Appears in IEEE Computer Architecture Letters*

Code Layout Optimization for Near-Ideal Instruction Cache

Ali Ansari, Pejman Lotfi-Kamran, *Member, IEEE,* and Hamid Sarbazi-Azad

Abstract—Instruction cache misses are a significant source of performance degradation in server workloads because of their large instruction footprints and complex control flow. Due to the importance of reducing the number of instruction cache misses, there has been a myriad of proposals for hardware instruction prefetchers in the past two decades. While effectual, state-of-the-art hardware instruction prefetchers either impose considerable storage overhead or require significant changes in the frontend of a processor. Unlike hardware instruction prefetchers, code-layout optimization techniques profile a program and then reorder the code layout of the program to increase spatial locality, and hence, reduce the number of instruction cache misses. While an active area of research in the 1990s, code-layout optimization techniques have largely been neglected in the past decade. We evaluate the suitability of code-layout optimization techniques for modern server workloads and show that if we combine these techniques with a simple next-line prefetcher, they can significantly reduce the number of instruction cache misses. Moreover, we propose a new code-layout optimization algorithm and show that along with a next-line prefetcher, it offers the same performance improvement as the state-of-the-art hardware instruction prefetcher, but with almost no hardware overhead.

Index Terms—Instruction cache miss, instruction prefetcher, code-layout optimization, basic-block reordering

1 Introduction

Instruction cache misses are a major source of performance degradation in processors. Upon an L1 cache miss, even OoO processors cannot make forward progress because there is no instruction to execute. Instruction cache misses have been a major performance bottleneck since the 1980s [6], [7], [12]. Ever since the size and complexity of programs are exponentially increasing, leading to 100s of MB of complex code footprint in existing server workloads [9]. Consequently, the instruction-supply bottleneck is exacerbated with existing massive and complex code base [1], [9].

One way to reduce the number of instruction cache misses is through instruction prefetching. The simplest instruction prefetcher is a next-line prefetcher (NLP) that upon access to a cache block, sends a prefetch request for the subsequent cache block. Due to its simplicity, almost all commercial processors use a next-line prefetcher. Figure 1 shows the performance of an NLP as compared to an ideal cache in which every cache access is a hit. The figure clearly shows that there is more than 10% gap between the performance of an NLP and an ideal cache. To shrink this gap, the research community proposed many prefetchers for server workloads.

Temporal prefetchers are an important class of instruction prefetchers. Temporal prefetching record and replay the sequence of past accesses/misses [3], [4]. The state-of-the-art temporal instruction prefetcher is *Proactive Instruction Fetch (PIF)* [5] that records and replays the sequence of L1i instruction cache accesses to reduce the number of L1i misses significantly. While effective, PIF needs to store past L1i accesses, which imposes significant storage overhead (over 200 KB per core).

To reduce the storage overhead, the most recent proposal (i.e., Shotgun [10]) is not a temporal prefetcher and uses the branch target buffer (BTB) for prefetching. Shotgun uses the BTB content to go ahead in the control flow and prefetches cache blocks that are not in the cache. While Shotgun requires considerably smaller storage as compared to PIF, the area reduction comes at the cost of significant changes in the frontend of a processor.

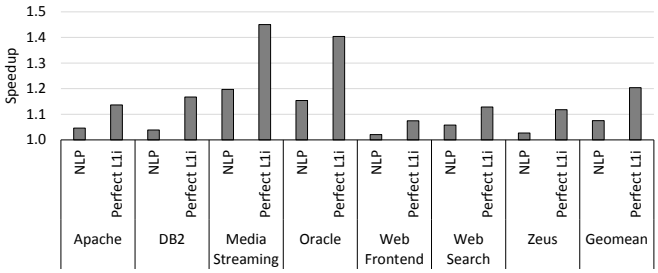

Fig. 1. Performance of a next-line prefetcher versus an ideal cache.

Code-layout optimization (CLO) is another way to mitigate frontend bottleneck. For this purpose, CLO techniques place basic blocks that tend to be executed one after another spatially next to each other in the optimized layout. To do this, they may inspect the code itself to reorder basic blocks [6] or may run the code with sample inputs, profile the execution, and use the profiling information for reordering the basic blocks [13], [14], [15], [16], [19], [22]. Not surprisingly, profile-based basic-block reordering offers better results [13].

Profile-based basic-block reordering techniques construct the control flow graph (CFG) of basic blocks where nodes correspond to basic blocks and edges show how basic blocks follow one another. By profiling the execution on a sample input, these techniques associate a frequency to each edge in the CFG. As basic-block reordering techniques are applied to basic blocks within a procedure, they start with the basic-block entry of a procedure. Using the CFG, they find a basic block that most frequently follows the entry basic-block and place it right after the entry basic-block. Then they follow the same algorithm for the most recently placed basic block (e.g., the basic block right after the entry basic-block). It has been shown that this placement technique improves the spatial locality of instruction cache accesses, resulting in fewer cache misses.

While performance degradation due to instruction cache misses has motivated many researchers to tackle this problem and offer proposals, very few pieces of work in the past 18 years studied/used code-layout optimization techniques. Partially, this lack of interest in code layout optimization is because many researchers (e.g., [11]) believed that it provided only partial improvements due to complex control flow and massive code footprints in server workloads. In this work, we assess classical code-layout optimization techniques in the context of modern server workloads and show that with small

- *A. Ansari is with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. E-mail: aansari@ce.sharif.edu*
- *P. Lotfi-Kamran is with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM).*
 He was supported in part by a grant from INSF.

- *H. Sarbazi-Azad is with the Department of Computer Engineering, Sharif University of Technology and the School of Computer Science, Institute for Research in Fundamental Sciences (IPM).*

modifications, they can increase the spatial locality of instruction accesses in today's highly sophisticated server software to the point that a simple next-line prefetcher offers a level performance that matches that of the state-of-the-art prefetchers.

Our experiments show that following the most frequent successor of each basic block which is observed in previous accesses, correctly predicts the future accesses with 90% probability, on average. As a result, code-layout optimization can be a reliable solution to address the front-end bottleneck. But the scope of basic-block reordering should be larger than a procedure. With modern software engineering principle, code developers attempt to make procedures as small as possible. More explicitly, the most frequent successor of 22 to 45% of basic blocks are out of their procedure. Consequently, techniques that limit the scope of basic-block reordering to a single procedure lose an opportunity. Moreover, we find that placing basic blocks one after another just based on the forward frequency (i.e., how many times basic-block B followed basic-block A) leads to suboptimal results. This way, the basic block placement algorithm only considers what is best for the current basic block and not for the subsequent basic block. Instead, we suggest a new basic-block placement algorithm that takes into account the best placement for both basic blocks. Finally, we show that the combination of the optimized layout and a simple next-line prefetcher offers a performance improvement that matches that of state-of-the-art instruction prefetchers like PIF.

Algorithm 1 Pick the Best Successor Algorithm

```
1:  procedure PBS( Basic Block A)
2:                                  ▷ after blocks and before blocks are sorted
3:      if the best successor is already calculated then:
4:          Return the_Best_successor_of[A]
5:      end if
6:      for AB in after blocks of A do:                ▷ AB: an After Block
7:          if AB is already chosen as a best successor then:
8:              continue
9:          end if
10:         for BB in before blocks of AB do:          ▷ BB: a Before Block
11:             if BB is A then:
12:                 Return AB
13:             else:
14:                 best_of_BB = PBS(BB)
15:                 if best_of_BB is not AB then:
16:                     continue
17:                 else:
18:                     remove the edge between A and AB
19:                     B1 = PBS(A)
20:                     W1 = Freq(A to B1) + Freq(BB to AB)
21:                     bring back the edge between A and AB
22:                     remove the edge between AB and BB
23:                     B2 = PBS(BB)
24:                     W2 = Freq(A to AB) + Freq(BB to B2)
25:                     bring back the edge between AB and BB
26:                     if W1 > W2 then:
27:                         Return B1
28:                     else:
29:                         Return AB
30:                     end if
31:                 end if
32:             end if
33:         end for
34:     end for
35:     Return nothing!
36: end procedure
```

2 Pick the Best Successor (PBS)

Pick the Best Successor (PBS) is a heuristic algorithm based on the profiling information. First, it constructs a directed control flow graph (CFG) of basic blocks. Unlike prior work, this CFG includes the whole control flow of the program rather than a specific procedure. A node in the graph is a basic block, and an edge shows the frequency of a basic block followed by another. If node A has an edge to node B, A is called a *before block* for B and similarly, B is an *after block* for A. When the *after* and *before blocks* of a basic block are determined, PBS sorts them in descending order of frequency. Then, Algorithm 1 is called separately for each basic block to find its best successor. For each basic block A, Algorithm 1 finds the most frequent *after block*, named AB which is not already determined as the best successor for another basic block. Then, it examines the *before blocks* of AB in descending order of their frequency.

The best successor of A is AB if A is the most frequent *before block* of AB or AB is not the best successor for its *before blocks* with higher frequency than A. Otherwise, the algorithm must choose between two cases:

- AB is the best successor for one of its *before block* BB
- AB is the best successor for A

The algorithm considers these two cases. For each case, it attempts to find another best successor for the other one. Then it compares the *sequential weight* of each case (i.e., sum of the frequency of edges). Based on the heavier *sequential weight*, the best successor of A is determined. Note that this procedure only finds the best successor for A while it may be called recursively for other basic blocks.

When PBS finds the best successors of all basic blocks, it starts to chain them. Starting from the first basic block of the main procedure, PBS chains the best successor of the last inserted basic block to it. When a basic block does not have the best successor, we backtrack to the root to find a node that has a successor that meets these conditions: It is not already placed in the layout and is not the best successor of other basic blocks. If no node has the requirements, a breadth-first search (BFS) is executed starting from the stalled node to find a node that meets the conditions. If PBS finds no node, it performs BFS starting from the root to find a node that meets the requirements. If no node is found, the BFS is executed starting from the root to find a node that is not already in the layout. If PBS finds no node, the final layout is ready.

With the new layout, if the current successor of a basic block is different from the original successor, the control-flow instruction at the end of the current basic block needs to be modified for the correct execution of the program.

- *Conditional Branch*: The branch condition is negated, and its target points to the location of the original subsequent basic block.
- *Unconditional Jump or Return*: These instructions never let the execution to enter the subsequent basic block. In consequence, they need no modification.
- *Call*: The *return* instruction of this *call* should change the control flow to the basic block that originally was right after the call. So we add an unconditional branch instruction after the *call* to point to the location of the original subsequent basic block.

3 Methodology

We show the key elements of the evaluation methodology in Table 1. In this section, we include the details of the evaluated processor, workloads, competing L1i miss reduction techniques, and the simulation infrastructure.

We model a server processor that resembles Intel Xeon Processor E7-4850 v4 [8]. The server processor has 16 cores and 32 MB

TABLE 1
Evaluation parameters.

Parameter	Value
Processing Nodes	14 nm, UltraSPARC III ISA, Sixteen 2 GHz OoO cores 3-wide dispatch/retirement, 128-entry ROB
Instruction Fetch Unit	32 KB, 8-way, 64B block, 4-cycle load-to-use 32-entry pre-dispatch queue TAGE [17] branch predictor, 2K-entry Branch Target Buffer
L1-D Cache	32 KB, 8-way, 64B block, 4-cycle load-to-use, 32 MSHRs
Shared LLC	32 MB, 16-way, 16 banks, 18 cycles access latency
Network-on-Chip	4×4 2D Mesh, Router: 5 ports, 3 VCs/port, 5 flits/VC 2-stage speculative pipeline, 1-cycle link traversal [2]
Main Memory	60 ns access latency, 85 GB/s peak bandwidth

TABLE 2
Server applications.

OLTP - Online Transaction Processing (TPC-C)	
DB2	IBM DB2 v8 ESE Database Server 100 Warehouses (10 GB), 2 GB Buffer Pool
Oracle	Oracle 10g Enterprise Database Server 100 Warehouses (10 GB), 1.4 GB SGA
Web Server (SPECweb99)	
Apache	Apache HTTP Server v2.0 16 K Connections, fastCGI, Worker Threading
Zeus	Zeus Web Server v4.3 16 K Connections, fastCGI
CloudSuite	
Media Streaming	Darwin Streaming Server 6.0.3 7500 Clients, 60 GB Dataset, High Bitrates
Web Frontend	Nginx 1.0.10 Built-in PHP 5.3.5 & APC 3.1.8
Web Search	Nutch 1.2/Lucene 3.0.1, 230 Clients 1.4 GB Index, 15 GB Data Segment

of last-level cache (LLC). This processor includes four DDR4 memory-channels that provide up to 85 GB/s bandwidth. Each core is a three-way out-of-order design with 128-entry re-order buffer (ROB), 64-entry load-store queue (LSQ), and 32 KB 8-way set-associative L1i and L1d caches.

We simulate the workloads listed in Table 2. We include a variety of server workloads from competing vendors, including online transaction processing, cloud web serving system, streaming server, and web server benchmarks.

For estimating system performance of server workloads, we use Flexus full-system simulator. Flexus extends the Virtutech Simics functional simulator with timing models of cores, caches, on-chip protocol controllers, and interconnect. The simulator models the SPARC v9 ISA and runs unmodified operating systems and applications.

We use the SimFlex multiprocessor sampling methodology [20] that extends the SMARTS sampling framework [21] for multiprocessors. For each workload, we have hundreds of measurements. For each measurement, we launch simulations from checkpoints with warmed caches, branch predictors, and prefetchers, and run 200 K cycles to achieve a steady state of detailed cycle-accurate simulation before collecting statistics for the subsequent 200 K cycles. Performance measurements have confidence level (interval) of 95% (less than 4%).

3.1 Competing Techniques

We evaluate the following techniques:

Next-Line Prefetcher (NLP): NLP [18] is the most-widely used instruction prefetcher. On every access to a cache block, a prefetch request will be sent for the following block if the block is not in the cache. The hardware overhead of NLP is negligible.

Classical Basic-Block Reordering (CBBR): CBBR [14] chains the basic blocks of a procedure. We profile a program based on a sample input and build a flow-edge graph where the nodes correspond to basic blocks and edges show how many times the destination basic block followed the source basic-block. CBBR sorts the edges and picks the one with the largest value. If the source basic-block of the edge does not already have a successor and the destination basic block does not already have a predecessor, the two blocks are chained together. CBBR continues until it examines all the edges. At this point, the procedure has one or more chains. CBBR sorts the chains based on the execution count of their first basic block. It first places the chain containing the procedure entry point, followed by the rest of the chains in decreasing order.

PBS: PBS is similar to CBBR with two changes: (1) instead of per-procedure basic-block reordering, PBS reorders the basic blocks of the whole program, and (2) PBS attempts to choose a basic block *B* to follow basic block *A* so that not only *B* is the best successor for *A* but also *A* is the best predecessor for *B*.

PBS-Local: PBS-Local uses the PBS algorithm to optimize the layout of procedures individually.

CBBR-Global: CBBR-Global is the same as CBBR but optimizes the CFG of the whole program instead of optimizing the individual procedures.

CBBR+NLP, *CBBR-Global+NLP* and *PBS+NLP:* These approaches use a next-line prefetcher on the optimized layout.

Proactive Instruction Fetch (PIF): PIF [5] records cache accesses in the retirement order and replays them to eliminate potential future misses. We evaluate PIF with 8 K-entry Index Table and 32 K-entry History Buffer as in the original proposal. This prefetcher requires 240 KB of per core storage.

3.2 Optimization Infrastructure

For code-layout optimization techniques, we profile the program to determine the frequency of edges by executing 2 billion instructions. In all cases, the actual measurements start after the profiling period, and the profiling and measurement periods do not overlap. Profiling is done using 10% of the simulation samples, and evaluation metrics are collected based on the simulations of the rest of the samples. We construct a CFG based on the control flow changes in the profiling period. Then, we reorder basic blocks based on the various code-layout optimization algorithms (some instructions may be changed to maintain the execution control flow, as discussed in Section 2). The operating system, shared-libraries, and application binaries are optimized in all code-layout optimization techniques. However, we maintain the code for each part in their own address space in the memory.

4 Evaluation Results

Figure 2 shows the coverage, overprediction, and performance of all designs, normalized to a system without an instruction prefetcher. On average, PBS+NLP offers the best and the second best overprediction and coverage, respectively, as compared to evaluated prefetchers, and as such, delivers the same performance as PIF, which is the state-of-the-art instruction prefetcher and requires over 200 KB of storage per core. At the same time, PBS+NLP offers 10% higher performance than CBBR, which is the classical code-layout optimization technique [14].

Comparing PBS-Local against CBBR reveals that taking into consideration the choices of the subsequent basic block is of little use when we limit the scope of code-layout optimization techniques to just the body of a procedure. When the scope of code layout optimization becomes larger than a procedure, there is more opportunity for optimization and PBS by considering not only what

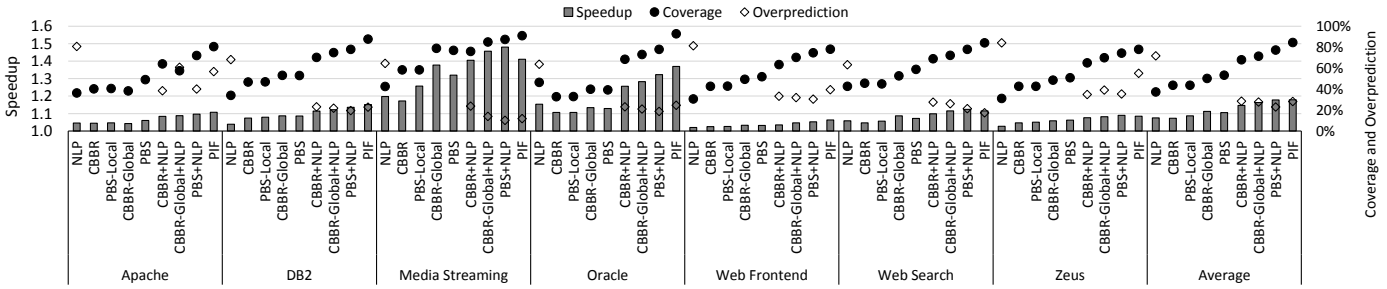

Fig. 2. Performance, coverage, and overprediction of evaluated methods, normalized to a baseline system with no instruction prefetcher.

is best for the current basic block but also for the subsequent basic block offers more performance improvement.

Despite all the benefits, when a next-line prefetcher does not accompany code-layout optimization techniques, their effectiveness is limited, and they cannot reach the peak potential. When we apply a next-line prefetcher to the optimized layout, the actual benefits of code-layout optimization techniques emerge. PBS+NLP due to rearranging basic blocks in a way that optimizes for both current and subsequent basic blocks achieves the highest performance improvement.

We also compare CBBR-Global+NLP with PBS+NLP. As both approaches optimize the whole CFG, the difference between these two proposals is due to PBS's policy to consider the backward edges during the optimization. The results indicate that this optimization offers 8% more miss coverage, 5% lower overprediction, and 2% higher speedup, as compared to CBBR-Global. Note that CBBR-Global is not a prior work and we include it to evaluate PBS' successor selection mechanism.

5 Conclusion

In this paper, we evaluated a classical basic-block reordering technique and showed that if accompanied by a next-line prefetcher, it can significantly reduce the number of instruction cache misses and boost performance. We also proposed a new reordering technique that unlike the classical one considers both what is best for the current basic block and the basic block that will follow the current one. We showed that if the proposed reordering technique, named PBS, is accompanied by a next-line prefetcher and applies to a scope larger than a procedure, it offers a similar performance improvement as the state-of-the-art hardware instruction prefetcher with over 200 KB of per-core storage.

Acknowledgments

The second author is supported in part by a grant from Iran National Science Foundation (INSF).

References

[1] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory Hierarchy for Web Search," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2018, pp. 643–656.

[2] M. Bakhshalipour, P. Lotfi-Kamran, A. Mazloumi, F. Samandi, M. Naderan-Tahan, M. Modarressi, and H. Sarbazi-Azad, "Fast Data Delivery for Many-Core Processors," *IEEE Transactions on Computers (TC)*, vol. 67, no. 10, pp. 1416–1429, Oct. 2018.

[3] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino Temporal Data Prefetcher," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2018, pp. 131–142.

[4] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of Hardware Data Prefetchers on Server Processors," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 52:1–52:29, Jun. 2019.

[5] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive Instruction Fetch," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2011, pp. 152–162.

[6] S. J. Hartley, "Compile-Time Program Restructuring in Multiprogrammed Virtual Memory Systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 11, pp. 1640–1644, Nov. 1988.

[7] W.-m. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Jun. 1989, pp. 242–251.

[8] Intel, "Intel Xeon Processor E7-4850 v4," https://ark.intel.com/products/93806/.

[9] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a Warehouse-scale Computer," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Jun. 2015, pp. 158–169.

[10] R. Kumar, B. Grot, and V. Nagarajan, "Blasting Through the Front-End Bottleneck with Shotgun," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2018, pp. 30–42.

[11] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A Metadata-Free Architecture for Control Flow Delivery," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2017, pp. 493–504.

[12] S. McFarling, "Program Optimization for Instruction Caches," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 1989, pp. 183–191.

[13] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, Jun. 1990, pp. 16–27.

[14] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero, "Code Layout Optimizations for Transaction Processing Workloads," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Jul. 2001, pp. 155–164.

[15] A. Ramirez, J. L. Larriba-Pey, C. Navarro, X. Serrano, M. Valero, and J. Torrellas, "Optimization of Instruction Fetch for Decision Support Workloads," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, Sep. 1999, pp. 238–245.

[16] W. J. Schmidt, R. R. Roediger, C. S. Mestad, B. Mendelson, I. Shavit-Lottem, and V. Bortnikov-Sitnitsky, "Profile-directed Restructuring of Operating System Code," *IBM Systems Journal*, vol. 37, no. 2, pp. 270–297, Apr. 1998.

[17] A. Seznec and P. Michaud, "A Case for (Partially)-TAgged GEometric History Length Branch Prediction," *Journal of Instruction-Level Parallelism (JILP)*, vol. 8, pp. 1–23, Jan. 2006.

[18] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978.

[19] J. Torrellas, C. Xia, and R. Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 1995, pp. 360–369.

[20] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, July-August 2006.

[21] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Jun. 2003, pp. 84–97.

[22] C. Xia and J. Torrellas, "Instruction Prefetching of Systems Codes with Layout Optimized for Reduced Cache Misses," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, May 1996, pp. 271–282.