Exploiting Procedure Level Locality to Reduce Instruction Cache Misses

Ravi V. Batchu   Daniel A. Jiménez

Department of Computer Science
Rutgers University
{batchu,djimenez}@cs.rutgers.edu

Abstract

High instruction fetch bandwidth is essential for high performance in today's wide-issue out-of-order processors. Instruction caches must provide a low miss rate as well as low latency. We introduce Procedure Level Relocation*, a class of dynamic feedback-directed optimizations that substantially reduce the instruction cache miss rate by exploiting the temporal locality of procedure usage. Based on the observation that half of all procedures executed are at most 128 bytes in length, we present a* Small Procedure Cache*, a small and fast explicitly managed memory for storing small procedures. We show that Procedure Level Relocation into a Small Procedure Cache reduces the instruction cache miss rate by an average of 15%.*

1 Introduction

Modern wide-issue processors require high instruction fetch bandwidth to fully utilize their resources and achieve high performance. However, the trend over the past few decades has been a steadily increasing gap between the processor cycle time and memory latency. Therefore it is crucial to reduce the instruction cache miss rate.

At a basic level, programs are written as groups of procedures. The object code generated by the compiler, therefore, has groups of instructions corresponding to each procedure in the high level language. There is a temporal locality in the usage of procedures [5, 4]. Procedures accessed at a certain point in time during execution tend to be accessed again in the future.

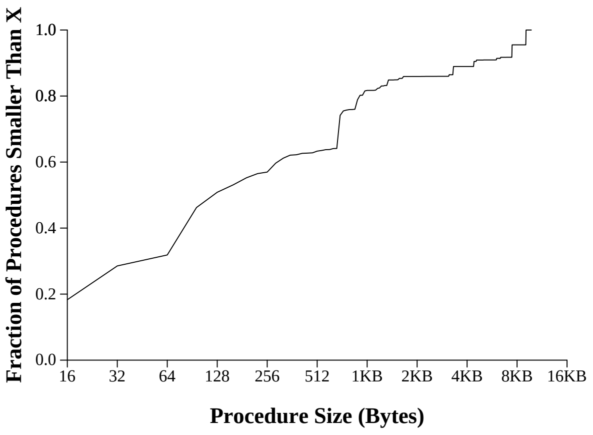

Figure 1. Half of all procedure invocations call a procedure that is at most 128 bytes.

In this paper we introduce the *Small Procedure Cache* (SPC), a hardware/software mechanism that exploits the temporal locality in the usage of procedures by caching frequently used procedures on faster memory which is closer to the processor core. Since it is possible that large procedures from which only a few instructions are accessed can have a detrimental effect on our strategy by unnecessarily occupying more expensive memory, we restrict our strategy to procedures which are smaller than a certain size. Most procedures are small enough so that our strategy does find candidates for relocation. Figure 1 shows the relative frequency with which procedures of less than a certain size are executed. Half of all procedure executed are less than

128 bytes. Thus, there is a large opportunity to exploit procedure-level locality using a small, fast memory that caches small procedures.

The SPC is located adjacent to the L1 cache, occupying a small portion of the physical address space. The physical memory references generated by the Memory Management Unit (MMU) from the virtual memory references issued by the core are compared with a fixed value to determine whether the addressed location is in SPC or the I-cache. The reference is then sent to the appropriate structure. In addition there is a small portion of code which is executed very infrequently, after every 100 million instructions to relocate the frequently accessed procedures.

This paper makes the following contributions:

1. We introduce Procedure Level Relocation (PLR), a class of dynamic feedback-directed optimizations that exploit locality at the procedure level.

2. We present the Small Procedure Cache, an instance of PLR that relocates frequently used small procedures to an explicitly managed fast memory.

3. We show that the Small Procedure Cache optimization reduces instruction cache miss rates 15% on average versus a traditional instruction cache with the same hardware budget over a set benchmarks representative of real-world workloads.

This paper is organized as follows: Section 2 gives an overview of the related work in the microarchitecture and feedback-directed optimization literature. Section 3 describes the Small Procedure Cache in detail. Section 4 describes our experimental methodology for simulating the Small Procedure Cache. Section 4.8 describes the benchmarks and inputs used for this study. Section 6 concludes and points to future research on Procedure Level Relocation.

2 Related Work

In this paper we present a dynamic optimization technique which exploits temporal locality in the usage of procedures to reduce instruction cache misses. While we know of no prior attempt to optimize programs by relocating procedures at run time, there have been efforts to re-order procedures at compile time and there have also been efforts to exploit program locality at run time at other levels of granularity.

2.1 Static procedure re-ordering techniques

Hatfield and Gerald [10], Ferrari [12], McFarling [18], Pettis and Hanson [20], and Gloy and Smith [13] have presented methods to rearrange the procedures to improve locality based on profile data.

Most of these use profile data in the form of a *weighted call graph* (WCG) where a weighted edge between two nodes represents the frequency with which one procedure calls another. By rearranging the program so that procedures with strong connections are placed adjacent to one another, conflict misses can be avoided.

2.2 Static vs. dynamic optimization

The effectiveness of static code layout techniques depends on the stability of profile data from one run to the next. Producing an input set representative of the workloads in real life can often be very difficult or impossible, especially when the application offers multiple functionalities. Modern applications also use Dynamically Loaded Libraries (DLLs). Application software is often shipped as a collection of DLLs. Dynamic linking imposes limits on compile time code layout strategies. Dynamic code generation environments, like Java virtual machines, also make static code layout techniques impractical. Thus, our research proposes to do procedure relocation at run-time, rather than at compile time.

2.3 Dynamic binary translation

There are several efforts involving code transformation during run time. Dynamic binary translation systems run binaries compiled for one platform on a totally different platform [16, 11, 8, 25, 22] The key factor in all of these technologies is the judicious choice of sufficiently frequently executed pieces of code for translation into native code, while interpreting the less frequently used portions of the program.

2.4 Dynamic code placement

Other techniques have been proposed for dynamic code placement. Chen and Leupen [6] present *just in time code layout*, which copies procedures into memory when they are first invoked, resulting in a reduction in the footprint of the executable by 50%.

2.5 Dynamic optimization

There are also dynamic optimization systems which exploit instruction locality to aggressively optimize at run time. Dynamo [1] and Mojo [7] are both user level software efforts to optimize programs executing on the HPUX/PA-RISC and Windows/x86 platforms, respectively. Replay [19] is a new processor framework that supports dynamic optimization, in hardware. Kistler and Franz [15] presented a dynamic optimization system by utilizing the idle time and dynamically collecting execution profiles.

The above techniques for dynamically optimizing code involve disassembling the binary executable followed by aggressive optimization of code. Because of the high overhead, they have to be fairly selective to ensure that the code which is optimized would in fact be repeatedly executed in the future. There are cases for which Dynamo and Morph "bail out," i.e., stop optimizing the binary and just execute it normally. Simpler heuristics like procedure caching, having a lesser overhead, would be less susceptible to such failures.

2.6 Trace Cache

A trace cache is a specialized instruction cache that exploits instruction locality by organizing instructions in the order they are executed, rather than in their static program order[21].

One disadvantage of trace cache is code duplication [14, Page 448]. Due to conditional branches, different hot paths often share the same blocks. This results in the duplication of code in the trace cache. The amount of duplication grows exponentially with the trace length and is an obstacle in building longer traces. Other costs, like trace identification and address mapping, also increase with the trace length. While the outcome of a direct conditional branch provides a good hint for the next time the same instruction is executed, the same is not true of indirect branches and indirect calls.

Unlike schemes for procedure re-ordering, trace caches, operating with physical addresses[1], cannot relieve the pressure on TLB since the stream of virtual addresses remains unaltered with the addition of a trace cache.

3 Procedure Level Relocation

Procedure Level Relocation (PLR) is a dynamic feedback-directed optimization that exploits temporal locality in the usage of small procedures. PLR relocates heavily used small procedures into a *Small Procedure Cache* (SPC), a portion of memory that resides on the same physical structure as the L1 instruction cache. The SPC operates with the same latency and bandwidth as that of the L1 cache. The SPC is essentially an explicitly managed instruction cache used by the operating system for storing frequently used procedures.

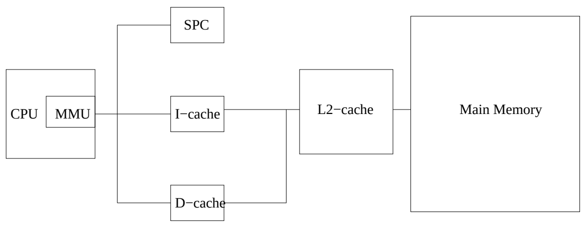

Figure 2. Procedure Cache in Memory Hierarchy

[1]Trace caches accessed by virtual addresses are not promising. Multiple processes can have the same virtual address for different locations. Combining the virtual address and process ID results in the duplication of the code in dynamically linked libraries

3.1 Instruction reference generation

Instruction references to the portion of memory mapped to the SPC are satisfied by the SPC, while other instructions references go through the traditional memory hierarchy. The SPC is located in a fixed area at one end of the physical address space so that checking whether a reference is for the SPC involves a single comparison. The MMU is set up by the kernel so that the SPC appears in the same virtual address space in all the processes. All references are first checked to see whether they can be satisfied by the instruction prefetch buffer and undergo virtual to physical address translation in the Memory Management Unit (MMU) before they are directed to the SPC or the L1 instruction cache.

3.2 Frequency of relocations

There is a trade-off associated with procedure relocation. Relocation results in an additional overhead while improved locality results in better performance. We do relocations only after very long periods of execution, so that the benefit of relocation can be amortized over it. As described in Section 4, the relocation is done by the operating system after every 100 million instructions. We estimate that at this granularity, the overhead from procedure relocation will be far less than that of other operating system functions, such as processor scheduling.

3.3 Relocating procedures

Each procedure keeps a count of how often it has been invoked. After a *relocation epoch* of 100 million instructions has passed, the operating system runs an algorithm to perform procedure relocations. The following is an overview of the steps that are taken:

1. Procedures are prioritized for relocation based on an estimate of their temporal localities.
2. Procedures are copied contiguously in priority order until the SPC is full or there are no more procedures.
3. Each old instance of a procedure is patched so that the next time it is invoked, the caller will be patched to invoke the new copy in the SPC. Note that this happens only once per call site; each subsequent invocation is automatically directed to the new copy.
4. Call sites of procedures residing in the previous generation of the SPC are patched back to call their original targets.

3.4 Relocation policy

The 100 million instruction relocation epoch is divided into four quarters. At the time of relocation, procedures are selected based on a priority computed by dividing the total frequency of their usage in the relocation epoch by the procedure size. Procedures are then chosen using an estimate of their future temporal locality, based on their activity in the previous four quarters. First, procedures which were accessed in all four quarters of the relocation are copied into SPC in the order of decreasing priority. If there is any remaining space, procedures which were used in the last quarter and at least two other quarters are copied in the order of decreasing priority. In the remaining space, procedures which used in the last quarter are copied. Finally in any left over space, procedures are selected solely on their priority. In practice, the SPC is usually full after the first round of copies.

We refill the SPC on each relocation from scratch. This is equivalent to removing the procedures from SPC and placing them back. Since procedures do not have a fixed size this approach makes it easy to decide the placement of procedures in the SPC. Refilling the SPC has the downside that the machine would have to fix the call sites again. However, since the relocation is a very infrequent event, this approach is not likely to have a great impact. Our approach, in this matter, is similar to the complete flush of the fragment cache in Dynamo [1].

3.5 Implementation details

Implementation of the relocation algorithm brings up a number of details:

Patch-up code To handle calls to relocated procedures, *patch-up* code is inserted into the beginning of the original copy of the procedure. This patch-up code redirects call sites to the new location in the SPC. This lazy approach to updating the call site only after it is actually encountered during execution saves considerable overhead compared to an approach which corrects the call sites immediately after relocating the procedure. This is because not all call sites for the relocated procedure are likely to be encountered. There has also been considerable amount of work [3] demonstrating execution path locality which indicates that only a small fraction of the call sites really need to be patched. The patch up code also maintains a record of the call sites it modified to facilitate restoration of the call site when the procedure is relocated back to the original location.

Replacement After a relocation phase, call sites must be redirected back to the original instances of procedures that were in the previous generation of the SPC. When the call site was originally modified, it was placed on a queue. During relocation, call sites on the queue are visited and restored. Since only a few call sites will have been patched, this queue will be short and the redirection phase will be quick.

Code expansion When procedures are selected for relocation the original area where they appear is left intact and another copy is created in the SPC. Leaving the original code intact allows us to simply overwrite the copy in the SPC when it is replaced with another procedure. The above strategy of copying rather than moving means that there would be two copies of some procedures. Nevertheless, code expansion is minimal because the size of the SPC is on the order of one kilobyte. There is also likely to be a beneficial effect on the availability of physical page frames. Code which is executed frequently would be copied into the SPC, making references to the original copy unnecessary and (possibly) freeing up the physical page frame containing the original copy.

Maintaining proper control transfer To maintain program semantics in the presence of dynamic relocation, control instructions should continue to transfer control to the intended destination. Position independent code is generated using pc-relative addressing. This would ensure that all the intra-procedure control transfers would operate correctly. Returns to the relocated procedures are handled by examining the stack at the time of relocations. Any returns to procedures which get relocated (either into the SPC or back to their original location) are updated at relocation time.

3.6 System wide relocation

Previous work [5, 4] has shown that besides the fact there exists a working set of procedures for each program, the working set size varies as programs execute. While dynamically linked libraries used simultaneously by multiple processes may be mapped into different areas of the individual virtual address space, typically using a system call like `mmap`, there is actually only a single copy of the library in the physical memory. Therefore, the SPC is used for caching procedures from all the processes executing on the machine.

4 Methodology

In this Section, we describe our methodology for studying procedure level relocation and simulating the procedure cache.

4.1 Simulator code base

Our simulator is based on the Bochs x86 simulator[17]. Bochs simulates the entire machine platform so that it can support the execution of a complete operating system and applications that run on it. This allows us to include the effects of the execution of the operating system code due to context switches, system calls and kernel daemons running in the background. Modern applications (e.g. mozilla) are often shipped as multiple executables running as multiple processes. Almost all applications use dynamic linking. Our simulation strategy can support all of the above scenarios. For all of our experiments we simulate the Red Hat Linux operating system version 6.0.

Name	Description	Input Data Set	Instructions executed in 100 Million
`186.crafty`	Plays chess using alpha-beta search	train input crafty.in	307
`176.gcc`	gcc compiler for the M88100	train input cp-decl.i	35
`253.perlbmk`	Perl v5.005_03 interpreter	train input diffmail.in	307
`255.vortex`	Object oriented database program	train input lendian.raw	148
`mozilla`	A web browser	Copy of HPCA web page	25

Table 1. Benchmarks and inputs used for this study

4.2 Instruction cache simulations

To evaluate the effects of dynamic procedure relocation we performed a complete machine simulation at the instruction level. Our simulator performs the following actions:

1. It captures the sequence of memory references that occur while running the workload on a normal machine,

2. It keeps track of the location of all procedures and processes as the machine runs.

3. It monitors the usage of procedures in each relocation epoch and simulates the relocation of procedures into the procedure cache.

4. It builds a new sequence of memory references based on the sequence in step (1) and the knowledge of the location of all procedures in the presence of dynamic procedure relocation.

5. It then feeds the two sequences to a cache simulator simulating separate instruction caches to estimate the resulting miss rates.

Next, we describe how we achieve the above steps.

4.3 Identifying procedures

As the simulator runs, it has to identify the current procedure being executed based on the contents of the simulated instruction pointer. This requires the knowledge of the location of all the procedures in the virtual address space of the process being simulated. Hence the simulator has to keep track of the location of all the procedures in the corresponding virtual address space and the associated processes. Events like process creation, process termination, procedure resolution when a dynamically linked procedure is first called, and the loading of code when an application is initially launched are all very high level and involve several machine instructions. We instrument the Linux kernel to inform our simulation infrastructure when one of these high-level events occurs. Based on the observations from Figure 1, we relocate only those procedures with a size at most 128 bytes.

4.4 Accounting for dynamically linked libraries

Dynamically linked procedures do not appear in the executable. When the application is initially launched, all the call sites for a dynamically linked procedure contain a call to a Procedure Linkage Table (PLT) stub which directs the control to a *resolver* in the run time linker. The resolver finds out the location of the procedure being called and patches the program so that future calls go directly to the dynamically linked procedure. We modified the resolver, a component of `glibc`, so that the simulator could detect the existence of the newly linked procedure. Our changes, both in the kernel and the run time linker, are very minimal.

4.5 Cache simulator

The information about the memory references is available in the existing bochs data structures and our simulator accesses them to capture the memory reference stream on the normal machine and transform it to reflect our optimization. We developed a cache simulator to estimate the miss rates for the different memory reference streams. Since the LRU replacement strategy is expensive to implement, we selected a replacement strategy called the clock algorithm (or the second chance algorithm) for the replacement strategy. This strategy results in low miss rates like the LRU and has low implementation overhead, comparable to that of FIFO. The simulator counts the number of instructions executed in the simulated machine and after every 100 million instructions it simulates a relocation by updating the address of the relocated procedures. The relocations are based on the usage counts for the procedures which are obtained during the simulated execution.

4.6 Cache details

For this study, the base cache configuration is a 65KB instruction cache with 64 byte blocks and 2-way set associativity. The SPC configuration is a 64KB instruction cache with 64 byte blocks and 2-way set associativity plus a 1KB small procedure cache. Cache miss rates are reported as the number of times the I-cache is filled from the L2 cache divided by the total number of blocks requested from the first-level structure (either I-cache or SPC). The simulated CPU prefetches an entire block from the first level structure into a prefetch buffer so that subsequent references to the same block without intervening references to other blocks are all counted as one access.

4.7 Simulated environment

For our experiments we simulate a Pentium-class workstation with an NE2000 network interface card, a VGA monitor, 2 ATA channels connected to 3 IDE hard drives and a CD-ROM drive. The simulated computer is loaded with a Red Hat 6.0 distribution and the workloads are launched in a terminal while running the X Window System. Cache simulation makes the simulation many orders of magnitude slower. We turn on the cache simulation only when we start executing the workloads on the simulated machines.

4.8 Benchmarks

To evaluate the effects of PLR on the instruction cache miss rate we selected those integer benchmarks from SPEC CPU2000 suite that are known to have large instruction footprints (gcc, vortex, perl and crafty) as well as Mozilla. The four SPEC benchmarks have been shown to incur a 25% to 30% performance penalty due to the limited space in the I-cache while the rest of the benchmarks in this suite have a performance penalty less than 3% [24]. Previous work [26, 24] on reducing instruction cache misses have used this same subset of SPEC benchmarks for their studies. We believe that these benchmarks characterize real-world applications that would be used in a production environment.

One of the prerequisites for running a workload on our simulation platform is that the executable(s) constituting the workload have a symbol table in them so that information about procedures can be extracted. This is not true of the applications like Netscape which comes pre-packaged in the Red Hat distribution on our simulated box. Since the simulated computer runs much more slowly than a real machine, building large programs like mozilla on the simulator is not feasible. We created an environment similar to that on the simulated box to build the mozilla binaries which had symbol tables in them. Preparing large real world applications which are likely to have large instruction footprints, for the simulation platform is a non-trivial task.

Table 1 describes each workload. They were all run in an X windows environment in the simulated machine. The cache simulation was only turned on when these workloads were running.

5 Results

In this Section, we describe the results of our simulation experiments with the Small Procedure Cache.

5.1 Miss rates

Figure 3 shows the instruction miss rates for two configurations: a 65KB instruction cache and a 64KB instruction cache with a 1KB SPC. Miss rates for each benchmark as well as the arithmetic mean miss rate are given. In each case, the SPC yields a lower miss rate than the larger instruction cache alone. On average, the SPC configuration yields a miss rate of 1.82%, while the larger instruction cache alone yields a miss rate of 2.15%.

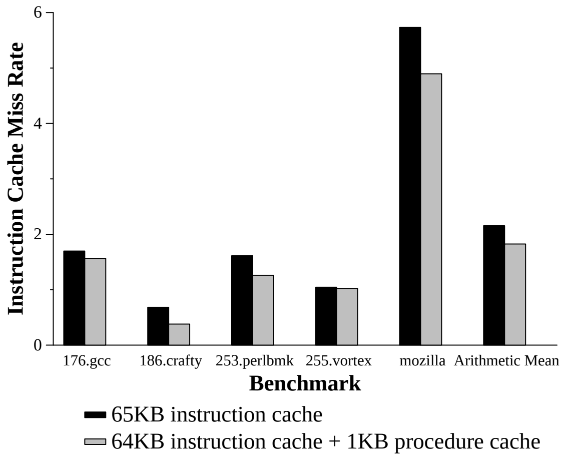

Figure 3.

Figure 4 gives the percentage decrease in miss rates for the two configurations, showing the improvement given by the SPC. The largest improvement is in `186.crafty`, where the miss rate is decrease by 44%, from 0.68% to 0.38%. For `mozilla`, the benchmark with the largest miss rates, the SPC gives a 15% decrease in miss rates, from 5.73% to 4.90%.

5.2 Discussion

Programs have been shown to have large scale behavior extending over billions of instructions [23]. A combination of the SPC and L1 cache performs better than a larger L1 cache because a more intelligent scheme is used in deciding how to retain information in the faster and more expensive memory at the L1 cache level. During the operation of a complete machine, there is "noise" in the references due to the sporadic execution of code in the

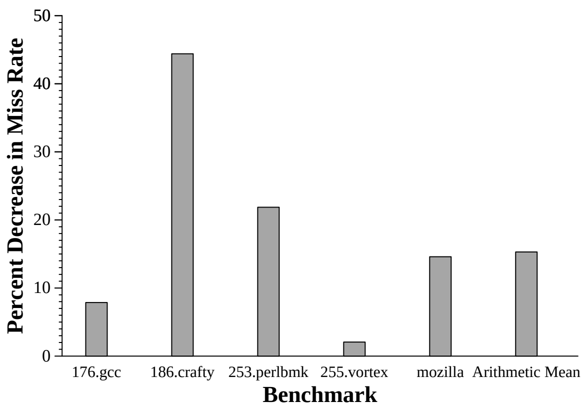

Figure 4.

interrupt service routines, context switch code and background processes. References from such code in a plain L1 cache can result in conflict misses which replace heavily used code. A combination of L1 cache and SPC turns out to be less susceptible to such vulnerabilities.

The usual cache generally operates at a fixed granularity; the one we compare against in this paper uses a 64 byte cache line like the I-cache in Pentium 4. This can result in wastage of space in the I-cache if only a few bytes are used in the cache line. On the other hand in the SPC we pack procedures closely. As mentioned earlier more than half of the invocations of the programs we studied were made to small procedures. While the SPEC benchmarks we studied were statically linked, in real programs there are likely to be more small procedures invocations because of the stubs (consisting of only 3 instructions and 16 bytes in the case of x86) for dynamically linked procedures.

6 Conclusions

In this paper we present a system-wide dynamic feedback-directed technique to do Procedure Level Relocation (PLR). To the best of our knowledge this is the first such effort to exploit locality at the level of procedures. We show that a 64KB instruction cache augmented with a 1KB Small Procedure Cache has a miss rate which is on average 15% lower than an L1 cache which is 65KB in size.

We simulate the complete machine in enough detail to bring up an operating system. This enables us to take into account all the memory references generated and select heavily used small procedures for relocation from any application, dynamic library or the underlying kernel.

Much work remains to be done to explore all the merits of exploiting locality at the level of procedures. We intend to explore the benefits of better placement strategies for procedures within the SPC. Minimizing the number of cache lines each procedure occupies in the SPC could improve the conventional sequential prefetch of instruction into the CPU core. Another promising area for future research is to vary the length of the relocation epoch. It can be dynamically changed by automatically detecting the collective phase behavior of all the processes. Techniques which adapt to the phase behavior have been explored in other areas like clustered microarchitectures [2] as well as multi-configuration I-caches [9].

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[2] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proceedings of the 30th annual international symposium on Computer Architecture*, 2003.

[3] Thomas Ball and James R. Larus. Using paths to measure, explain and enhance program behavior. *IEEE Computer*, 33(7):57–65, July 2000.

[4] Ravi Batchu and Saul Levy. Working sets at function level. Technical Report DCS TR-480, Department of Computer Science, Rutgers University, 2002.

[5] Ravi Batchu, Saul Levy, and Miles Murdocca. A study of program behavior to establish temporal locality at function level. Technical Report DCS TR-475, Department of Computer Science, Rutgers University, 2002.

[6] J. Bradley Chen and Bradley D. D. Leupen. Improving instruction locality with Just-In-Time code layout. In *Proceedings of the USENIX Windows NT Workshop*, pages 25–32. USENIX Association, August 1997.

[7] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[8] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

[9] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th international symposium on Computer Architecture*, 2002.

[10] D.J.Hatfield and J.Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[11] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Transaction on Computers*, 50(6):529–548, June 2001.

[12] Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, November 1974.

[13] Nikolas Gloy and Michael D. Smith. Procedure placement using Temporal-Ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, September 1999.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3 edition, May 2002.

[15] Thomas Kistler and Michael Franz. Continuous program optimization: Design and evaluation. *IEEE Transaction on Computers*, 50(6):549–566, June 2001.

[16] Alexander Klaiber. The technology behind Crusoe(tm) processors. Transmeta White Paper, January 2000.

[17] Kevin Lawton. Bochs: The open source IA-32 emulation project. http://bochs.sourceforge.net.

[18] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1989.

[19] Sanjay J. Patel and Steven S. Lumetta. rePLay: a hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

[20] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[21] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, 1997.

[22] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *Proceedings of the IEEE 2001 Workshop on Binary Translation*, September 2001.

[23] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the Tenth international conference on architectural support for programming languages and operating systems*, pages 45–57, 2002.

[24] Viji Srinivasan, Edward S. Davidson, Gary S. Tyson, Mark J. Charney, and Thomas R. Puzak. Branch history guided instruction prefetching. In *Proceedings of the Seventh International Conference on High Performance Computer Architecture (HPCA)*, pages 291–300, January 2001.

[25] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS conference on measurement and modeling of computer systems*, pages 68–79, 1996.

[26] Yi Zhang, Steve Haga, and Rajeev Barua. Execution history guided instruction prefetching. In *Proceedings of the Sixteenth ACM International Conference on Supercomputing (ICS)*, June 2002.