One Method At A Time Is Quite A Waste Of Time

Andreas Gal, Michael Bebenita, and Michael Franz

Computer Science Department
University of California, Irvine
Irvine, CA, 92697, USA
`{gal,bebenita,franz}@uci.edu`

Abstract. Most just-in-time compilers for object-oriented languages operate at the granularity of methods. Unfortunately, even "hot" methods often contain "cold" code paths. As a consequence, just-in-time compilers waste time compiling code that will be executed only rarely, if at all. We discuss an alternative approach in which only truly "hot" code is ever compiled.

1 Introduction

Many modern object-oriented languages such as Smalltalk [7], Java [10, 1] and C# [8] have a virtual machine-based execution model. Just-in-time compilation is often used to translate the virtual machine bytecode into native machine code for faster execution.

Just-in-time compilers used in virtual machines are often quite similar in structure to their static counterparts. In case of static compilation, the compiler processes the program code method by method, constructing a control-flow graph (CFG) for each method, and performing a series of optimization steps based on this graph. In the final step the compiler traverses the CFG and emits native code.

Most dynamic compilers behave essentially identically: pick a method, construct its CFG, and generate native code for it. In order to strike a balance between startup latency and long term efficiency, JIT compilers often operate in a mixed mode environment instead of compiling the entire program. In the case of Java, bytecode is first executed through an interpreter. Methods that are invoked often are identified as "hott" and are dynamically compiled into native code.

In a static compiler, using methods as compilation units is a natural choice. In static compilation there is usually no profiling information available that could reveal whether any particular part of a method is "hotter" and thus more "compilation worthy" than others, it actually makes perfect sense to always compile entire methods and all possible paths through them. After all, for a static compiler, all these different paths look equally likely to be taken at runtime.

This is dramatically different in case of a dynamic compiler. In contrast to its static counterpart, a dynamic compiler has access to runtime profile information that the virtual machine can collect easily while it interprets code. With this profile information, the dynamic compiler can decide which parts of a method actually contribute to the overall runtime, and which parts are rarely taken and are in fact irrelevant from a global perspective as far as optimization potential is concerned.

Region-based compilation was proposed to address this issue. Suganuma et al. [13], proposed a region-based just-in-time compiler for Java. It uses runtime profiling to select code regions for compilation and uses partial method inlining to inline profitable parts of method bodies only. The authors observed not only a reduction in compilation time, but also achieved better code quality due to rarely executed code being excluded from analysis and optimization.

However, region-based compilation really only addresses the symptoms by trying to exclude unprofitable code areas from compiled methods, instead of addressing the actual problem, which is the choice of an unsuitable compilation unit in the first place.

The unsuitability of methods as compilation units becomes even more apparent when trying to deal with long running hot code regions inside a method that is currently being interpreted. Once such a method is discovered to be hot, it is subsequently compiled and optimized to directly executable native code. The runtime system then has to perform an expensive and complex process called *on-stack replacement* [9] to substitute the newly compiled code for the interpreted version. For a virtual machine, being able to deal with on-stack replacement means having to support side-exits from running interpreted methods, and side re-entries into compiled method bodies. Both of these processes are so complex that very few open-source or research virtual machines actually support on-stack replacement. Therefore it is a feature found mostly only in commercial VMs and large-scale research efforts such as Jikes RVM [2, 4].

We have been exploring a different approach to building dynamic compilers in which no CFG is ever constructed, and no source code level compilation units such as methods are used. Instead, we use runtime profiling to detect frequently executed cyclic code paths in the program. Our compiler then records and generates code from dynamically recorded *code traces* along these paths. It assembles these traces dynamically into a tree-like data-structure that covers frequently executed (and thus compilation worthy) code paths through hot code regions. [5, 6]

Our tree-based code representation has a number of advantages. On the one hand, it subsumes and greatly simplifies on-stack replacement. In our scheme, compiled code is always entered independently of method boundaries. Thus replacing interpreted code with the compiled equivalent becomes trivial. When a trace has been recorded, the interpreter stops at the loop header (which is the entry point of the associated trace tree) and the entire tree is recompiled. The trace tree can immediately be executed, replacing the previously compiled copy. Since recording is only performed after the native code has side exited into the interpreter, its not necessary to actually implement an on-stack replacement mechanism that translates from one compiled state into another.

The other major benefit of our approach is that our trace tree data structure only contains actually relevant code areas. Edges that are not executed at runtime (but appear in the static CFG) are not considered in our representation, and are delegated to the interpreter in the rare cases they are taken. Unlike compilers that use basic-block based CFG analysis where advanced optimizations are expensive, our tree-based compiler can perform advanced optimizations quickly. The lack of control flow merge points in our tree-based representation simplifies optimization algorithms.

2 Trace Compilation

Our trace-based JIT compiler targets the JVM bytecode language and starts the execution of class files through an interpreter, much like traditional JIT compilers do. To detect "hot" code areas that are suitable candidates for dynamic compilation, we use a simple heuristic first introduced by Bala et al. [3]. The interpreter is augmented to keep track of frequently executed backwards branches. The targets of such branches are often the loop headers of hot code areas. Once such a loop header is discovered, we attempt to record a trace through the loop region associated with the header. Starting at the loop header, the interpreter records subsequent instructions until the original loop header is reached again.

During the recording phase, branch instructions are recorded as *guard* instructions and indicate possible loop exits that must be safeguarded against during native code execution. Our JIT compiler emits appropriate code to check that a previously observed result of a guard condition still applies during future executions of the compiled trace. If the condition fails, a side exit is performed and the compiled code hands control back to the interpreter and resumes it in a state consistent with the side exit detected during the native code execution.

During trace recording, method invocations are inlined into the trace. In case of a static method invocation, the target method is fixed and no additional runtime checks are required. For dynamically dispatched methods, the trace recorder inserts a guard instruction to ensure that the same actual method implementation is reached as was found during trace recording. If the guard fails, a regular dynamic dispatch is repeated in the interpreter. If it succeeds, we have effectively performed method specialization on a predicted receiver type. Since our compiler can handle multiple alternative paths through a trace, eventually our compiler specializes method invocations for all commonly occurring receivers.

Guard instructions that appear in inlined methods must carry enough information to be able to reconstruct the interpreter's state in case a side exit occurs. In case of a side exit in the same method scope that the trace was entered, all that has to be done is to write back any values held in machine registers into the appropriate stack and local variable locations. In case of a "deep" side exit inside a method invocation that is being inlined, guard instructions must store enough information to allow the virtual machine to fully construct the interpreter state, which includes constructing method frames on the virtual machine stack.

During native code execution, if a guard instruction fails and a side exit occurs, our JIT compiler must resume execution through the interpreter. Since this switch can be expensive, our JIT compiler attempts to include traces that splinter off at side exits into the original trace. For this, at every side exit we resume interpretation, but at the same time also re-start the trace recorder to record instructions starting at the side exit point. These secondary traces are completed when the interpreter revisits the original loop header. This results in a tree of traces, spanning all observed paths through the original program's loop.

Figure 1 shows the control flow graph of a nested loop, and the corresponding trace tree that is recorded once instruction 2 is detected as a loop header. From left to right, straight line tree paths represent the traces that were recorded. The first trace to be
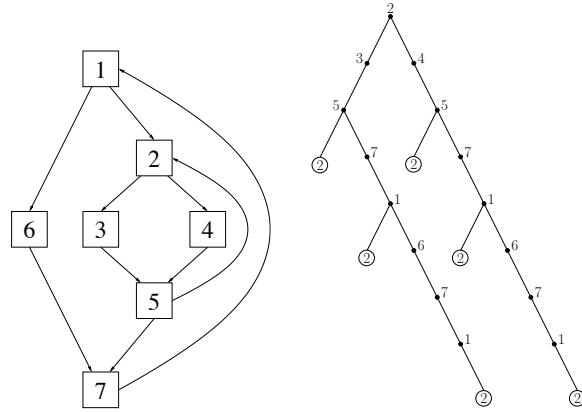
Fig. 1. Example of a nested loop and a set of suitable traces through the nested loop. Each basic block in the control-flow graph of the loop (left) is represented as a numbered node. (1) is the loop header of the outer loop, (2) the loop header of the inner loop. Since (2) is executed more frequently than (1), it will become the anchor node of the trace tree (right) that is recorded for this loop. In the trace tree every trace starts at the anchor node and also terminates at it. Since (2) is the anchor node of the trace tree, every trace ends with node (2). The loop header of the outer loop is inlined into the trace tree as just another trace originating and terminating at node (2).

recorded is $\{2, 3, 5, 2\}$ followed by $\{2, 3, 5, 7, 1, 6, 7, 1, 2\}$ and then $\{2, 3, 5, 7, 1, 2\}$, etc. After every secondary trace that is recorded and added to the trace tree, our JIT compiler recompiles the entire trace tree and emits new native code for it. At the next entry of the compiled code through the loop header, this new version is executed until another side exit is encountered, in which case the tree can be extended further.

The first recorded trace is usually the hottest path through the loop. Secondary traces are less likely to occur during execution and thus benefit less from compilation. Our compiler limits the growth of trace trees to an experimentally chosen number of traces.

3 Evaluation

We have built a dynamic compiler for Java based on the described trace tree compilation method. The compiler was implemented as part of the JamVM [11] Java Virtual Machine, which is a fast Java interpreter but previously lacked a just-in-time compiler.

To detect trace tree entry points we modified JamVM's interpreter to keep track of the execution frequency of JVML branch instructions. Once a certain threshold is surpassed the trace recorder will attempt to record a primary trace starting at the destination of such frequently executed branch instructions.

After compiling the primary trace, the trace is executed in compiled form. The trace recorder is started whenever a side exit occurs, attempting to grow the associated trace tree.

Our current compiler prototype only performs a limited set of optimizations including constant propagation, copy propagation, arithmetic optimizations, loop invariant

code motion and array bounds check elimination. The optimized intermediate representation of the trace tree is then converted to PowerPC machine code.

With the current set of optimizations and our rather simplistic PowerPC code generator that does not perform instruction scheduling, we are able to speed up the execution of section 2 of the Java Grande benchmark [12] set by factor 5 to 12.

On average, our system generates approximately 1500 bytes of machine code for each benchmark program in section 2, which is significantly smaller than the code emitted by method-based just-in-time compilers such as Sun's Hotspot JVM [14]. The latter generates approximately 30kB of code for each Section 2 benchmark program, but also achieves an additional speedup of 1.5 to 2 over the performance of our prototype compiler.

For most benchmarks in section 2 the recorded trees achieve near-ideal coverage of the performance critical part of the loop with 2-4 traces. This means that adding additional traces will no longer produce any substantial speedup since all frequently executed paths are covered. The largest trace tree is produced for *HeapSort*. The second and third trace added to the tree produce a speedup of 2.5 over the initial trace by reducing side exit frequency. The fourth trace produces another speedup of 2. Adding additional traces beyond this point will not produce any visible speedup.

Also noteworthy about our prototype compiler is the compilation speed. Our system spends 100 to 750 times less time in the just-in-time compiler than Sun's Hotspot JVM. While in part this can be explained with the fact that we compile less code in general, this would only explain a compilation time speedup of approximately factor 20. Thus, the overall compilation performance highlights that our trace-based compilation approach is not only more selective when deciding what code to compile, but is also less time consuming per instruction compiled.

4   Discussion

Traditional dynamic compilers are often built based on the same principles and ideas that were invented for their static counterparts. We introduced a novel intermediate representation which we call trace trees that eliminates the inherent weaknesses of using methods as compilation units in a dynamic compiler. The trace tree data structure introduced in this paper enables our just-in-time compiler to incrementally discover alternative paths through a loop and then optimize the loop as a whole, regardless of a possible partial overlap between some of the paths.

When programs execute, the dynamic view of basic blocks and control flow edges that one encounters can be quite different from the static control flow graph. Our trace-tree representation captures this difference and provides a representation that solely addresses "hot" code areas and "hot" edges between them. All other basic blocks and instructions never become part of our compiler's intermediate representation and therefore do not create a cost for the compiler.

Using trace trees as compilation units instead of source code constructs such as methods also provides a intuitive boundary between "cold" code that is interpreted and "hot" code that needs to be compiled, and defines a simple and efficient method to transition between such areas (trace entry and trace side exits).

5 Acknowledgement

References

1. K. Arnold and J. Gosling. *The Java Programming Language (2nd ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
2. M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.
3. V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. *Hewlett Packard Laboratories Technical Report HPL-1999-78. June 1999*, 1999.
4. M. Burke, J. Whaley, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, and H. Srinivasan. The Jalapeño dynamic optimizing compiler for Java. *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, 1999.
5. A. Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, September 2006.
6. A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 144–153, New York, NY, USA, 2006. ACM Press.
7. A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.
8. A. Hejlsberg, P. Golde, and S. Wiltamuth. *The C# Programming Language*. Addison-Wesley Professional, 2003.
9. U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deopti-mization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, New York, NY, USA, 1992. ACM Press.
10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
11. R. Lougher. JamVM virtual machine 1.4.3. http://jamvm.sf.net/, May 2006.
12. J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *Proceedings of the ACM 1999 Java Grande Conference, San Francisco*, 1999.
13. T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):134–174, 2006.
14. Sun Microsystems. The Java Hotspot Virtual Machine v1.4.1, Sept. 2002.