Efficient Code Caching to Improve Performance and Energy Consumption for Java Applications

Yu Sun, Wei Zhang
Dept. of Electrical and Computer Engineering, Southern Illinois University Carbondale
Carbondale, IL 62901
sunyu@engr.siu.edu, zhang@engr.siu.edu

ABSTRACT

Java applications rely on Just-In-Time (JIT) compilers or adaptive compilers to generate and optimize binary code at runtime to boost performance. In conventional Java Virtual Machines (JVM), however, the binary code is typically written into the data cache, and then is loaded into the instruction cache through the shared L2 cache or memory, which is not efficient in terms of both time and energy. In this paper, we study three hardware-based code caching strategies to write and read the dynamically generated code faster and more energy-efficiently. Our experimental results indicate that writing code directly into the instruction cache can improve the performance of a variety of Java applications by 9.6% on average, and up to 42.9%. Also, the overall energy dissipation of these Java programs can be reduced by 6% on average.

Categories and Subject Descriptors

B.3.2 [**HARDWARE**]: Memory StructuresDesign StylesCache Memories; D.3.4 [**SOFTWARE**]: Programming LanguagesCode Generation

General Terms

Design,Performance

Keywords

Instruction Cache, JIT Compiler, Code Generation, Code Caching, Java Virtual Machine

1. INTRODUCTION

In dynamic optimization systems such as Java Virtual Machines (JVM), the code is typically compiled and stored while the program is being executes. A typical path of generating and executing Java code in a Java Virtual Machine [1] is shown in Figure 1. The dynamic/adaptive compiler treats the generated code as normal data and stores them back to the memory through the data cache. When this binary code segment is needed by the processor, it is then loaded from the memory to the instruction cache. Such a code generation and fetching path, however, is not efficient due to the following reasons.

1. Since the binary code is stored into both data and instruction caches, the aggregated cache space for the binary code is doubled.
2. Instruction cache misses always occur when the binary code is invoked for the first time, since the code is written to the data cache and cannot be directly fetched from the instruction cache.
3. The L1 data cache may be polluted by the binary code generated. Unlike a static compilation system, where all the binary codes are generated before the program is executed, in a JVM, the binary code is generated at runtime, while the program also read and write data at the same time in the same data cache. As a result, writing binary code into the data cache may evict useful data from the data cache, leading to worse data cache locality and longer execution time.
4. Cache synchronization is needed for each dynamically generated code segment so as to guarantee the data consistency among the data cache, the memory and the instruction cache, resulting in substantial CPU stalls and flushing of cache blocks while the binary code is generated.

This inefficient path is particularly problematic for embedded Java applications, in which both performance and energy efficiency are crucial [19, 20]. In this paper, we study three different techniques that aim at improving the efficiency of Java binary code generation and fetching, including 1) I-cache D-cache Data Path (IDDP), 2) Dynamically-generated Instruction Cache (DIC), and 3) Writable Instruction Cache (WIC). Specifically, IDDP adds a path between the onchip instruction and data caches, which allows the processor to access the data cache directly (rather than the L2 or memory) after an instruction cache miss. The motivation is that it is very likely to find the recently generated code from the data cache. DIC is an additional cache used to only store the dynamically generated code by the JIT compiler, which can be accessed by the processor in parallel to the instruction cache. WIC enables write operations to the instruction cache. Thus all the dynamically generated code can be directly written into and read from the instruction

CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.

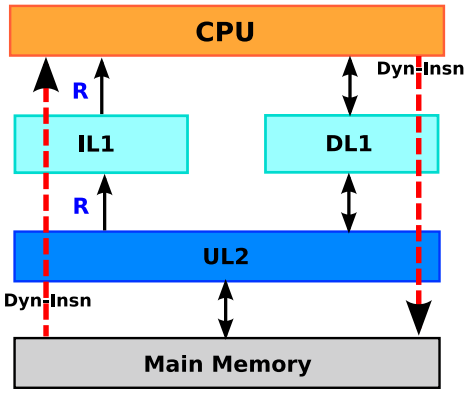

Figure 1: A typical path of generating and executing Java code.

cache without touching the data cache. More details of these three schemes can be found at Section 3. Our evaluation of these three code caching optimizations indicates that WIC is the most effective and efficient approach to improving the performance and energy efficiency for Java applications.

The rest part of this paper is organized as follows. In Section 2, we discuss the related work. We present the design of three code cache optimization schemes in Section 3. Section 4 shows the experimental results. Finally, we make concluding remarks in Section 5.

2. RELATED WORK

There have been a number of research efforts to optimize the code cache for improving performance [2, 3] or energy efficiency [4]. Hazelwood and Smith [2] proposed generational cache management of code traces in dynamic optimizers, which categorizes code traces based on their expected lifetime. The advantage of the generational code cache management is to filter out short-lived code traces to reduce fragmentation. As a result, long-lived code traces can be kept longer to benefit the performance. Hazelwood and Smith [3] also showed that using a medium-grained FIFO (First-In, First-Out) eviction policy for code caches can effectively balance cache management complexity and cache miss rates. Compared with the above efforts that focus on enhancing the code cache management policies, our study aims at bypassing the inefficient path of writing and fetching Java native code in conventional JVMs, which is orthogonal and complementary to the prior work in [2, 3].

Chen et al. [4] studied energy-efficient code cache management techniques for memory-constrained Java devices. Their work assumes that in addition to on-chip instruction and data caches, there is an independent on-chip code cache to store Java native code, which is similar to the DIC scheme studied in this paper. Chen's work [4] showed that selective use of code cache can reduce energy dissipation substantially as compared to the pure interpreted and the pure (performance-oriented) compilation strategies for memory-constrained Java devices. Compared to this study, our work on the WIC cache does not assume the existence of an independent code cache, which is certainly more expensive in terms of the hardware cost. Nevertheless, the WIC cache proposed in this paper can benefit both performance and energy dissipation.

Traditionally, there have been many studies on organizing instruction caches and data caches efficiently [5, 6, 7] to achieve better performance. Generally, the instruction and data caches can be either separated or unified. Computer architectures have proposed a variety of smart cache architectures to exploit the locality of instruction and data accesses to maximize performance, for instance victim caches [8], column-associative caches [9], cache bypassing [10] and split caches [11, 12, 13, 14], etc. However, all these studies are focused on programs that are statically compiled (e.g. C programs). By comparison, this paper studies the WIC cache that is more efficient for dynamically compiled applications such as Java programs. Also, unlike traditional cache architectures (in which the instruction caches are typically read-only), the WIC cache enables write operations directly to the instruction cache, which is suitable to store binary code generated at runtime.

3. CODE CACHE OPTIMIZATION SCHEMES

3.1 IDDP (I-cache D-cache Data Path)

The path of storing and loading dynamically-generated code in IDDP is shown in Figure 2. As we can see, an additional data path is added between the L1 instruction cache and the L1 data cache. With this fast inter-cache data path, the processor can access the L1 data cache directly for each L1 instruction cache miss. If the instruction is found in the L1 data cache (i.e., a L1 instruction cache miss but a L1 data cache hit), the instruction can be efficiently transferred to the L1 instruction cache. However, if the processor cannot find the needed instruction from the L1 data cache (i.e. a L1 instruction cache miss and a L1 data cache miss), then the L2 cache or the main memory has to be accessed to return the code to the L1 instruction cache as usual. The motivation behind the IDDP scheme is that the binary code needed by the instruction cache is most likely in the L1 data cache when it is needed, since this code has been just written into the L1 data cache. By providing a short and fast inter-cache data path, it is expected that the cost of handling instruction cache misses caused by the dynamically generated Java binary code can be reduced. In our evaluation, we assume that for the IDDP scheme, an instruction cache hit takes 1 cycles, and an instruction cache miss but a data cache hit takes 2 cycles (i.e., we assume that fetching instructions from the data cache directly through the I-cache D-cache data path takes 1 cycle).

3.2 DIC (Dynamically-generated Instruction Cache)

In the DIC scheme, an extra cache structure (i.e. the DIC cache) is added into the system, which is in parallel with the level 1 instruction and data caches, as shown in Figure 3. The DIC cache has access control that allows only dynamically-generated Java code to be stored, while at the same time the Java native code is also written to the L1 data cache. Nevertheless, the DIC cache forbids the read or replacement operations from low-level caches or main memory. As a result, the DIC cache is guaranteed to always contain the most updated Java native code generated by the JIT compiler. When the binary code is needed, the processor can fetch the code from the L1-I cache and the DIC cache simultaneously. If there is a DIC cache hit, the instructions

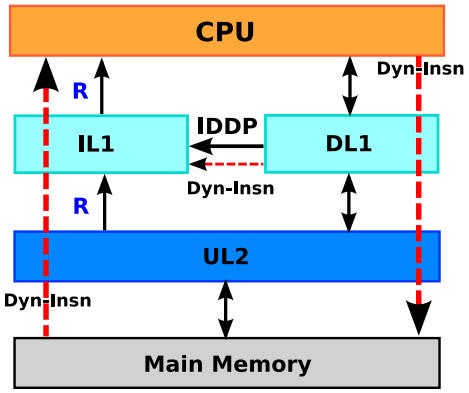

Figure 2: The path of Java binary code generation and fetching in the IDDP scheme.

will be immediately returned to the CPU. Otherwise, a L1-I cache hit or a L1-I cache miss may occur to supply the needed instructions to the pipeline (note that a DIC cache does not handle a DIC cache miss as it is not connected to the L2 cache).

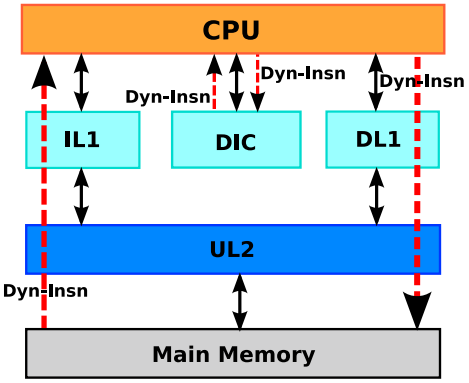

Figure 3: The path of Java binary code generation and fetching in the DIC scheme.

3.3 WIC (Writable Instruction Cache)

The WIC cache is another way to simplify the Java binary code generation and fetching path by enabling writes to the L1 instruction cache, as depicted in Figure 4. Traditionally, the L1 instruction cache is read-only because the instructions are typically already generated and stored in the memory before they are executed, which usually do not need to be modified at runtime. However, in a dynamic optimizing system such as JVM, the byte codes need to be compiled and optimized at runtime to generate high-quality Java native codes. Thus it is desirable to enable write operations to the instruction cache so that the binary code can be directly written into and read from the instruction cache without having to bother the L1 data cache.

The WIC cache can not only reduce the pollution to the L1 data cache, but also make the Java binary codes closer to the fetch unit that needs to use them (as can be seen from Figure 5 (b)). Moreover, without a WIC cache, the L1 data cache has to be synchronized with the memory and the L1 instruction cache to ensure that all the recently generated Java binary codes are written back to the main memory and no outdated binary code will stay in the instruction cache after it is recompiled. With the availability of a WIC cache, these expensive data cache synchronization operations can be removed. However, it should be noted that one possible problem of the WIC cache is that writing binary code directly into the IL1 (i.e., the level-1 instruction cache) may evict certain useful instructions from the IL1, possibly leading to some instruction cache misses.

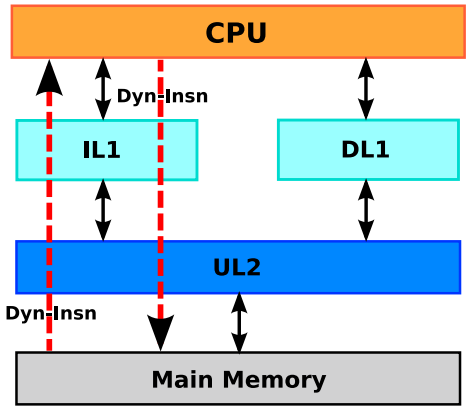

Figure 4: The path of Java binary code generation and fetching in the WIC scheme.

The writable instruction cache functions like a write-back data cache. In other words, only when an instruction is replaced, it is written back to the L2 cache if it is modified. This writeback policy is expected to mitigate the overhead of updating the low-level memory hierarchy caused by writing to the instruction cache.

The architecture support of the WIC cache is depicted in Figure 5, in which Figure 5 (a) shows a 5-stage instruction pipeline of the baseline processor with a traditional instruction cache, and Figure 5 (b) illustrates the modified pipeline of the processor that employs a WIC cache. Compared with the baseline instruction pipeline, the memory stage of the processor with WIC is connected to both the data cache and the WIC cache through a multiplexer. We assume that this processor supports an additional *store* instruction to write *code* (not data) into memory, which is called *Storei* instruction in this paper. Also, we assume an additional control signal called $IsIns$ is used, in conjunction with an existing signal $MemWrite$ (i.e. to enable write operations to the memory), to control the multiplexer. After an *Storei* instruction is decoded, the $IsIns$ signal will be enabled (which will be cleared by other instructions). Since $IsIns$ is $logical-anded$ with the $MemWrite$, the multiplexer will select the WIC cache (instead of the data cache) as the target to write Java binary code. Once $IsIns$ is cleared, the memory stage will write data to the data cache as usual. As we can see from Figure 5 (b), the hardware overhead to support the WIC cache is insignificant.

4. EXPERIMENTS

4.1 Evaluation Methodology

We evaluate the proposed three optimization schemes by using Dynamic SimpleScalar (DSS) simulator [16]. We sim-

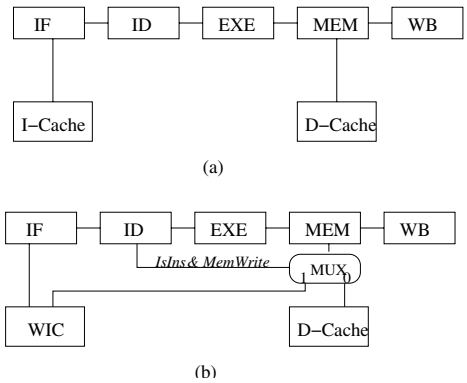

Figure 5: The instruction pipeline for a processor with (a) a traditional instruction cache, (b) the WIC cache.

Hardware	Parameters
L1 I-Cache	16KB, direct-mapped, 32-byte block 1 cycle latency
L1 D-Cache	16KB, 4 way, 32 byte-block 1 cycle latency, LRU
DIC Cache	4KB, fully associative, 32 byte-block 1 cycle latency, LRU
L2 Cache	256KB, 4 way, 64-byte block 6 cycle latency, LRU
Memory	32 cycles, unlimited size

Table 1: Configuration parameters and their values in our base configuration for the simulated processor.

ulate a 4-issue PowerPC-based superscalar processor with 32 integer and 32 floating point registers. The important parameters of memory hierarchy are given in Table 1.

The Java virtual machine running on the simulator is Jikes RVM [18], which uses the baseline compilation mode that compiles every method of Java applications. We extended DSS by fully implementing four cache synchronization instructions used by Jikes RVM, including **dcbst**, **sync**, **icbi** and **isync**, which were simply treated as NOPs in the original DSS.

Two sets of benchmarks are used in our experiments to evaluate the effectiveness of the code cache optimization schemes proposed for Java Virtual Machines. In addition to SPECjvm 98 benchmark suite [15], five smaller Java applications are collected to represent light-load client-side Java applications, for which startup time is crucial. Table 2 gives the description and salient characteristics of these benchmarks.

In our experiments, we compare the performance of different schemes by evaluating the relative performance improvement as defined below.

$$I = \frac{P_{orig} - P_{opt}}{P_{orig}} \quad (1)$$

In the equation above, P_{orig} is the original performance, and P_{opt} stands for the optimized performance, both in terms of the number of CPU clock cycles.

4.2 Comparison of 3 Schemes

Figure 6 shows the performance improvement achieved by the IDDP, DIC and WIC schemes. It is obvious that IDDP gets almost the same performance as the base scheme without using any code cache optimization. While DIC attains better results for several benchmarks, WIC achieves the best performance improvement for most benchmarks. The reasons of these different performance results are the following.

1. WIC is the only scheme that can eliminate the cache synchronization overhead. Moreover, WIC benefits from better L1 data cache performance since no binary code will be written into the L1 data cache.

2. The DIC scheme makes an extra copy of recently generated Java code in the DIC cache, so the number of instruction misses can be reduced. In particular, we find that the DIC cache can reduce almost all instruction cold misses.

3. The IDDP scheme can potentially reduce the instruction cache miss latency by fetching instructions directly from the L1 data cache if it hits in the DL1. However, since the binary code written into the DL1 may be overlapped by data references or other binary code, the number of cold instruction misses that hit in the L1 data cache is relatively too small to have a noticeable impact on the overall performance.

As can be seen in Figure 6, the WIC scheme can boost performance for most of the benchmarks. The only two exceptions are `edge` and `compress`. The reason is that both these two benchmarks have quite small instruction cache miss rate (as well as the data cache miss rate) as shown in Table 2. Therefore, WIC cannot noticeably enhance the overall performance of these two benchmarks by simply reducing the instruction cache misses. On average, the WIC scheme improves the performance by 9.6%, indicating the effectiveness of WIC.

We also observe that generally the WIC cache can benefit small benchmarks more than the SPECjvm 98. The reason is that for larger benchmarks such as SPECjvm 98, the instruction cache miss latency can be amortized by a large number of instruction reuses. Nevertheless, even for SPECjvm 98, we still observe noticeable performance enhancement for most benchmarks, since the WIC cache can both reduce the cache misses and eliminate cache synchronization overhead. More specifically, the average performance improvement of SPECjvm 98 by using the WIC cache is 4.6% (up to 7.8% for `javac`), demonstrating that WIC can effectively boost performance for many Java applications.

Since both IDDP and DIC schemes make insignificant improvement and particularly DIC is quite expensive in terms of hardware overhead, we will focus on the WIC scheme in the rest of this paper.

4.3 Breakdown of Performance Improvement

Figure 7 shows the breakdown of the performance improvement in terms of reducing synchronization and decreasing cache misses for the WIC scheme. As we can see, avoiding cache synchronization is the major factor to improve performance, for both small benchmarks and SPECjvm 98. Specifically, on average 70.2% performance improvement by the WIC cache is due to the elimination of cache synchronization overheads. These results also explain why WIC

Set	Benchmark	Description	I$ Miss Rate	D$ Miss Rate
Small Bench	db_s	Simplified database	1.56%	1.18%
	edge	Image edge detection	0.11%	0.04%
	fft_2d	2D FFT	1.46%	1.30%
	hello	Hello world	1.55%	1.27%
	xml	XML parser	1.65%	1.23%
SpecJVM 98	compress	File compression	0.52%	0.75%
	jess	Java expert system	2.69%	1.85%
	raytrace	Raytrace of dinosaur	2.22%	0.97%
	db	Database	1.77%	1.83%
	javac	Java compiler	2.37%	1.91%
	mpegaudio	Audio en/decoder	1.77%	0.58%
	jack	Java parser generator	3.58%	1.71%

Table 2: Benchmark description and salient characteristics.

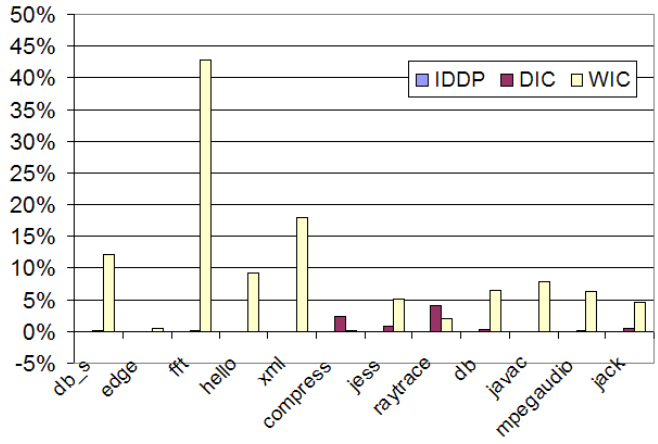

Figure 6: Compare the performance improvement of 3 schemes.

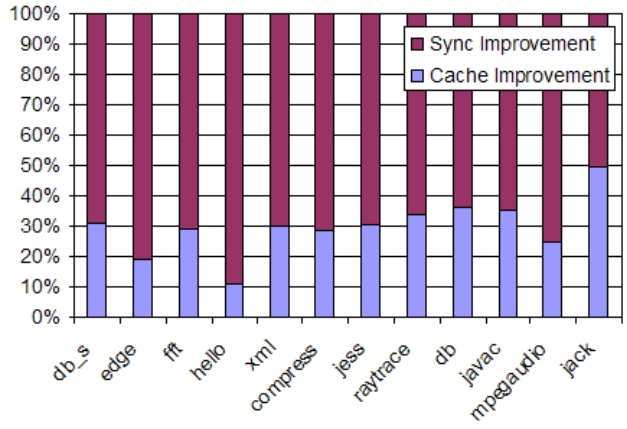

Figure 7: Performance Improvement Distribution

adequately outperforms DIC, which still needs cache synchronization to guarantee that the processor can fetch the correct (i.e. most updated) Java binary code.

4.4 Sensitivity to the Cache Size

We have also studied the effectiveness of WIC for instruction caches with different sizes. Figure 8 shows the normalized execution cycles of the base and WIC schemes with the L1 I-cache size varying from 4KB to 8KB and 16KB, which are normalized with the base with a 4K instruction cache. As we can see, WIC is always superior to the base scheme for instruction caches with various sizes. In particular, WIC seems to be slightly more effective for larger instruction caches. On average, the WIC scheme can increase the base performance by 6.7%, 7.4% and 8.3% for an instruction cache with 4KB, 8KB and 16KB respectively. The reason may lie in the fact that a larger instruction cache can hold more Java binary code that is written directly from the processor, which is likely to be used later to benefit performance. Interestingly, we also find that a 8KB WIC cache can achieve performance very close to or even better than a 16KB instruction cache (without WIC), for instance `fft`, `xml` and `db`, which demonstrates that WIC can be a cost-effective approach to boosting instruction cache performance (i.e. without increasing the cache size).

4.5 Energy Results

We also evaluate the energy consumption of the WIC scheme by using the DSS simulator [16], which reports the energy dissipation results based on the Wattch energy model with all its default parameters [17]. Figure 9 and Figure 10 give the energy consumption and energy-delay product (EDP) of the processor (including the datapath and the memory energy dissipation) for the base and WIC schemes, which are normalized with the energy consumption and EDP respectively of the base, whose configuration has been shown in Table 1. As we can see, except `edge` and `compress`, there is improvement of both energy efficiency and energy-delay product for most of the benchmarks. This is because the WIC cache can avoid the inefficient binary code writing and fetching path in a traditional code cache, which can be translated into energy savings. On average, the WIC scheme reduces the energy dissipation by 6%, and reduces the energy-delay product by 14.5%, indicating that the WIC cache can benefit both performance and energy for Java programs.

Table 3 gives the energy saving of the L1 instruction cache, the L1 data cache, and the L2 cache with the WIC scheme, as compared to the base scheme. As we can see, the WIC scheme can improve the energy efficiency of all these three cache memories. Particularly, the L1 instruction cache energy consumption is reduced because WIC can decrease the cold instruction misses by storing Java binary code directly into the L1 instruction cache. The energy efficiency of the L1 data cache is improved since the WIC scheme can decrease the L1 data cache pollution by avoiding writing binary code directly into the L1-Dcache. Also, we observe that the L2 energy consumption can be reduce more than

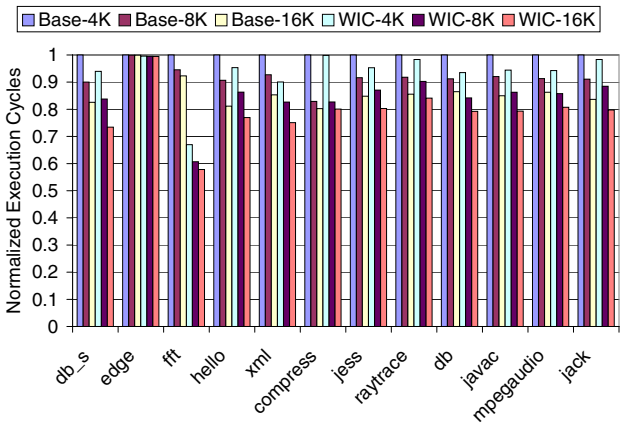

Figure 8: Execution cycles of the base and WIC schemes with the L1 I-cache size varying from 4KB to 8KB and 16KB, which are normalized with the base with a 4K instruction cache.

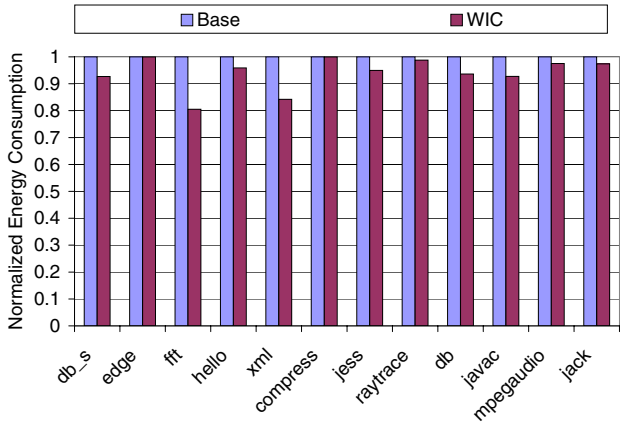

Figure 9: Normalized energy dissipation of the base and WIC schemes, which is normalized with respect to the energy consumption of the base.

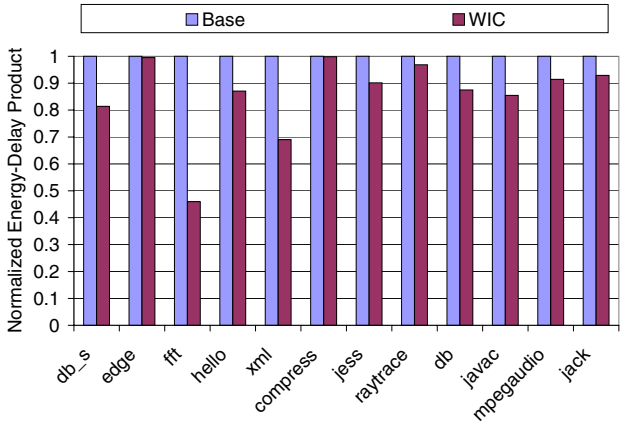

Figure 10: Normalized energy-delay product of the base and WIC schemes, which is normalized with respect to the energy-delay product of the base.

Benchmark	L1-Icache	L1-Dcache	L2 Cache
db_s	3.49%	4.35%	7.54%
edge	0.08%	0.05%	0.17%
fft	10.28%	9.86%	26.09%
hello	2.19%	2.69%	5.03%
xml	8.36%	9.85%	18.96%
compress	0.06%	0.07%	0.25%
jess	2.91%	3.53%	6.30%
raytrace	0.61%	0.76%	1.11%
db	3.45%	4.27%	7.19%
javac	4.18%	4.92%	8.54%
mpegaudio	1.53%	1.47%	4.43%
jack	1.63%	1.57%	3.31%
AVERAGE	3.23%	3.62%	7.41%

Table 3: Energy reduction of the L1 I-cache, L1 D-cache and L2 cache.

those of L1 caches. The reason is that the WIC cache can completely remove the cache synchronization involved with the L2 cache, leading to more significant energy saving.

5. CONCLUSION

This paper explores the hardware-based code cache optimizations for Java applications. We have studied three different schemes to shorten the path of Java binary code generation and fetching, among which the WIC scheme achieves the best performance improvement. The WIC cache allows the Java binary code to be written directly into the instruction cache, which can both reduce the pollution to the L1 data cache and remove the needs of cache synchronization due to the dynamically generated binary code. Our experimental results indicate that the WIC cache improves the performance for a variety of Java applications by 9.6% on average. Also, WIC can reduce the energy dissipation of Java programs by 6% on average.

Acknowledgment

This work was funded in part by the NSF grant CNS 0613633. We would like to thank the anonymous referees for the detailed comments that helped us improve the paper.

6. REFERENCES

[1] M. G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In Proc. of the ACM 1999 Conference on Java Grande, 1999.

[2] K. Hazelwood and M. D. Smith. Generational cache management of code traces in dynamic optimization systems. In 36th Annual International Symposium on Microarchitecture (MICRO), 2003.

[3] K. Hazelwood and J. E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In Proc. of the International Symposium on Code Generation and Optimization (CGO), 2004.

[4] G. Chen, G. Chen, M. Kandemir, N. Vijaykrishnan and M. J. Irwin. Energy-aware code cache management for memory-constrained Java devices. In Proc. of International Systems-On-Chip (SOC) Conference, 2003.

[5] A. Smith. Sequential program prefetching in memory hiearchies. IEEE Computer, 11(2):7-21, 1978.

[6] A. Smith. Cache memories. Computing surveys, 1493):473-530, September 1982.

[7] D. Patterson and J. Hennessy. Computer organization and design: the hardware/software interface. Published by Morgan Kaufmann Publishers, 2005.

[8] N.P. Jouppi. Improving direct-mapped cache performance by the audition of a small fully-associative cache and prefetch buffers. In Proc. of the Annual International Symposium on Computer Architecture, 1990.

[9] A. Aggarwal and S. D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In Proc. of 20th Annual International Symposium on Computer Architecture.

[10] J. L. Baer and T. F. Chen. An effective on-chip pre-loading scheme to reduce data access penalty. In Proc. of the Supercomputing 1991.

[11] C. Gonzalez, A. Aliagas and M. Valero. Data cache with multiple caching strategies tuned to different types of locality. In Proc. of the International Conference on Supercomputing, July 1995.

[12] V. Milutinovic, M. Tomasevic, B. Markovic and M. Tremblay. The split temporal/spatial cache: initial performance analysis. SCIzzL-5, March 1996.

[13] J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporal locality based design. In Proc. of the International Conference on Parallel Processing, August 1996.

[14] F. J. Sanchez, A. Gonzalez and M. Valero. Software management of selective and dual data caches. IEEE TCCA Newsletters, March 1997.

[15] Standard Performance Evaluation Corporation. SPECjvm98 benchmarks. http://www.spec.org/osg/jvm98.

[16] X. Huang, J. E. B. Moss, K. S. Mckinley, S. Blackburn and D. Burger. Dynamic SimpleScalar: simulating Java virtual machines. Technical Report TR-03-03, University of Texas at Austin, Feb. 2003.

[17] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimization. In Proc. of the 27the Annual IEEE/ACM International Symposium on Computer Architecture (ISCA), 2000.

[18] B. Alpern, C. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan and J. Whaley. The Jalapeno virtual machine. In IBM System Journal. 39, 1, pp. 211-238, January 2000.

[19] G. Chen, B. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and R.Chandramouli. Energy-aware compilation and execution in Java-enabled mobile devices. In Proc. of the 17th International Parallel and Distributed Processing Symposium (IPDPS), April 2003.

[20] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, H. Yahyaoui, S. Zhioua and N. Tawbi. E-Bunny: a dynamic compiler for embedded Java Virtual Machines. Journal of Object Technology 4(1): 83-108, 2005.