

Energy-Aware Code Cache Management for Memory-Constrained Java Devices

G. Chen, G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin
CSE Department, The Pennsylvania State University
University Park, PA 16802
{guilchen, gchen, kandemir, vijay, mji}@cse.psu.edu

ABSTRACT

This paper focuses on the influence of memory size limitation on the dynamic translation of Java methods into native code. Specifically, we address the issue of managing a “code cache,” a small memory space allocated for storing the dynamically-generated native code. We show that by adopting a smart bypassing strategy we can enhance the effectiveness of a code cache based system significantly.

1. Introduction

Embedded systems are typically constrained in memory space and energy resources. In this work, we focus on the issue of managing the limited memory space available for supporting energy-aware embedded Java compilation. Due to the proliferation of Java-enabled devices, supporting energy-efficient and memory-conscious embedded Java compilation is important. In contrast to current embedded Java environments that only support interpretation, embedded compilation while being an attractive alternative for Java code execution in terms of performance also imposes additional memory size constraints for storing translated native code. In this work, we focus on managing a small special purpose buffer, called the “code cache,” to store the translated code. The size of the code cache places an upper bound on the memory space that can be used for storing the compiled code, and can be varied to model different degrees of memory constraints (in different architectures). Our objective is to obtain the best energy consumption behavior under a given memory space allocated for storing the native code. Since many Java methods contend for the code cache during their execution, it is important to make the best use of this space from the energy consumption perspective.

The choice of replacement policies and compilation decisions influence the effectiveness of code cache management. Specifically, the replacement policy needs to give a careful consideration to the future invocation patterns as an evicted code when re-invoked in the future will incur the energy cost of re-compilation. This re-compilation cost is a key difference from systems with no code cache size limitations (i.e., in an environment without memory limitation every compiled method can remain in memory as long as needed). In systems

that support both interpretation and compiled execution, selective compilation can reduce the pressure on the code cache without degrading performance. While selective compilation is used in high-performance virtual machine implementations, the goal of those schemes is to predict whether the cost of compilation will be amortized. As a result of the memory constraint imposed on storing translated code in a code cache based system, the traditional means of deciding which methods to compile or interpret needs a fresh look based on re-compilation costs. In particular, it may be beneficial to interpret methods (i.e., “bypassing” the code cache) that would have been translated when no code cache constraints exist.

In this work, we consider different policies for code cache management that select whether to translate or interpret a method and choose the victim method(s) to evict in order to create space for the currently-invoked method. It should be noticed that, in our constrained-memory environment, the decision of whether to compile or interpret a method is tightly coupled with the decision of whether the method can be placed in the code cache (if it is compiled). We compare the energy behavior of the proposed technique with pure interpretation and pure compilation that uses a simple LRU scheme for code cache replacement. The results of our evaluation based on six Java benchmarks from the SPECJVM98 suite reveal that the compilation decisions in constrained memory environments need to be different from traditional dynamic compilation decisions. Particularly, selective use of code cache (hence, compiled execution) can reduce energy significantly over both the pure interpreted and the pure (performance-oriented) compiled solutions.

2. System Architecture and Execution Model

A sketch of the memory hierarchy of our target system is illustrated in Figure 1. In addition to the on-chip data and instruction caches, there is an on-chip memory allocated for storing the translated native code of Java methods called the “code cache.” The address space of this architecture spans both the code cache memory and the off-chip memory. All addresses in the code-cache are accessed directly, while the references mapped to the off-chip memory address space are accessed through the on-chip caches. In other words, the pro-

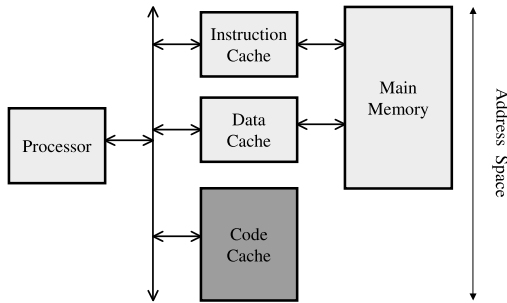


Figure 1: A sketch of our embedded architecture. Note that the total address space is comprised of off-chip main memory and on-chip code cache.

cessor can obtain its instruction stream either from the instruction cache or the code cache.

The Java virtual machine (JVM) that we use in this system can either compile a method into native code, install it in the code cache and then execute this translated native code, or simply interpret the bytecodes of this method found in the data cache hierarchy using the interpreter code in the instruction cache hierarchy. The tradeoffs to consider in this choice are that interpreted execution is typically slower than compiled code execution, and the efficiency of the compiled code needs to amortize the initial compilation cost before it becomes preferred. Traditional decisions to select between compilation and interpretation consider only this amortization tradeoff and hence require only the invocation count of this method. However, in our architecture, the choice of compilation influences the code buffer management. As a result, an energy-efficient compilation decision is inextricably linked to the invocation pattern of other Java methods that are competing for the same code cache space.

There have been several recent efforts at designing Just-in-Time (JIT) compilers that address the issue of memory space requirements of the compiler code and that of the intermediate code generation in (e.g., [6, 1]). While previous work studied several code cache management strategies for applications written in other languages, to our knowledge, this is the first study that attacks the problem in a Java environment. While the work in [5] requires that the entire application to be compiled before execution, our approach allows mixed-mode execution (i.e., compilation/interpretation).

3. Our Approach

As discussed earlier, effective management of code cache is of utmost importance, as failing to do so can result in large performance and energy loss. When JVM finds that compiling a method will bring benefits, it takes the following steps: (1) It determines the length of the native code for this method. If this method has never been compiled before, we estimate

the length of its native code by counting the number of its bytecodes of each type. It is known that a naive compiler compiles bytecodes to native code by simply mapping each bytecode into a native code fragment. For the bytecodes of the same type, the lengths of the native code are the same. Therefore, by counting the number of the bytecodes of each type and by multiplying this number with the length of the native code sequence for this type, we can estimate the size of native code for the given method. Such bytecode counting is performed in the verification phase when Java class is loaded. It should be noted that the actual length of the native code may vary due to compiler optimizations. However, since the memory is constrained, we do not perform any optimizations that may cause the code length to expand. (2) It allocates memory from the code cache. If the code cache has no free blocks that are large enough, some “less important” methods will be evicted to make room for the current method. (3) It compiles the method and puts the compiled code in memory space that has been allocated from the code cache.

Since the code cache is a shared resource (that can potentially be utilized by all methods), in allocating its space for methods, one should consider the relative importance (criticality) of different methods with respect to each other. In particular, if one method is more important than the other, it should be given priority in using the code cache. One might adopt several strategies in determining the importance of a method such as recency of use, frequency of use, compilation cost, and so on. In this paper, we adopt a strategy that combines recency of use with frequency of use. More specifically, if a method is used frequently enough, it gets its chance to be stored in the code cache. Conversely, if a method not used very frequently is invoked, it is interpreted (instead of being compiled and placed into the code cache). In other words, we “bypass” the code cache for the methods not invoked very frequently. In addition, once a method is stored in the code cache, it stays there as long as it is used frequently enough in the recent past. It should be noted that, if a method is invoked, this method should not be evicted before it returns.

To illustrate why such a bypassing-based code cache management strategy might be preferable over the default (pure LRU based) strategy, let us consider the example scenario depicted in Figure 2. The top portion of this figure shows an access pattern where four methods (m1, m2, m3 and m4) are invoked (the order is from left to right). In the middle part of the figure, we give the activity performed by the default compilation strategy (LRU-based) that always compiles and our bypassing-based strategy. Here, I, C, E refer to interpretation, compilation, and execution, respectively, and it is assumed that the code cache can keep only one method at a time. C, E means compilation followed by execution of the native code. In the default strategy, all methods are compiled before execution. In a bypass-based strategy, a method is compiled and placed in the code cache after the method has been executed a certain number of times (on second execution in our example),

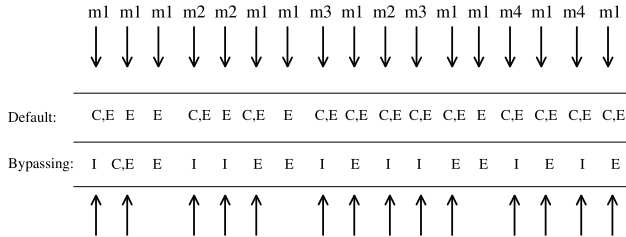


Figure 2: An example scenario showing the difference between bypassing and the default execution strategy. The top portion shows the method invoked (from left to right). The bottom portion indicates the cases where our approach differs from the default strategy. For this example, our approach performs eleven less compilation (or re-compilations) as compared to the default strategy.

Strategy	Threshold?	Threshold Reset?	Weight Reduction	Victim Selection
Default	No	N/A	N/A	LRU
Strategy 1	Yes	No	N/A	LRU
Strategy 2	Yes	Yes	Local	LRU + Weight Comp.
Strategy 3	Yes	Yes	Global	LRU + Weight Comp.
Strategy 4	Yes	Yes	Global	Min Weight + Weight Comp.
Strategy 5	N/A	N/A	N/A	N/A

Table 1: Summary of the code cache management strategies evaluated in this work. Default always compiles whereas Strategy 5 always interprets. Strategy 1 is the normal performance-oriented strategy, whereas Strategies 2, 3, and 4 are our strategies.

provided that it can find space in the code cache by evicting existing methods. A method is evicted from the cache only if its number of executions (up to the current execution point) — also called its “weight”— is smaller than that of the currently activated method. Consequently, for the method invocation scenario depicted in Figure 2, our approach places method m1 in the code cache, and keeps it there throughout the execution. As a result, the invocations of the other three methods result in interpretation. The bottom part of this figure shows the cases (execution points) where these two strategies take different actions. As one can see, the default strategy performs eleven additional compilations as compared to the bypassing strategy. The bypassing strategy avoids additional compilation by choosing to interpret rather than compile and execute for some of the methods. As one can expect the energy consumed in compilation and native execution to be larger than just pure interpretation (in fact, this is the reason why that some adaptive JIT compilers do not activate compilation until the method is invoked a certain number of times), we anticipate a bypassing based scheme to be more energy-efficient.

In this paper, we explore the strategies summarized in Table 1. The **Threshold?** column indicates whether a method should be executed a certain number of times before it can be compiled and placed into the code cache. The **Threshold Reset?** column tells whether we apply threshold each time the method is evicted from the code cache. For our weight-based

strategies, we assign an initial weight (also called the “installation weight”) whenever a compiled method is placed into the code cache. The installation weight prevents a method from being evicted immediately after being brought into the code cache (i.e., it helps to amortize the compilation cost before being evicted). In our weight-based strategies, each time a method is invoked, its weight is increased by one. Also, the weights of some or all methods in the code cache (except the invoked one if it is already in the cache) are decreased to a certain percentage of their original value. The **Weight Reduction** column in Table 1 indicates how the weights are reduced. The local policy reduces only the weight of the method that is selected by the replacement policy. The global policy, on the other hand, reduces the weights of all the methods in the code cache. The **Victim Selection** column indicates how the victim method is selected. The default scheme and Strategy 1 use pure LRU. In Strategies 2 and 3, the victim method is selected by LRU; however, its weight (after reduction) is compared to that of the invoked method. The victim is replaced only if its weight is smaller than that of the invoked method. In Strategy 4, the victim is the one with the minimum weight (among all the code cache residents). Its weight is compared to the weight of the invoked method, and it is replaced only if its weight is smaller. In our strategies, if evicting the victim method does not provide sufficient space for the invoked method, we select an additional victim (using the same victim selection policy), and use the “combined weight” of the victims in the weight comparison. Updating the weights of the methods in the cache is important to capture the variance in method locality.

4. Experimental Setup and Evaluation

4.1 Experimental Setup

In order to evaluate the proposed strategies, we used six applications from the SPECJVM98 suite [7] (we had difficulty in executing db under our optimization; so, we excluded it). The important characteristics of these applications are provided in Table 2. The second and third columns provide, respectively, the static and dynamic counts of method invocations. The last column provides the memory size required if all methods are compiled. We can see from this last column that, if, for example, we have an 8K code cache, we can accommodate only compress in its entirety. Therefore, effective management of the code cache space is of utmost importance.

To obtain detailed energy profiles, we have customized an energy simulator and analyzer using the Shade (SPARC instruction set simulator) tool-set [3], and simulated LaTTe JVM [10] executing a Java code. Our energy simulator tracks the accesses to the different caches and off-chip memory and obtains the overall energy consumption by multiplying the number of accesses with the per access energy costs for the different caches. The energy models from SimplePower energy simulator [8] are employed in this work. The accuracy of the memory energy models employed in the simulator is within

Benchmark Name	Number of Methods	Number of Method Invocations	Size of Compiled Code (bytes)
compress	21	18155643	6328
jess	269	5406843	57120
javac	569	1459765	173620
mpegaudio	147	9255658	53556
mtrt	126	20118945	40256
jack	215	677318	112720

Table 2: Benchmarks used in our experiments and their important characteristics. The last column provides the memory size required if all methods are compiled. We see that an 8K code cache can only accommodate `compress` in its entirety. Therefore, effective management of the code cache space is of utmost importance.

2.4% of actual values. The default instruction cache (data cache) used in our experiments is 8K (16K), 2-way (2-way), with a block size of 32 (32) bytes. In the simulated architecture, the Load, Store, Branch, ALU (Simple), ALU (Complex), and NOP instructions take, respectively, 4.81nJ, 4.48nJ, 2.87nJ, 2.85nJ, 3.73nJ, and 2.64nJ. The energy consumption values for load/store operations include the energy spent in cache accesses. We have also customized the simulator infrastructure to identify references from the JVM to breakdown the accesses as occurring during interpretation (I), compilation (C), or compiled execution (E). The default parameters used in all experiments (unless mentioned otherwise) are a code cache of 8K, a compilation threshold of 10 (invocations), an installation weight of 10, and a weight update policy that reduces the original weight by 25%.

4.2 Results

Figure 3 shows the energy comparison of the different strategies with respect to the LRU-compile approach (i.e., the default execution strategy). It can be observed that even a pure interpreter (Strategy 5) is better than the LRU-compile in three out of our six applications. In these cases, the memory space limitation of the code cache prevents the compiled code from staying in the cache long enough for amortizing the compilation cost. Hence, the cost of repeated compilation dominates the LRU scheme and constitutes more than 80% of overall energy consumption. However, for `compress`, `jess` and `mtrt`, the interpreter’s performance is much poor as compared to the pure LRU-based compilation strategy. This is because these applications benefit from repeated execution of some methods that amortize the cost of compilation in spite of the code cache constraints. In `compress`, this happens because the execution of two methods dominates the application. Since the compiled codes of these two methods fit in the code cache, LRU-compile performs much better than interpreted execution.

Next, we observe that Strategy 1 is not effective in our constrained memory environment as it only bypasses the compilation of methods that cannot amortize the cost of compilation. Since it has no consideration for the duration for which the compiled code can be kept in the code cache, the number of recompilations in this approach is similar to that of LRU-

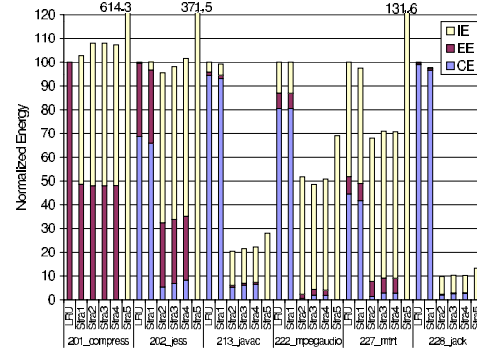


Figure 3: Normalized energy consumption with the base parameters. IE, EE, and CE correspond to, respectively, interpretation energy, execution energy, and compilation energy. We see that the average energy improvements provided by Strategies 1, 2, 3, 4, and 5 are 0.44%, 41%, 40%, 40%, and -132%, respectively.

compile. Due to the better management of the code cache, the proposed Strategies 2, 3, and 4 consume only 59%, 60% and 61% (averaged across all benchmarks) of the energy expended by the LRU-compile strategy. Further, they also outperform the interpreted version in all six benchmarks. Specifically, Strategies 2, 3, and 4 consume only 49%, 50% and 51% (averaged across all benchmarks) of the energy consumed by the interpreted approach.

5. Conclusions

This work focuses on memory constrained Java environments and proposes an optimization technique that targets a code cache-based system. Our results indicate that proper management of the code cache is critical to the energy-efficiency of dynamic compilation.

6. REFERENCES

- [1] M. Chen and K. Olukoton. Targeting Dynamic Compilation for Embedded Environments. Proceedings of the 2nd USENIX Java Virtual Machine Research and Technology Symposium, August 2002.
- [2] CLDC and the K Virtual Machine (KVM). <http://java.sun.com/products/cldc/>.
- [3] B. Cmelik and D. Keppel. Shade: A Fast Instruction Set Simulator for Execution Profiling. Proceedings of ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, pp. 128–137, May 1994.
- [4] K. Farkas, J. Flinn, G. Back, D. Grunwald, and J. Anderson. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. Proceedings of SIGMETRICS’00, Santa Clara, CA, June 2000.
- [5] K. Hazelwood and M. D. Smith. Code Cache Management Schemes for Dynamic Optimizers. Proceedings of the Workshop on Interaction between Compilers and Computer Architecture (Interact-6), Boston, MA, February, 2002.
- [6] N. Shaylor. A Just-in-Time Compiler for Memory Constrained Low-Power Devices. Proceedings of the 2nd USENIX Java Virtual Machine Research and Technology Symposium, August 2002.
- [7] SPECJVM98 Benchmarks. <http://www.specbench.org/osg/jvm98/>.
- [8] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower. Proceedings of the International Symposium on Computer Architecture, June 2000.
- [9] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy Behavior of Java Applications from the Memory Perspective. Proceedings of the 1st USENIX Java Virtual Machine Research and Technology Symposium, Monterey, California, April, 2001.
- [10] B. S. Yang, S. M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioğlu, and E. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pp. 128–138, October 1999.