Trace Based Compilation in Interpreter-less Execution Environments

Michael Bebenita, Mason Chang, Karthik Manivannan, Gregor Wagner,
Marcelo Cintra, Bernd Mathiske, Andreas Gal, Christian Wimmer, Michael Franz

University of California, Irvine

Trace based compilation is a technique used in managed language runtimes to detect and compile frequently executed program paths. The goal is reduced compilation time and improved code quality since only "hot" parts of methods are ever compiled. Trace compilation is well suited for interpreter based runtime environments because the control flow of an application program is highly visible and recordable. In this technical report, we show that trace compilation is also feasible and beneficial in runtime environments without interpreters. We present the implementation of a trace based Java compiler for the meta-circular Maxine Virtual Machine from Sun Microsystems. Maxine uses a tiered compilation strategy. Methods are first compiled with a non optimizing JIT compiler in order to collect profiling information, and then recompiled with an optimizing compiler for long term execution. We record traces via dynamic insertion and removal of instrumentation code in non optimized methods. We then optimize and link recorded traces using the already existing optimizing compiler.

1. INTRODUCTION

Many modern object-oriented languages such as Smalltalk [Goldberg and Robson 1983], Java [Gosling et al. 2005] and C# [Hejlsberg et al. 2003] 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 performs 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 runtime systems, bytecode is first executed through an interpreter. Methods that are invoked often are identified as "hot" 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. In contrast to its static counterpart, a dynamic compiler has access to runtime profile information that the virtual ma-

chine 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.

Suganuma et al. [2006], proposed a region-based just-in-time compiler for Java to address this issue. 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.

Gal [2006] proposed an 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, they use runtime profiling to detect frequently executed cyclic code paths in the program. The 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. A major benefit of this approach is that the 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 the trace representation, and are delegated to an interpreter in the rare cases they are taken. The absence of control flow merge points in this tree-based representation greatly simplifies optimization algorithms and this results in optimization passes being quicker than compilers that use traditional CFG based analysis.

In this work, we have built a trace based compiler for the meta-circular Maxine Virtual Machine from Sun Microsystems. The Maxine VM does not have an interpreter and it uses a template-based lightweight JIT as the baseline compiler. The execution performance of this sort of JIT compiler is much faster than that of a traditional interpreter. When building a trace compiler for such an execution environment it is very important to keep the tracing overhead low, since the execution cost for the baseline JIT is very low. Tracing overhead becomes more acceptable in the case of an interpreter based execution environment where the baseline execution cost, i.e. interpretation cost, is high. This paper makes the following contributions:

—Identification and recording of trees of frequently executed code traces using minimally invasive dynamic instrumentation.

—An efficient calling convention and side exit mechanism for trace trees.

The remainder of this paper is organized as follows - Section 2 is a general overview of trace tree based compilation which we use to capture and compile frequently executed code regions. In Section 3 we describe our trace recording, compilation and execution techniques by explaining our implementation of trace compilation for the Maxine VM. In Section 4 we evaluate our trace based compiler on a set of benchmarks. Related work is discussed in Section 5. The paper ends with conclusions in Section 7 and an outlook on future work.

2. TRACE COMPILATION

Trace based compilation uses a collection of code paths through long running loops, as the unit of JIT compilation. Frequently executed code paths are identified by

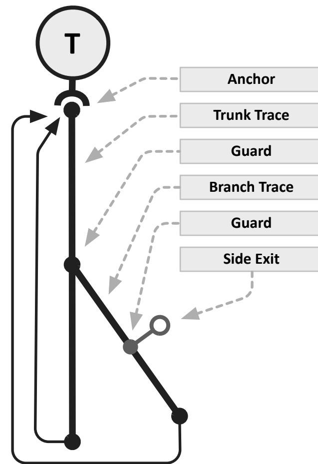

Fig. 1. A tree with two traces, a trunk trace and one branch trace. The trunk trace contains a guard to which a branch trace was attached. The branch trace contains a guard that may fail and trigger a side exit. Both the trunk and the branch trace loop back to the tree anchor, which is the beginning of the trace tree.

using runtime profiling. The linear nature of code paths, and the ability to capturing such paths through hot code, allows the trace compiler to easily perform optimizations that are highly effective.

2.1 Structure of a Trace Tree

A *trace* is a linear sequence of successive instructions observed during the execution of a program loop. A trace is linear in the sense that it covers only one program path within a loop i.e the path that is observed when the trace is recorded. A *trace tree* is a grouping of connected traces, covering multiple program paths inside a loop. Figure 1 shows the various components of a trace tree. A *trace anchor* is a point where a trace begins. The first trace that is recorded in a trace tree is called the *trunk trace* and its trace anchor is the *tree anchor*. A tree anchor is usually the beginning of a loop and all the traces in a trace tree loop back to it. A trace that gets attached to the trunk trace is called a *branch trace*. Branch traces capture control flow that is not observed during the recording of the trunk trace. Any instruction on a trace that can cause control flow to diverge from the trace is guarded using *guard* instructions. For instance, if the conditional branch opcode *if_icmpge* (branch greater than) instruction is observed to be taken during the recording of a trace, then a guard instruction, *guard-greater-than* is inserted in the recorded trace to ensure that the branch is always taken during the execution of the trace tree. When a guard fails, a *side exit* is said to have occurred, and execution is returned to the code generated by the baseline JIT. Since the JITed code uses the JVML stack based execution model, appropriate stack restoration has to be performed before switching to JITed code.

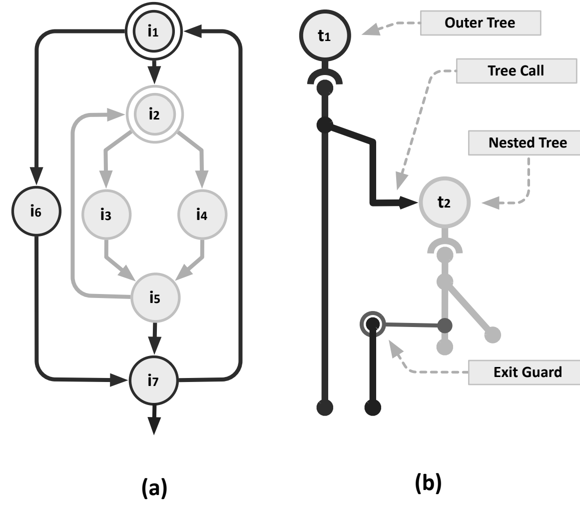

Fig. 2. (a) Control flow graph of a nested loop with a control flow branch in the inner loop. (b) Nested trace trees - an inner tree captures the inner loop, and is nested inside an outer tree which "calls" the inner tree. The inner tree returns to the outer tree once it exits along its loop condition guard .

2.2 Discovering and Compiling Trace Trees

Traces are identified by inserting counters at the targets of backward branches. These counters are incremented whenever control flow reaches them, and when a counter reaches a certain threshold, it is considered to be a trace anchor, and trace recording is started. The recorder stores the traced bytecodes in a TSSA [Gal 2006] based intermediate representation (IR). Guards are inserted at points of potential control flow divergence. If the trace being recorded reaches back to the trace anchor, then trace recording is stopped and the trace is compiled into machine code. This compiled code is executed whenever program execution subsequently reaches this trace anchor.

Recording traces without limiting their scope to a single loop could lead to excessive tail duplication. For example, a trace tree recorded for Figure 2(a) could include the traces $\{i_2, i_3, i_5, i_7, i_1, i_6, i_7, i_1, \delta\}$ and $\{i_2, i_4, i_5, i_7, i_1, i_6, i_7, i_1, \delta\}$, where δ indicates a loop back to the tree anchor. The problem here is that several iterations of the outer loop have become inlined into traces recorded for the inner loop. Gal et al. [Gal et al. 2009] propose Nested Trace Trees as a solution for this problem. In this approach, trace scope is limited to a single loop and trace trees for an outer loop could call a trace tree built for its inner loop. Figure 2(b) shows the control flow graph, for Figure 2(a), captured using the nested trace trees. Tree t_2 captures the traces $\{i_2, i_4, i_5, \delta\}$ and $\{i_2, i_3, i_5, \delta\}$ for the inner loop and tree t_1 captures the traces $\{i_1, i_6, i_7, \delta\}$ and $\{i_1, callt_2, i_7\}$ for the outer loop. Nested trace trees also make it possible to efficiently capture traces, for otherwise impossible cases, like the loops shown in Figure 3

Fig. 3. (a) Control flow graph of a loop with two nested loops that are nested one level deep (left) and corresponding nested trace tree representation (right). The outer tree calls the two inner nested trace trees and places guards at their side exit locations.

3. IMPLEMENTATION

This section describes our trace recording, compilation and execution techniques by explaining our implementation of trace compilation for the Maxine VM.

3.1 Maxine VM

Maxine is a meta-circular Java Virtual Machine developed at Sun Microsystems. The VM is mostly written in Java and this allows it to leverage the benefits of meta-circularity, highly componentized design and reduced prototyping time. The Maxine VM does not include a bytecode interpreter and instead has a low overhead template-based baseline JIT. This JIT generates code just-in-time by looking up a translation table that maps each JVML bytecode to a template or sequence of machine instructions. The lightweight JIT compiler concatenates these sequences together to compile a method. It's important to note that JVML stack semantics are preserved by this JIT. This just-in-time compilation method is also sometimes referred to as inline threading. Maxine also has a Continuation Passing Style(CPS) based optimizing compiler which has higher compilation overhead than the lightweight JIT but produces better target code. The Maxine RVM can operates in a mixed mode environment where baseline JITed methods that exceed a certain invocation count are recompiled with the optimizing compiler.

3.2 Trace Recording

The main prerequisite for building a trace compiler is the capability to record arbitrary program traces. Building a trace recorder is fairly easy in an interpreter based execution environment.

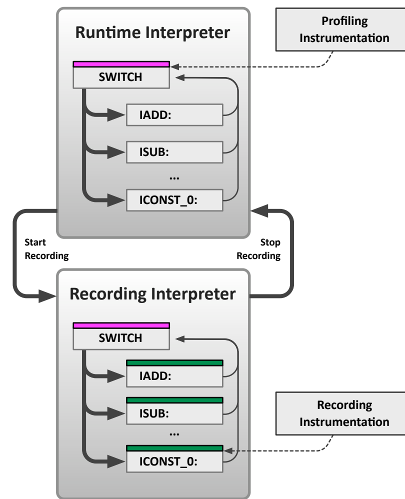

Fig. 4. Design of a simple interpreter based trace recorder. In order to minimize instrumentation overhead, two interpreters are used. A runtime interpreter is used to profile and execute instructions, while a recording interpreter is used to record instructions. The virtual machine can switch between the two interpreters at any time.

3.2.1 *Interpreter Based Trace Recording.* Instrumentation code can be inserted in the interpreter loop before or after each opcode handler. This instrumentation can record the execution of each opcode as well as inspect results. Unfortunately, instrumentation overhead impacts the general interpreter performance. Since trace recording is relatively infrequent,VM a two interpreter trace recording system can used to minimize instrumentation overhead. One runtime interpreter is responsible for executing opcodes while a functionally equivalent recording interpreter is used to record traces. Once the runtime interpreter decides to record a trace, it can switch to the recording interpreter. Similarly, once trace recording is complete, the recording interpreter can switch back to the runtime interpreter.

3.2.2 *Lightweight JIT Based Trace Recording in Maxine VM.* Since the lightweight JIT performance is much higher than that of an interpreter, it becomes more difficult to pay back the overhead introduced by trace recording using trace compilation. In other words, the relative benefit of trace compilation over the baseline is inversely proportional to performance of the host execution environment. The more efficient the host environment is, the more important it becomes to reduce tracing instrumentation overhead.

In order to reduce tracing overhead we use a technique similar to the one used in the dual interpreter approach. We maintain two compiled versions for each method that is subject to trace recording, a *profile instrumented* version and a *trace instrumented* version. The first version is sprinkled with profiling instrumentation or *anchors* at select program locations. These locations are generally loop headers and are discovered using an inexpensive static analysis that utilizes the JVM bytecode verifier. During program execution these anchors profile program behavior and trig-

ger the recording and compilation of program traces. Trace recording is performed by replacing the profile instrumented version of the method with the trace instrumented version. This second version contains tracing instrumentation at every bytecode location that signals the execution of the bytecode. Once trace recording is complete, execution is resumed in the profile instrumented version. Switching between the two versions is possible because they are semantically equivalent and share the same frame layout, namely the JVML stack frame layout.

3.2.3 *Tracer.* At runtime, execution is guided by the *Tracer.* The Tracer is a thread local runtime component that receives messages from instrumented methods and triggers the recording and execution of traces. Instrumented methods interact with the tracer by sending two types of messages `visitAnchor` (a_i) and `visitBytecode` (b_j). The tracer in turn responds with a resume address where the instrumented code should resume. If the resume address is *zero*, the execution falls through to the next bytecode instruction.

The pseudo code presented below is the instrumentation code inserted at each anchor and bytecode location. The `visitAnchor` message tells the Tracer which anchor is about to be executed and what the current frame pointer is. The Tracer uses this information to profile the anchor's execution behavior and trigger trace recording. If the Tracer wants to start recording, it replies to the `visitAnchor` message with the instruction address of the first bytecode to be recorded in the trace instrumented version of the method. Effectively, execution is transfered from the profiled version of the method to the trace instrumented version. If the Tracer wants to keep profiling the anchor, it replies with the *zero* address.

```
resumeAddr = visitAnchor(anchor, RBP);
if (resumeAddr != 0) {
    jump(resumeAddr);
}
```

The `visitBytecode` message tells the Tracer which bytecode is about to be executed. It does this by passing along the current instruction pointer, as well as the stack pointer and the frame pointer. The Tracer can infer bytecode level information from the instruction pointer using metadata produced during lightweight JIT compilation. Using the stack pointer, the Tracer can also inspect the current values on the JVML stack which is necessary for trace recording. After processing the message, if the Tracer wants to continue recording, it replies to the `visitBytecode` message with the *zero* address which resumes the execution of the current bytecode. Otherwise, if the Tracer wants to stop recording it can reply with the instruction address of the equivalent current bytecode in the profile instrumented version of the method. Effectively, switching back to the more efficient non-trace instrumented method version.

```
resumeAddr = visitBytecode(RIP, RSP, RBP);
if (resumeAddr != 0) {
    jump(resumeAddr);
}
```

Let $m_0, m_1, ..., m_n$ be a ordered sequence of messages. And let $m \mapsto action$ indicate the *action* taken by the Tracer upon receiving the message m, the default

action being *continue*. Moreover, let $f\colon m$ signify the message m sent from method f, and f' the trace instrumented version of method f. Finally, let a_i and b_i be the `visitAnchor` and `visitBytecode` message respectively.

The sequence of messages sent in a canonical tracing example can be written as: $..., f\colon a_0, f\colon a_0 \mapsto record, ..., f'\colon b_0, f'\colon b_1, ..., f'\colon a_0 \mapsto stop, f\colon a_0 \mapsto execute, ...$

Anchor messages $f\colon a_0$ are received, until the Tracer decides to start recording. A sequence of bytecode messagesges b_i are then received until the anchor message $f'\colon a_0$ is received, which indicates that a cycle has been recorded that trace recording should stop. Execution is then resumed in f, after which $f\colon a_0$ messageges trigger the execution of the compiled trace. To execute a trace, the Tracer simply returns the address of the compiled trace.

Complications arise during the recording process if methods are invoked. An inlining policy dictates which methods should be partially inlined, (or traced through). If a method is inlined, the trace instrumented version of the callee method is invoked which will continue to send bytecode messages. If the method is not inlined, the Tracer is placed on hold and is only resumed once the callee method returns. While the Tracer is on hold, execution in the callee method, or further down on the execution stack may trigger yet other traces to be recorded and executed. Putting the Tracer on hold, may prohibit these traces from being recorded. For this reason, we use a stack of Tracers. Once one Tracer is on hold waiting for the callee method to return it is pushed on the Tracer stack, and a new Tracer is created that is ready to record and execute additional traces. Once the callee method returns the old Tracer is poped off the Tracer stack and recording is continued. At any one point, any number of Tracers can be on the stack, but only one Tracer is active, while the remaining ones are on hold. Using our previous notation, a trace recording example involving a method call can be written as:
$f\colon a_0 \mapsto record, f'\colon b_{invoke} \mapsto push, ..., f'\colon b_{invoke+1} \mapsto pop, ..., f'\colon a_0 \mapsto stop$

3.3 Compilation

In order to integrate neatly into the Maxine VM, we compile trace trees as methods. We rely on the method compilation infrastructure to make other subsystems in the virtual machine work. Stack walking, garbage collection, debugging support all rely on the method as being the sole computational element. Trace trees can be modeled into methods by constructing in/out method parameters for each of the used live variables flowing into the trace tree. Live variables that are only used within the tree can be passed by value while live variables that are modified within the tree are passed by reference. Liveness and type information is computed during class loading by the bytecode verifier. Side exits write live variables into out parameters and return the guard that fails. We use this calling convention to support the execution of trace trees from lightweight JIT methods as well as from other trace trees as can be the case for nested trace trees.

Calling trace trees from lightweight JIT methods requires the use of an adapter frame which maps live variables to the in/out parameters expected by the trace tree. Once the trace tree method returns, the adapter frame resumes execution in the lightweight JIT method at the appropriate side exit program location. Calling trace trees from other trace trees does not require the use of an adapter frame, and is more efficient.

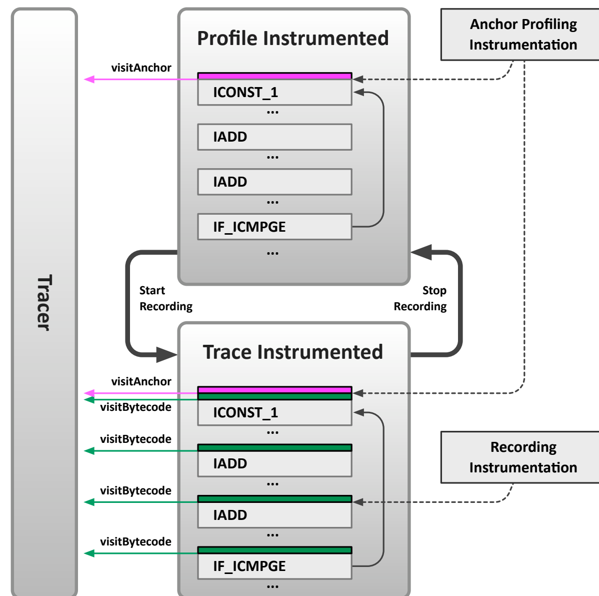

Fig. 5. Design of a lightweight JIT based trace recorder. In order to minimize instrumentation overhead, two versions of a method are compiled. The first, profile instrumented, version contains *anchors* that monitor the execution frequency of select program locations and trigger trace recording and compilation. The second, trace instrumented, version contains recording instrumentation that is used to record traces. The two method versions can be used interchangeably because they share a common stack frame layout.

3.4 Side Exits

One of the challenging aspects of trace compilation is building an efficient side exiting mechanism. Although trace trees attempt to capture as much control flow as possible, a significant number of side exit still occur. Also, not all side exits occur with the same frequency. An array index guard may never fail, while a loop condition guard may fail quite frequently. A side exiting strategy should take this into account, and provide an efficient side exiting mechanism for guard that fail frequently.

3.4.1 *Fast Side Exits.* Fast side exits are used for guards that are inserted for explicit bytecode branching instructions. Traces are more likely to exit at these instructions than they are at array index checks, null checks, or type checks. Fast side exits are compiled right into the trace trees. Machine code is emitted that writes back live variables into the in/out trace tree method parameters when a fast side exit occurs. Fast side exits are limited in that they cannot be used for guards inserted due to method inlining, or deep guards, as opposed to shallow guards. Deep guards require the construction of additional lightweight JIT method frames and is an expensive operation. An additional limitation is that fast side exits require the generation of machine code which increases compilation time, and code cash size. In our experiments we have observed that 80% of the compiled trace tree code is

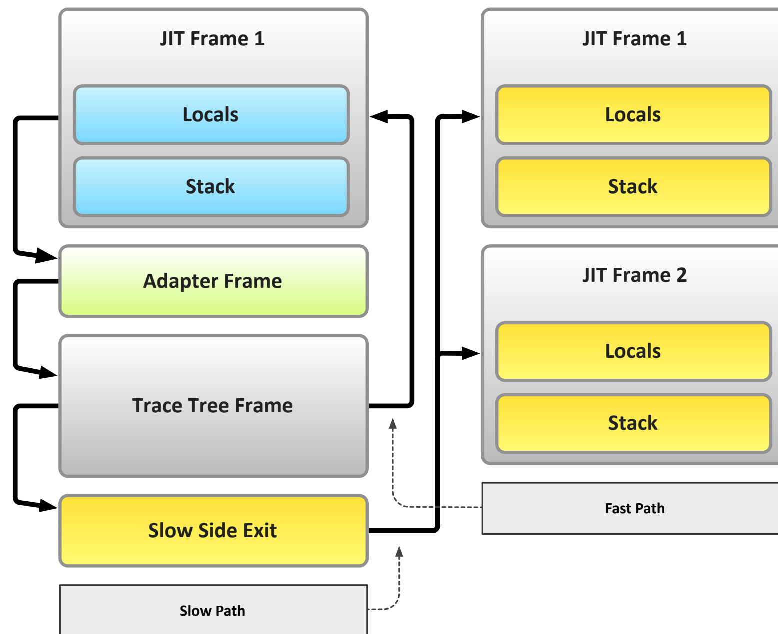

Fig. 6. Invocation of a trace tree from a lightweight JIT method. Live variables are passed to the trace tree via an adapter frame. For fast side exits the trace tree updates live variables and returns back to the adapter frame, which restores the lightweight JIT state. For slow side exits, the trace tree uses a more complicated side exiting process that involves the creation of additional lightweight JIT method frames if the side exit occurs in an inlined method.

side exit code if all guards use fast side exits. This is clearly inefficient, and is the reason for which we use a hybrid side exiting approach, where only some of the guards use fast side exits and others use slow side exits.

3.4.2 *Slow Side Exits.* At every slow side exit, the compiler keeps track of the register/stack locations of live variables and maintains an auxiliary data structure with this information. If a slow side exit occurs, a side exit routine is called that inspects this data structure and reconstructs lightweight JIT frames with the updated live variables. The side exit routine works by first saving all machine registers and then reading and writing values from the preceding trace tree frame and lightweight JIT frame respectively. This is very similar to the de-optimization process in many virtual machines. Although slow side exits are slower than fast side exits, they drastically reduce compilation time and code cash size which indirectly increases the performance of trace trees.

An added benefit of slow side exits it that they provide a convenient way to detect when side exits occur. Knowing when and where trace trees side exit is necessary for recording additional traces, or nesting trace trees. In our implementation trace trees can only be extended at slow side exits. If a slow side exit occurs, a `visitSideExit` message is sent to the Tracer. If trace recording is commenced, execution is resumed in the trace instrumented version of the method at which the side exit occurred.

3.5 Trace Tree Life Cycle

Trace trees start off in the *discovery* phase. In this phase, anchor instrumentation decides when the trace tree should be recorded. In the current implementation we use execution counters to make this decision. However, execution counts are not always the best strategy. The optimal strategy would be to find the sweet spot, trace compilation overhead is payed off by the benefit gained from the compiled trace tree. For example, a simple counting loop takes takes roughly 5ms to compile using our trace compiler.

```
int sum = 0;
for (int i = 0; i < count; i++) {
    sum += i;
}
```

This loop would need to execute roughly 500,000 iterations before the trace compilation overhead is payed off. Compilation overhead for more complicated loops is payed off earlier. In essence, a better heuristic would be to scale the recording threshold by the amount of work done in the loop body.

After the initial compilation the tree enters the *growth* phase. All guards are compiled as slow side exits to permit additional traces to be recorded. The tree is incrementally grown and recompiled as new side exits are explored. The tree exits the growth phase after a few milliseconds and enters the *running* phase. The tree is recompiled with fast side exits whenever possible. Future extensions of the tree are no longer permitted. At this point, the tree is also *welded* to the lightweight JIT method by patching the anchor instrumentation code with a direct call to the trace tree method, thus increasing efficiency. This shortcut, eliminates the sending of `visitAnchor` from lightweight JIT methods to the Tracer. This is a neat optimization trick, but it increases complexity of the Tracer, since it no longer gets notified when a trace tree is executed.

If a trace tree cannot be recorded, for any reason, the anchor instrumentation can be removed by patching it with NOPs or a near jump over the instrumentation code.

4. EVALUATION

In this section we evaluate the performance of our trace based compiler in the Maxine VM [Sun Microsystems 2008]. We also provide a comparison to the highly optimized C/C++ implementation of Sun's Java HotSpot VM [Sun Microsystems, Inc. 2006]. The Maxine VM is still in its infancy, and does not provide many of the advanced optimizations found in production virtual machines. Maxine is written almost entirely in Java and it relies on its own fairly simplistic optimizing compiler to compile itself. It does not benefit from any of the more advanced optimizations available in C/C++ compilers, and therefore it's at a severe disadvantage when compared to many C/C++ based virtual machines. Maxine also lacks an efficient garbage collector. For these reasons we have chosen the Java Grande 2 (REF) benchmark suite because it offers a wide range of problem sizes. From small scale micro benchmarks (section 1) and larger benchmark kernels (section 2) to large problems in (section 3). These benchmarks are less sensitive to the performance

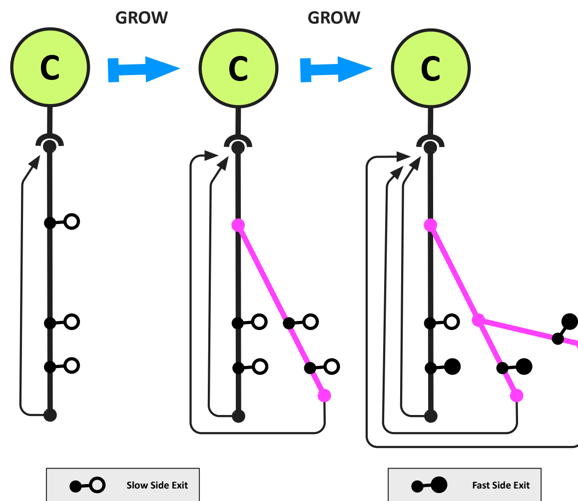

Fig. 7. A trace tree in various growth phases.

of virtual machine subsystems such as garbage collection, stack walking, exception handling, etc. These are problem domains that are outside the scope of trace compilation.

In our prototype implementation, we piggyback on Maxine's optimizing compiler to perform register allocation and the generation of machine code. The compilation benefits of traces are ignored at this level. Register allocation assumes a general control flow and does not do any trace specific optimizations. The same is true for the assembler, and other compiler backed subsystems in the virtual machine. Therefore, the quality of trace compiled code is the same as that of regular methods.

5. RELATED WORK

Tracing is a well established technique for dynamic profile guided optimization of native binaries. Bala et al. [2000] introduced tracing as method for runtime optimizing native program binaries in their Dynamo system. They used backward brach targets as candidates for start of a trace, but did not attempt to capture traces of loops. Zaleski et al. [2007] used Dynamo-like tracing in order to achieve inlining, indirect jump elimination, and other optimizations for Java. Their primary goal was to build an interpreter that could be extended to a tracing VM.

Gal et al. [2006] proposed an 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, they use runtime profiling to detect frequently executed cyclic code paths in the program. The 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. A major benefit of this approach is that the trace tree data structure only contains actually relevant code areas. Edges that are not executed at runtime (but appear in the static CFG)

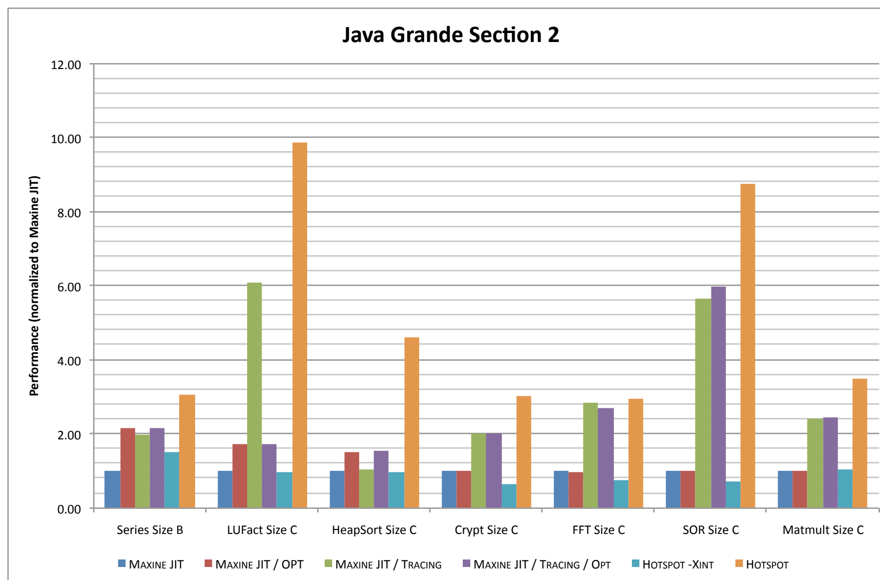

Fig. 8. Benchmark results for Java Grande Section 2. Results are normalize to the baseline Maxine JIT (higher is better.)

are not considered in the trace representation, and are delegated to an interpreter in the rare cases they are taken. The absence of control flow merge points in this tree-based representation greatly simplifies optimization algorithms and this results in optimization passes being quicker than compilers that use traditional CFG based analysis. The system relies on an interpreter to collect traces, while we utilize a just-in-time compiler.

Whaley [2001] uses *partial method compilation* to reduce the granularity of compilation to the sub-method level. His system uses profile information to detect never or rarely executed parts of a method and to ignore them during compilation. If such a part gets executed later, execution continues in the interpreter. Compilation still starts at the beginning of a method.

Similarly, Suganuma et al. [2006] propose *region-based compilation* to overcome the limitations of method-based compilation. They use heuristics and profiles to identify and eliminate rarely executed sections of code. In combination with method inlining, they try to group all frequently executed code in one compilation unit, but to exclude infrequently executed code. If an excluded code part has to be executed, they rely on recompilation and OSR. Our trace-based compilation reaches this goal without requiring complex heuristics. They 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.

Merrill and Hazelwood [2008] presented a solution, implemented on the Jikes RVM, for selecting trace fragments within an non-interpreter based JVM. Their system compiles each method into two equivalent binary representations: a low fidelity region with counters to profile hot loops and a high-fidelity region that

has instrumentation to sample every code block reached by a trace. When a hot loop has been identified, the low-fidelity code transfers control to the high-fidelity region for trace formation. Upon conclusion of a trace, execution jumps back to the appropriate low-fidelity region. This approach has to generate the above mentioned code regions even for methods that are never touched by a trace, and because the bytecodes are already lowered to machine code, high level compiler optimizations like CSE and loop invariant code motion cannot be performed on the traced code.

6. FUTURE WORK

Our current prototype does not support calling trace trees from other trace trees. Trace trees can only be executed from lightweight JIT methods. We plan on adding this feature, since it is important for programs that spend most of their time in recursions. This feature is also needed for nested trace trees. Currently nesting trace trees that appear in inlined methods is not supported. In our current approach, nested trees are actually compiled together as one compilation unit. If we allowed nesting of trace trees in inlined methods, we would see trees replicated at each nested tree call site and this would have a adverse effect on the code cash.

We are also investigating the use of linear traces. Trace trees have a hard time capturing code that is executed frequently but is not wrapped in a loop, such is the case for recursions and in many leaf methods. A static analysis of the control flow, along with runtime profiling can lead to better trace compilation results in such cases.

7. CONCLUSION

This paper describes how a trace compiler can be integrated into a interpreter-less execution environment. The paper explains the trace recording, compilation and execution techniques used in the Maxine VM. We show how dynamic instrumentation can be used to support trace compilation and execution. We present a hybrid side exit strategy and how it can be used to facilitate trace tree growth. Our experimental results show that our technique achieves a speedup comparable to that of a traditional method based compiler, but at a much reduced level of complexity.

REFERENCES

Bala, V., Duesterwald, E., and Banerjia, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1–12.

Gal, A. 2006. Efficient bytecode verification and compilation in a virtual machine. Ph.D. thesis, University of California, Irvine.

Gal, A., Eich, B., Shaver, M., Anderson, D., Kaplan, B., Hoare, G., Mandelin, D., Zbarsky, B., Orendorff, J., Bebenita, M., Chang, M., Franz, M., Smith, E., Reitmaier, R., and Haghighat, M. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press.

Gal, A., Probst, C. W., and Franz, M. 2006. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM Press, 144–153.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *The Java^{TM} Language Specification*, 3rd ed. Addison-Wesley.

HEJLSBERG, A., WILTAMUTH, S., AND GOLDE, P. 2003. *C# Language Specification*. Addison-Wesley.

Jikes 2009. *Jikes RVM*. `http://jikesrvm.org/`.

MERRILL, D. AND HAZELWOOD, K. 2008. Trace fragment selection within method-based JVMs. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM Press, 41–50.

SUGANUMA, T., YASUE, T., AND NAKATANI, T. 2006. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems 28,* 1, 134–174.

SUN MICROSYSTEMS. 2008. Maxine Virtual Machine. `http://research.sun.com/projects/maxine/`.

Sun Microsystems, Inc. 2006. *The Java HotSpot Performance Engine Architecture*. Sun Microsystems, Inc. `http://java.sun.com/products/hotspot/whitepaper.html`.

WHALEY, J. 2001. Partial method compilation using dynamic profile information. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM Press, 166–179.

ZALESKI, M., BROWN, A. D., AND STOODLEY, K. 2007. Yeti: a gradually extensible trace interpreter. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM Press, 83–93.