. The reordering is a simple transformation over the file. In this transformation, we do not insert spaces between functions.

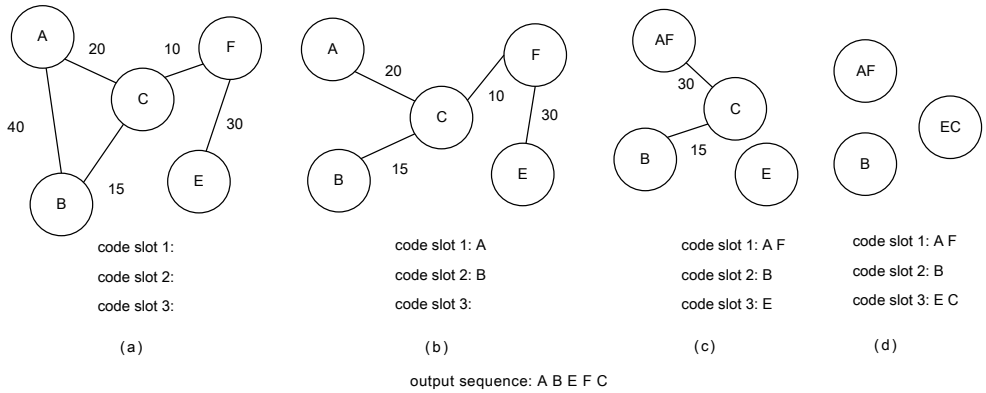


Fig. 2: An example of TRG Reduction. Assume cache is divided into 3 code slots. (a) shows the initial state of an example TRG graph. (b)(c)(d) show the TRG reduction process. The resulted sequence is (A B E F C).

E. Basic-block Reordering

Much of the literature in code layout optimization is intra-procedural. Compilers such as LLVM and GCC provide profiling-based basic block reordering, also within a procedure. In this work, we have developed a compiler solution that can reorder basic blocks across functions. When a function has branches, and each invocation executes only a small fraction of its body, inter-procedural reordering may be more beneficial than intra-procedural reordering.

Figure 3 gives an example program. It repeatedly calls two functions in a loop. In each invocation, only half of either function is executed. With inter-procedural reordering, we can extract two related basic blocks from each function and place them together, as shown by the example in Figure 3.

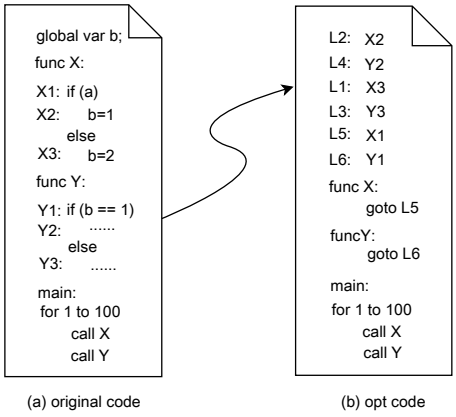


Fig. 3: An example of inter-procedural BB reordering

More specifically, the example shows two functions X, Y. Two pairs of basic blocks, $X2, Y2$ and $X3, Y3$, are always executed together. The other blocks, $X1, Y1$, execute with both pairs. The affinity-based layout places them in the order shown in the figure. The affinity layout is different from path-based layout since $X1, Y1$ are placed after the other four blocks.

Basic block reordering takes three steps: pre-processing, reordering and post-processing. In pre-processing, we add a jump instruction at the start of each function to jump to its first basic block. This is shown in the example in Figure 3.

Prog.	Instr Count		L1 Icache Miss		
	Dynamic (Billions)	Static (Bytes)	Solo	Co-run	
				Gcc	Gamess
perlbench	1937.32	788.67K	1.99%	2.39%	3.12%
gcc	149.80	1.90M	1.56%	1.99%	3.09%
mcf	832.27	86.91K	0.00%	0.05%	0.08%
gobmk	500.70	943.59K	2.73%	4.56%	6.96%
povray	1922.59	453.70K	2.10%	3.01%	4.38%
sjeng	4459.03	145.47K	0.60%	2.13%	4.68%
omnetpp	1605.31	523.97K	0.37%	1.66%	3.44%
xalancbmk	4635.75	16.28M	1.53%	2.92%	5.02%

TABLE I: The characteristics of 8 SPEC CPU2006 benchmarks

In addition, we append a jump instruction to some other basic blocks. In the normal code layout when a branch is not taken, the execution falls through to the adjacent block. To enable unrestricted code movement, we need an explicit jump to find the right fall-through block.

After preprocessing, all basic blocks are free to be reordered. We perform the reordering using a procedure similar to function reordering. The reordering step is also responsible for generating assembly code for each basic block according to the new sequence.

Finally, the post-processing step is responsible for sanity check, residual code elimination and other cleanup work.

F. System Implementation

The overall system has two main modules: locality modeling and program transformation. For a source program, the modeling step instruments the program and runs it using the test data input set. Then it gives the reordered sequence to program transformation.

As described before, the system implements two locality models: w-window affinity and the TRG model. It can instrument, analyze and transform at either function or basic-block granularity. The compiler support is implemented in the instrumentation phase, runtime phase, optimization phase and code generation phase of LLVM [14] . The output is

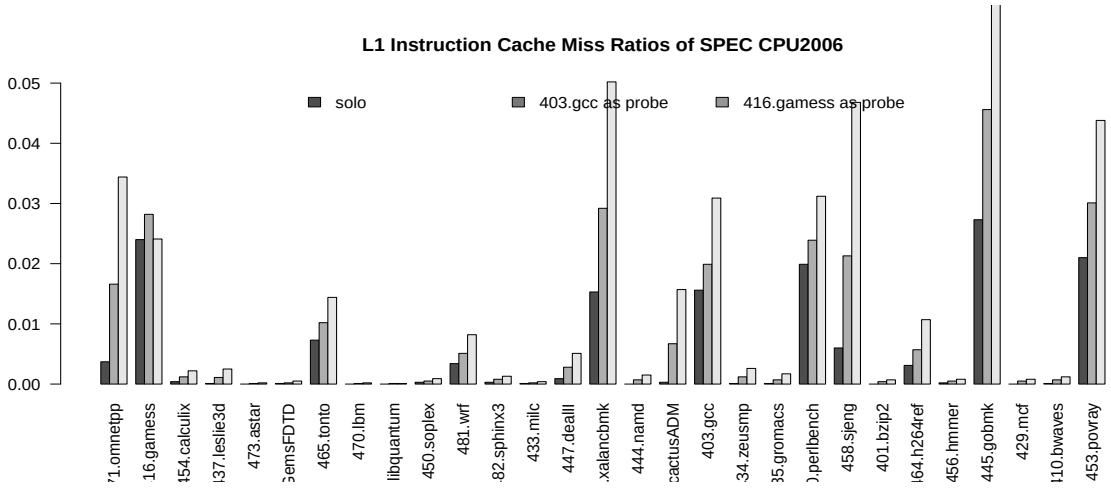


Fig. 4: L1 instruction cache miss ratios under solo- and co-run

four optimized binaries: function affinity, basic block affinity, function TRG and basic block TRG.

Instrumentation We instrumented each basic block or function in LLVM's intermediate representation. After a test-input run, we record the trace of all functions and all basic blocks in a file. Meanwhile, we record a mapping file to assign each basic block or function an index, which is used in representing the trace and in locality analysis.

Trace Pruning The size of basic-block traces may be too large for analysis. For example, 403.gcc in SPEC CPU2006 even with the test input has an 8 giga-byte trace. We prune the trace by selecting the 10,000 most frequently executed basic blocks and keeping only those occurrences in the trace. We used the technique of Hashemi et al. [9] to select the most popular functions and basic blocks during trace processing. We have also developed techniques for trace sampling to refine and extract a effective sub-trace without loosing too much information. Pruning, for example, typically keeps over 90% of the original trace.

Stack Processing In both models, we need to maintain a stack when analyzing the trace. We need to search the stack to find the currently accessed code block. For fast search, we use a similar technique in the memory management of Linux OS kernel, with a link list to manage the virtual pages, and a red-black tree for rapid insertion, search and deletion. In the kernel, the link list is used for maintain the order of allocated pages. Here we use a hash table plus a link list. The link list is to implement a stack. The hash table is for searching the stack rapidly. Through these improvements, we can analyze our test suite in time and space acceptable for a compile-time solution.

III. Evaluation

A. Experimental Setup

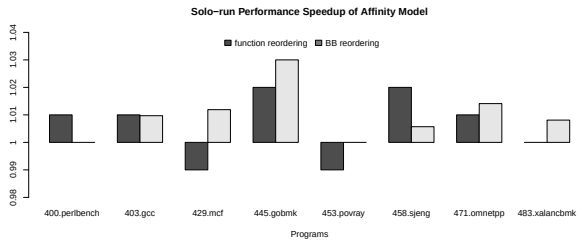
Machine Platform We performed our experiments both on a real machine and an instruction cache simulator. The real machine has two Intel Xeon E5520 3.2GHz processors. Each socket has four physical cores. Each core has two hyper-threads. We used the two hyper-threads to run our programs so they share the instruction cache. The OS kernel version is 2.6.34. The instruction cache simulator is based on Intel Pin tool [17] extended to simulate a CMP L1 instruction cache with size of 32KB and 4-way associativity. The cache configuration simulated is the same as on the real machine. We also use PAPI [3] libraries to measure the instruction cache miss ratios using hardware performance counters.

One may question whether hardware design can simply solve the problem by increasing the size of the instruction cache. The size of 32KB has not changed for successive processor generations in the past. On Intel machines, the size is chosen as a result of a clever trick which lets a processor, at a memory access, lookup the physically addressed cache while translating the virtual address to physical address. The trick depends on the difference between page size (4KB) and cache-block size (64B). Since the difference has not changed, the instruction cache size is unlikely to increase.

Test Suite SPEC CPU benchmarks have relatively small code sizes but are widely used and representative of workloads in science and engineering. We firstly run all benchmarks and measure the instruction miss ratio using hardware performance counters. We perform three rounds of experiments. In the first round, we get the L1 instruction cache miss ratios under solo-run and select those programs that have non-trivial miss ratios. In the second and the third rounds, we use *gcc* and *gamess* as peer programs to create contention in shared cache.

Based on the solo-run miss ratios shown in Figure 4, we choose *sjeng* and 7 other programs whose miss ratio is even higher. In this group, we exclude *tonto* and *gamess* since they are in Fortran, and our compiler is for C/C++. In their place, we add two programs, *omnetpp*, which shows high sensitivity to peer interference, and *mcf*, which we have found to be sensitive in spite of of the near zero miss ratio in a solo execution. Table I shows numerical measures for the 8 programs, including the length of the trace and the total instruction size (for the reference input).

The optimized code is compiled in both default and O2 to test the interaction with other compiler optimizations. For cross comparison, all function reordering is compiled by default optimization, and basic block reordering (best performing as



(a) Performance speedup

(b) Instruction cache miss ratio reduction, measured by hardware counters

Fig. 5: The solo-run effect of the two affinity optimizers.

Benchmarks	Function Affinity			BB Affinity			Function TRG		
	speedup	miss ratio reduction		speedup	miss ratio reduction		speedup	miss ratio reduction	
		hw cnt	simulated		hw cnt	simulated		hw cnt	simulated
400.perlbench	+0.43%	3.79%	5.82%	N/A	N/A	N/A	**+7.28%**	-6.30%	12.56%
403.gcc	+0.57%	9.11%	8.03%	+1.00%	16.86%	21.17%	**+1.45%**	-0.58%	13.89%
429.mcf	+0.57%	3.81%	33.72%	**+4.17%**	28.04%	4.17%	+0.00%	0.00%	0.00%
445.gobmk	**+7.22%**	38.70%	13.00%	+4.33%	9.75%	46.67%	+5.87%	28.33%	38.10%
453.povray	-0.57%	-11.43%	18.95%	N/A	N/A	N/A	**+2.26%**	-4.41%	11.48%
458.sjeng	+1.86%	7.62%	40.61%	+1.67%	4.99%	5.00%	**+10.23%**	6.94%	12.75%
471.omnetpp	+4.00%	35.79%	60.69%	**+4.83%**	43.96%	55.33%	+2.06%	32.25%	27.94%
483.xalancbmk	**+5.86%**	24.78%	31.33%	+1.67%	28.60%	34.17%	-1.95%	-6.19%	-1.76%

TABLE II: Average co-run speedup and miss ratio reduction by the three optimizers. The best speedup for each program is marked in bold.

we will show) is compiled by O2 option. When comparing the speedups of function reordering and those of BB reordering, we actually use a faster baseline for BB reordering to improve on. With sufficient space, we will report both types of reordering on both default and O2. Our compiler could correctly reorder functions for all test programs. For BB reordering, it had errors on two programs, *perlbench* and *povray*. We show these as "N/A" in data tables and zero in bar graphs.

B. The Solo-run Effect

We show only the effect from two affinity optimizers. The TRG optimizers are not better. Across the 8 benchmarks, the performance is changed between -1% and 2% by function reordering and 0% and 3% in BB reordering. While the improvement is modest at best, the reduction in the miss ratio of the instruction cache is dramatic, up to 34% by function reordering and 37% by BB reordering. These results confirm the conventional view that SPEC CPU benchmarks are data intensive. The instruction cache performance is good enough.

It should be noted that in compiler design, an improvement over 2% (3% speedup in *gobmk* and 2% in *sjeng*) by a single optimization pass is significant since a compiler typically has tens of passes working together.

BB affinity is showing consistency and superiority. When applied to 6 programs, it reduces the miss ratio in 5 of them by 21% to 37%. The exception is -2% for *sjeng* from 2.88% to 2.94%.

C. The Co-run Effect

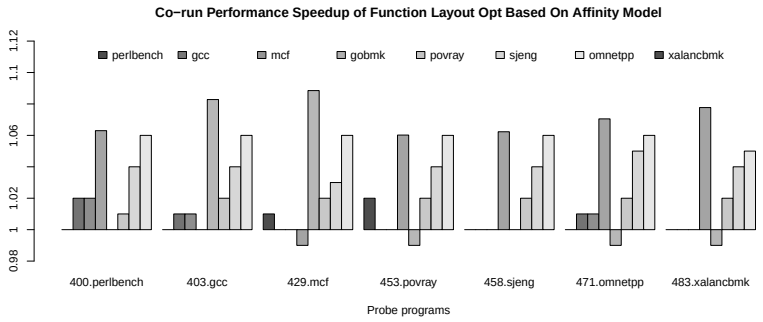
We have tested co-runs among the 8 programs, which means up to 8 co-runs per program. Each co-run is a pairing between original and optimized. We time the optimized and report the relative improvement, normalized to original-original pairing. The co-run speedups are plotted in Figure 6 for three optimizers. The BB TRG optimizer does not show improvement, so we omit it from now on. We have also plotted the miss ratio changes as measured by the hardware counters and a simulator but omit the figures to save space. For each program, we compute the average speedup across all co-run peers, and the average miss ratio reductions (hardware counted and simulated) and show them in Table II.

By looking at results in both performance and miss ratio and both individually and averaged, we see consistent trends emerge from the data to characterize the three optimizers.

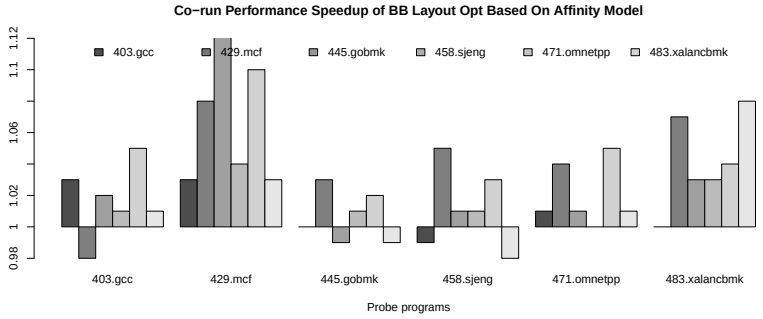
BB affinity is the most robust and best performing of the three. Of its 6 tests, it improves the average co-run performance by 4% to 5% for 3 (*mcf,gobmk,omnetpp*), and 1% to 2% for the remaining 3. The improvement coincides with significant miss ratio reduction, which ranges between 10% and 44% for the first 3 and 5% to 29% for the other 3, measured by hardware counters. Similar reduction, 4% to 55%, is observed by the simulator.

Function affinity shows robust but modest improvements. Of 8 programs, it improves 3 programs by 4% to 7.2%, 1 by 1.9%, and the remaining 4 by about 0.5%. The miss ratio reduction is as dramatic as by BB affinity except for one 11.4% miss increase for *povray* as measured by the hardware counters.

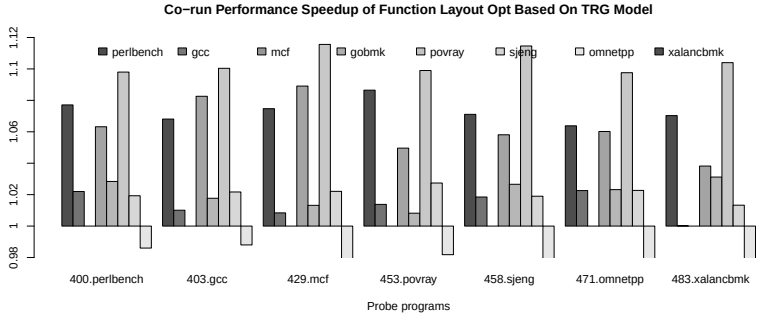
Function TRG shows two spectacular but mysterious improvements. It improves *perlbench* in 6 co-runs by 7.3% on average, but the miss ratio is 6.3% higher on average. It improves *sjeng* by over 10% (rare for code layout optimization) and reduces the miss ratio by 7% and 13% (measured and simulated). However, for the same program, function affinity



(a) Improvements of function affinity under co-run



(b) Improvements of BB affinity under co-run



(c) Improvements of function TRG under co-run

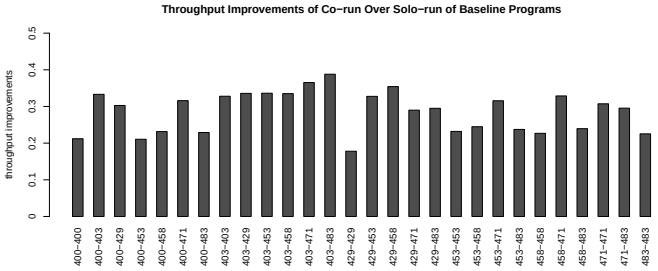
Fig. 6: Co-run speedup of three optimizers. The comparison is between co-runs of original+optimized and original+original. In each figure, the x-axis shows the original as probe programs. A bar show the speedup of an optimized program when co-running with the probe.

removes more misses (7.6% and 41%) but improves the performance for just 1.9%.

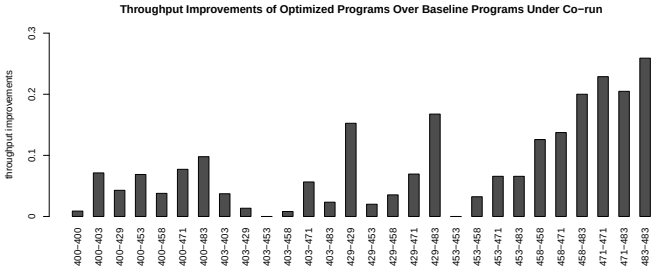
In individual results shown in Figure 6, affinity optimizers occasionally slow down a program in one co-run. On average, they always improve the co-run performance, about 0.5% at least. Function TRG is consistently beneficial except in one program for which it is consistently harmful (2% on average).

In miss ratio results, however, function TRG is counter productive in the majority of programs, 4 of the 7. In practice, TRG is sensitive to the window size $2C$. Its improvement is fragile as we try to pick the value that gives the best performance. The reason for improvement may come from factors other than the instruction cache locality. For BB reordering, TRG shows almost no improvement and many slowdowns. Still, when it is effective, it can produce the highest improvement and maintain it across all co-runs.

Overall, hardware counted cache miss ratio reductions are less than simulated ones. Simulated cache miss ratio reductions are accurate simulation results of co-run cache sharing due to merely simulating cache behavior, while in hardware, many other factors may reduce cache miss ratio reductions because of its complexity, for example prefetching. Our results in Table II confirms this point. However, the two trends of hardware counted and simulated cache miss ratio reductions coincide.



(a) Throughput improvement of co-run over solo-run. No optimization.



(b) Additional throughput improvement due to function affinity optimization.

Fig. 7: Throughput improvement showing first the benefit of sharing the cache and then the more effective sharing due to function affinity optimization.

D. Petrank-Rawitz Wall of Locality Optimization

In POPL 2002, Petrank and Rawitz showed a harsh limit — finding optimal data placement is not only NP-hard but also impossible to approximate within a constant factor if $P \neq NP$ [23]. Their result is universal, and the same limit applies to code as well as data. Observing the terms "memory wall" and "power wall", which architects use to draw attention to their problems, we may call this limit the *Petrank-Rawitz wall* to draw attention to program optimization.

Because of the theoretical limit, no practical solution can guarantee optimality or closeness to it. Still, effective solutions can be developed to capture specific patterns and optimize for these. Affinity and TRG are two such example patterns. Furthermore, having more varieties helps because an application may have more of one pattern than another. Different granularity, e.g. function or BB, effectively creates different applications for optimization since the pattern in function usage may differ from the pattern in BB usage.

The way to approach the Petrank-Rawitz Wall is through specificity and variety. We have developed four such solutions. The combined strength is greater. In solo-run, 7 out of 8 are improved by 1% to 3%. In co-run (highlighted in Table 6), 5 out of 8 gain 5% to 10.3%, and the rest at least 1.5%. These improvements are for major applications in compilation, combinatorial optimization, computer game, network simulation, document processing, and computer graphics.

Although the individual improvement is small, the technique is general and automated. We expect that SPEC results are representative, and a significant portion of real-world applications will gain a few percent performance from our techniques.

E. Boosting Hyper-threading

Figure 7(a) shows the benefit of hyper-threading by the throughput improvement of baseline co-run over baseline solo run. The downside is that programs have to share the cache and physical core and they may run slower. However, the benefit is that the time to finish both programs are 15% to over 30% faster.

Code layout optimization further improves the throughput. In Figure 7(b), we show the magnifying effect of function affinity optimization, which is the improvement from optimized-baseline co-run divided by the improvement from baseline-baseline co-run, i.e. Figure 7(a). For the 28 co-run pairs, the magnifying effect is over 5.6% for 16 pairs and 10% or more for 9 pairs (57% and 32% of all pairs). The largest is 26%. The arithmetic average is 7.9%. There is only one degradation, -8% for 453-453 co-run.

F. Combining Defensiveness and Politeness

Here we come to a negative finding. We have selected the three most improving programs from function affinity optimization and tested them in optimized-optimized co-run. Compared to optimized-baseline co-run, we see only negligible improvements (but no slowdown). This has several implications. First, the optimization is so effective that after it there is no contention in the instruction cache in optimized-baseline co-run. There is no room there to improve. Second, without benefits in L1, there is no further improvement in the unified cache in the lower levels. Because of the relative small size of instruction and the strength of affinity-based optimization, just optimizing one of the two co-run programs is enough to obtain the full benefit in shared cache. We conjecture that in cases where the active code size is large, e.g. database, and the number of co-run programs is high, combining defensiveness and politeness should see a synergistic improvement.

IV. Related Work

Peer Aware Optimization Zhang et al. shows significant benefits of locality optimization in multi-threaded code running in shared cache [33]. QoS-Compile from Tang et al. [26], [27] reduces the peer contention in program co-run. The compiler inserts the QoS markers which at run-time is adjusted to throttle back a program's execution to protect the primary program from having an excessive slowdown. Defensive tiling reduces the tile size to shorten the residence time of the tiled data in shared cache [2]. QoS makes a peer program nicer (more polite) by slowing down its speed. Defensive tiling makes a program less sensitive by reducing its activity time in shared cache. QoS is peer dependent, and defensive tiling is peer independent. The two transform program code but do not optimize the code layout, which is the goal of this work. Code layout optimization is peer independent, and the optimization improves both defensiveness and politeness.

Code Layout Optimization Traditionally a compiler compiles one function at a time. Inside a function, a major problem is to reduce the cost of branches. Code layout techniques include replication to remove conditional branches [22] and branch reordering to facilitate branch prediction [11]. A common tool is hot path profiling [1]. These techniques aim to improve the speed at which a processor core fetches the next instruction. They may improve both code layout and code optimization. However, they do not maximize the utilization of instruction and data cache. In addition, instruction fetch logic is not shared and hence not affected by program co-run.

For instruction footprint, cold (infrequent) paths are also important. Reference affinity finds the pattern of co-occurrence, both hot and cold, and places related code together. Zhang et al. reorganized the layout of basic blocks based on reference affinity and showed improvements over superblock-based code layout [32]. The transformation was intra-procedural.

TRG was invented to improve the function layout across the entire program [8]. A similar model is the Conflict Miss Graph (CMG), used for function reordering [12]. Other pairwise models have been used in cache-conscious data layout [4], [21], [24]. TRG and CMG have a different purpose than reference affinity. TRG and CMG are to reduce cache conflicts, and reference affinity is to improve spatial locality. They are also similar in that all three find co-occurred functions for special placement. A key technical difference is that TRG and CMG use a fixed size window while reference affinity uses windows of increasing sizes, as discussed in Section II-C.

TRG function layout was used to optimize the shared cache on SMT [13]. The study was simulation based and necessarily so because the Intel hyper-threading hardware was not yet released. By building a real system and using basic-block reordering, we have validated and extended their pioneering idea and quantified the improvement on real hardware.

V. Summary

This paper formalized the goals of shared cache optimization in terms of defensiveness and politeness and extended the reference affinity model so it can be used to reorganize tens of thousands of functions and basic blocks in a large program. Our work is the first experimental comparison between reference affinity and TRG. We found that affinity-based optimizations are robust, and all but basic-block TRG are effective for shared cache. Of all SPEC CPU benchmarks, 30% have non-trivial miss ratio in the instruction cache. For majority of them, our optimizations improve the whole-program performance by up to 3% in solo run and up to 10.3% in co-run. By better utilizing the shared cache, they magnify the throughput improvement of hyper-threading by 8%.

Acknowledgment

We thank Tongping Liu, Chen Tian, Quanzeng You and Amal Fahad for their help with types of discussions. We also thank the reviewers of ICPP 2014 for the insightful review comments and feedback.

References

[1] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM TOPLAS*, 16(4), 1994.

[2] B. Bao and C. Ding. Defensive loop tiling for shared cache. In *Proceedings of CGO*, pages 1–11, 2013.

[3] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.

[4] B. Calder, C. Krintz, S. John, and T. M. Austin. Cache-conscious data placement. In *Proceedings of ASPLOS*, pages 139–149, 1998.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of HPCA*, pages 340–351, 2005.

[6] C. Ding and P. Li. Cache-conscious memory management. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, 2014.

[7] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley. Looking back and looking forward: power, performance, and upheaval. *Communications of the ACM*, 55(7):105–114, 2012.

[8] N. C. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM TOPLAS*, 21(5):977–1027, 1999.

[9] A. Hashemi, D. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of PLDI*, pages 171–182, New York, NY, USA, 1997.

[10] Y. Jiang, K. Tian, X. Shen, J. Zhang, J. Chen, and R. Tripathi. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *IEEE TPDS*, 22(7):1192–1205, 2011.

[11] D. A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of PLDI*, pages 107–116, 2005.

[12] J. Kalamatianos and D. R. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of HPCA*, pages 244–253, 1998.

[13] R. Kumar and D. M. Tullsen. Compiling for instruction cache performance on a multithreaded architecture. In *Proceedings of MICRO*, pages 419–429, Los Alamitos, CA, USA, 2002.

[14] C. Lattner and V. S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of PLDI*, pages 129–142, 2005.

[15] P. Li and C. Ding. All-window data liveness. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2013.

[16] P. Li, C. Ding, and H. Luo. Modeling heap data growth using average liveness. In *Proceedings of ISMM*, 2014.

[17] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI*, pages 190–200, 2005.

[18] H. Luo, C. Ding, and P. Li. Optimal thread-to-core mapping for pipeline programs. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, 2014.

[19] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[20] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[21] N. McIntosh, S. Mannarswamy, and R. Hundt. Whole-program optimization of global variable layout. In *Proceedings of PACT*, pages 164–172, New York, NY, USA, 2006. ACM.

[22] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of PLDI*, pages 56–66, 1995.

[23] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of POPL*, 2002.

[24] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions in Embedded Computing Systems*, 2(2), 2003.

[25] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of ICS*, pages 1–12, 2001.

[26] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: mitigating contention for QoS in warehouse scale computers. In *Proceedings of CGO*, pages 1–12, 2012.

[27] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. ReQoS: Reactive static/dynamic compilation for qos in warehouse scale computers. In

Proceedings of ASPLOS, 2013.

[28] D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, 1987.

[29] X. Xiang, B. Bao, T. Bai, C. Ding, and T. M. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of PPoPP*, pages 91–102, 2011.

[30] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *Proceedings of ASPLOS*, pages 343–356, 2013.

[31] J. Yan, J. He, W. Chen, P.-C. Yew, and W. Zheng. ASLOP: A field-access affinity-based structure data layout optimizer. *SCIENCE CHINA Info. Sci.*, 54(9):1769–1783, 2011.

[32] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu. A hierarchical model of data locality. In *Proceedings of POPL*, pages 16–29, 2006.

[33] E. Z. Zhang, Y. Jiang, and X. Shen. The significance of CMP cache sharing on contemporary multithreaded applications. *IEEE TPDS*, 23(2):367–374, 2012.

[34] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of PLDI*, pages 255–266, 2004.