Incremental Dynamic Code Generation with Trace Trees

Andreas Gal
University of California, Irvine
gal@uci.edu

Michael Franz
University of California, Irvine
franz@uci.edu

Abstract

The unit of compilation for traditional just-in-time compilers is the method. We have explored trace-based compilation, in which the unit of compilation is a loop, potentially spanning multiple methods and even library code. Using a new intermediate representation that is discovered and updated lazily on-demand while the program is being executed, our compiler generates code that is competitive with traditional dynamic compilers, but that uses only a fraction of the compile time and memory footprint.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—*Incremental compilers; optimization*

General Terms Design, Experimentation, Performance

Keywords Java Virtual Machine, trace-based just-in-time compilation, compiler-internal representations, Trace Trees

1. Introduction

Just-in-time compilers are often quite similar in structure to their static counterparts. While they may be employing techniques specifically to reduce compilation time (for example, using linear-scan register allocation instead of a graph-coloring algorithm), they typically start off by constructing a control-flow graph (CFG) for the code to be compiled, then perform a series of optimization steps based on this graph, and as a final step traverse the CFG and emit native code. In addition to a simple CFG, more ambitious optimizing compilers often use an intermediate representation based on Static Single Assignment (SSA) form [8]. Generating SSA is an expensive operation, which is why in the dynamic compilation context this technique is used only for "workstation class" compilers.

In this paper, we explore a different approach to building compilers in which no CFG is ever constructed. Instead, our compiler records and generates code from dynamically recorded *code traces*. Each code trace represents a loop in the program and may potentially span several basic blocks across several methods, even including library code.

It has been shown [12] that generating SSA for a trace is much simpler than doing so for a general CFG, and that constructing a compiler based on trace-driven SSA generation has benefits. However, this earlier work requires the program to have a dominant trace that is taken on most loop iterations. If a program has several alternative execution paths that were all almost equally likely, then this existing technique is not competitive with CFG-based compilers.

The work presented in this paper closes the gap that remains between trace-based compilers and CFG-based ones. We describe a new way of building compilers based on dynamically discovered traces, and our method is successful even when there are several alternative traces of which none is dominant. Key to our approach is a new data structure for representing partially overlapping traces, such as a loop that contains an `if-then-else` condition. Our method is able to dynamically and lazily discover both alternatives and jointly optimize both paths through the loop.

We have built a prototype just-in-time compiler based on our new compilation method. In this paper, we present benchmarks showing that our compiler generates code that is almost as good as that generated by Sun's Java HotSpot compiler. The latter is a CFG-based compiler that is several orders of magnitude larger and slower and is a mature product developed by a large team of programmers over several years. Our compiler is a research prototype developed by a single graduate student in under a year.

The rest of this paper is organized as follows: In Section 2, we introduce the control-flow graph model that underpins the intermediate representations used in virtually every existing compiler. Section 3 discusses our alternative Trace Tree representation, its construction, and its on-demand extension. Section 4 explains how such Trace Trees are compiled. In Section 5, we discuss our prototype implementation of a dynamic compiler based on Trace Trees. Related work is discussed in Section 6, and the paper ends with our conclusions in Section 7.

2. The Traditional Control Flow Graph Model

The traditional control flow graph model represents a program as $G = (\mathcal{B}, \mathcal{E})$ where G is a *directed graph*, $\mathcal{B}$ is the set of *basic blocks* $\{b_1, b_2, \ldots, b_n\}$ in G, and $\mathcal{E}$ is a set of *directed edges* $\{(b_i, b_j), (b_k, b_l), \ldots\}$. Figure 1 shows the graphical representation of such a graph. Since *methods* can be understood as sub-programs, we can use the terms program and methods interchangeably in this context.

Each basic block $b \in \mathcal{B}$ is a linear sequence of instructions. A basic block is always entered at the top (first instruction), and always continues until the last instruction is reached. After executing all instructions in a basic block b_i, execution continues with an *immediate successor block* of b_i.

The existence of a direct edge from b_i to such a successor block b_j is indicated through an ordered pair (b_i, b_j) of nodes. Note that blocks can succeed themselves (a tight loop consisting of a single basic block), and thus the elements of said pair can be identical.

The set of all immediate successor blocks of a block b_i is characterized through a successor function $\Gamma^1_G(b_i) = \{b_j | (b_i, b_j) \in \mathcal{E}\}$, and it can be empty only for the *terminal node* $x \in \mathcal{B}$, which terminates the program: $\Gamma^1_G(x) = \emptyset$.

```
1:  code;
2:  do {
        if (condition) {
3:          code;
        } else {
4:          code;
        }
5:  } while (condition);
6:  code;
```

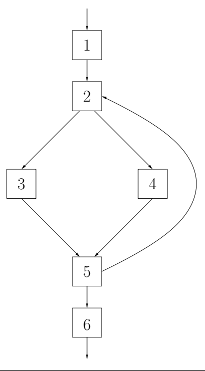

Figure 1. A sample program and its corresponding control flow graph. The starting point of each basic block is indicated through a corresponding label in the program code.

The set of *immediate predecessors* of b_j (the set of basic blocks that can branch to b_j) is characterized through the inverse of the successor function: $\Gamma_G^{-1}(b_j) = \{b_i | (b_i, b_j) \in \mathcal{E}\}$. It can be empty only for the *entry node* $e \in \mathcal{B}$, which is the first block executed when running P: $\Gamma_G^{-1}(e) = \emptyset$.

A *path* P along edges of a graph G can be expressed a sequence of nodes $(b_1, b_2, \ldots, b_n)$ of G. Each node in the sequence is an immediate successor of the predecessor node in the sequence: $b_{i+1} \in \Gamma_G^1(b_i)$. A path does not have to consist of distinct blocks and can contain the same blocks (and implicitly the same edges) repeatedly.

Figure 1 shows a sample program consisting of an `if-then-else` condition inside a `do/while` loop and the corresponding control flow graph with $\mathcal{B} = \{1, 2, 3, 4, 5, 6\}$ and edges $\mathcal{E} = \{(1, 2), (2, 3), (2, 4), (3, 5), (4, 5), (5, 6)\}$. In this example, both $(1, 2, 3)$ and $(1, 2, 4)$ are valid paths, and so is $(1, 2, 3, 5, 2, 3)$ since paths are allowed to contain the same node multiple times. $(1, 2, 5)$ on the other hand is not a valid path, because $(2, 5) \notin \mathcal{E}$.

A *cycle* C in a control flow graph is a path $(b_1, b_2, \ldots, b_n)$ where $b_1 = b_n$. In Figure 1, $(2, 3, 5, 2)$ and $(2, 4, 5, 2)$ are both valid cycles, and so is $(2, 3, 5, 2, 3, 5, 2)$. Cycles in a control flow graph correspond to loops in the original program. $(2, 3, 5, 2)$ and $(2, 4, 5, 2)$ are in fact two different cycles through the same loop and the `if-then-else` construct in it.

For the purpose of this paper, we will assume that no unconnected basic blocks exist in control flow graphs. This means that for every node b_i of a graph G there exists a valid path P of the form $(e, \ldots, b_i)$ that leads from the entry node e to b_i.

3. Representing Partial Programs as Trace Trees

A *trace tree* $TT = (\mathcal{N}, \mathcal{P})$ is a directed graph representing a set of related cycles (*traces*) in the program, where $\mathcal{N}$ is a set of nodes (*instructions*), and $\mathcal{P}$ is a set of directed edges $\{(n_i, n_j), (n_k, n_l), \ldots\}$ between them. Each node $n \in \mathcal{N}$ is labeled with an *operation* from the set of valid operations defined by the (virtual) machine language we are compiling. For the purpose of this paper it is sufficient to consider two operations: *bc* and *op*, with *bc* representing conditional branch operations (for example `ifeq` or `lookupswitch` in case of JVML [20]), and *op* representing all other non branching operations.

Each directed edge (n_i, n_j) in $\mathcal{P}$ indicates that instruction n_j is executed immediately before instruction n_i executes. Thus, we also refer to an edge $(n_i, n_j) \in \mathcal{P}$ as *predecessor edge*.

Similarly to the control flow graph, we define a successor function $\Gamma_{TT}^1(n_j) = \{n_i | (n_i, n_j) \in \mathcal{P}\}$, which returns all instructions that have a predecessor edge pointing to n_j, and thus can execute

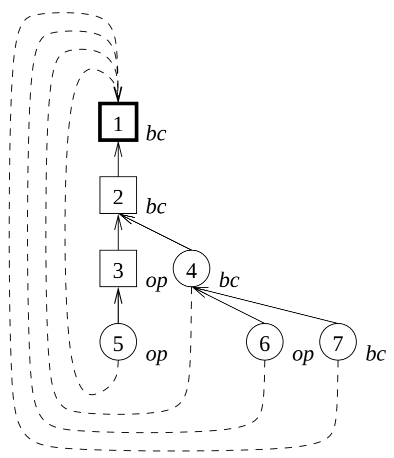

Figure 2. A sample trace tree. The tree shown in this example is not related to the control-flow graph in Figure 1.

immediately *after* instruction n_j. Instructions labeled with *op* have at most one successor instruction: $\forall n \in \mathcal{N} \wedge \texttt{label}(n) = op : |\Gamma_{TT}^1(n)| \leq 1$. These correspond to non-branching instructions in the virtual machine language (such as *add* or *mul*). Instructions labeled with *bc* can have an arbitrary number of successor instructions, including no successors at all. This is the case because edges are added to the trace tree lazily as it is built. If only one edge is ever observed for a conditional branch, all other edges never appear in the trace tree.

A node can have no successors if the control flow is observed to always return to the anchor node a (defined below), since this back-edge is considered implicit and does not appear in the predecessor edge set $\mathcal{P}$. Such instructions that have an implicit back-edge to a are called *leaf* nodes, and we denote the set of leaf nodes of a trace tree TT as *leaf set* $\mathcal{L}$. Leaf nodes can be branching (*bc*) or non-branching (*op*). A non-branching leaf node implies an unconditional jump back to the anchor node following the leaf node. A branching leaf node corresponds to a conditional branch back to the anchor node.

The predecessor function $\Gamma_{TT}^{-1}(n_i) = \{n_j | (n_i, n_j) \in \mathcal{P}\}$ returns the set of instructions an instruction n_i has predecessor edges pointing to. There is exactly one node a such that $\Gamma_{TT}^{-1}(a) = \emptyset$, and we called it the *anchor node*. All other nodes have exactly one predecessor node, and thus the predecessor function returns a set containing exactly one node: $\forall n \in \mathcal{N} \wedge n \neq a : |\Gamma_T^{-1}(n)| = 1$. This gives the directed graph the structure of a *directed rooted tree*, with the anchor node a as its root.

Each leaf node of the trace tree represents the last node in a cycle of instructions that started at the anchor node and ends at the anchor node, and we call each of these cycles a *trace*. The anchor node is shared between all traces, and the traces split up in a tree-like structure from there.

Figure 2 shows a visual representation of an example trace tree formed by set of nodes $\mathcal{B} = \{1, 2, 3, 4, 5, 6, 7\}$ and the set of predecessor edges $\mathcal{P} = \{(2, 1), (3, 2), (4, 2), (5, 3), (6, 4), (7, 4)\}$. Only solid edges are true predecessor edges. Dashed edges denote implicit back-edges, and are not part of the trace tree. The anchor node of the trace tree is 1 and is shown in bold. Leaf nodes $\mathcal{L} = \{4, 5, 6, 7\}$ are drawn as circles. Each instruction is labeled with its operation (*bc* or *op*).

3.1 Constructing a Trace Tree

In contrast to control flow graphs, our IR does not require complete knowledge of all nodes (basic blocks) and directed edges between them. Thus, no static analysis of the program is performed. Instead, our IR is constructed and extended lazily and on demand while the program is being executed by an *interpreter*.

Once a *trace graph* has been constructed, and every time it is extended, the graph is compiled (or recompiled) and subsequent executions of the covered program region are handed off to the compiled code instead of interpreting it.

Since loops often dominate the overall execution time of programs, the trace tree data structure is designed to represent loop regions of programs, and loop regions of programs only.

The first step to constructing a trace graph is locating a suitable *anchor node* $a \in \mathcal{B}$. Since we are interested in loop regions, *loop headers* are ideally suited as anchor nodes since they are shared by all cycles through the loop.

To identify loop headers, we use a simple heuristic that first appeared in Dynamo [1], a framework for dynamic runtime optimization of binary code.

Initially, our virtual machine interprets the bytecode program instruction by instruction. Each time a backwards branch instruction is executed, the destination of that jump is a potential loop header. The general flow of control in bytecode is forward, and thus each loop has to contain at least one backward branch.

To filter *actual* loop headers from the superset of *potential* loop headers, we track the invocation frequency of (potential) loop headers. After the execution frequency of a potential loop header exceeds a certain threshold, our VM marks the instruction as anchor node and will start recording bytecode instructions. Recording stops when a cycle is found, and the resulting trace is added to the tree.

Not all instructions and edges are suitable for inclusion in a trace tree. Exception edges, for example, indicate by their very semantics an exceptional (and usually rare) program state. Thus, whenever an exception occurs while recording a trace, the trace is voided, the trace recorder is disengaged and regular interpretation resumes.

Similarly, certain expensive instructions abort trace recording. Memory allocation instructions, for example, often take hundreds of cycles to execute, and thus the cost of the actual memory allocation operation by far exceeds the potential runtime savings that can be realized through compiling that code region to native code. This is often not a real limitation for performance critical code, since programmers are aware of the cost of memory allocation and tend to use pre-allocated data structures in-place in performance critical loops.

The third possible abort condition for trace recording is an overlong trace. This is necessary because in theory it would be possible to cover the entire program within a single trace tree, at the expense of creating a huge trace tree that would grow exponentially. This is clearly not what we strive for, and thus we limit traces to a certain length before they must either cycle back to the anchor node or be aborted.

The left side of Figure 3 shows a potential initial trace that could be recorded for the control flow graph in Figure 1. The trace starts at node 2, which is the anchor node for this new trace tree (which currently consists of a single trace). An alternative initial trace would have been $(2, 4, 5)$ instead of $(2, 3, 5)$ which is shown in the figure, since both start at the same anchor node $a = 2$. Which trace is recorded first solely depends on the conditional branch following the anchor node. If any particular cycle dominates a loop (i.e. the same path is taken through a loop most of the time), statistically the first recorded trace is likely to be that dominating cycle.

Nodes in Figure 3 labeled with *s* symbolize potential *side exits* from the trace tree. A side exit is a path originating from a node in the trace tree that is not covered by the trace tree itself. Every branching node (*bc*) that has fewer incoming edges in the trace graph than possible successor nodes in the corresponding control flow graph is thus a potential side exit.

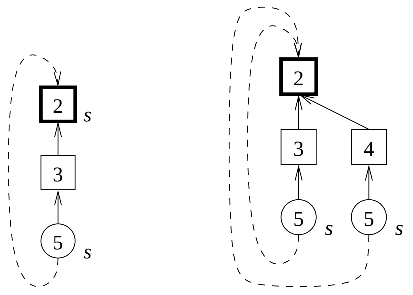

Figure 3. The left figure shows an initial trace recorded for the loop construct shown in Figure 1. Side exits nodes are labeled with *s*. The right figure shows lazy extension of the trace tree after executing the initial trace $(2, 3, 5)$ and encountering a side exit at node 2. The new trace shares all instructions between the anchor node and the side exit with the previous trace it branched off from, which in this example is only the anchor node 2 itself.

In case of the side exit in node 2 this is the edge to node 4, which is an alternative cycles through the loop. Since this path is also a cycle through the loop, it will be added to the trace tree if it is ever taken. The side exit edge originating at node 5, however, leaves the loop, and since it does not circle back to the loop edge, we are not interested in considering it as part of this trace tree and it will remain a side exit edge.

When a trace tree is compiled to native code, side exit edges generate code that restores the virtual machine state and then resumes interpretation of bytecode instructions by the virtual machine. The rationale of this approach is that we want to compile only frequently executed *loop code*, while infrequently executed non-loop code is interpreted by the virtual machine.

Once a trace tree has been constructed (initially containing only a single trace), it is compiled to native code (which we will discuss in more detail in Section 4). For subsequent executions starting from the anchor instruction a, the compiled native code is executed instead of interpreting a.

As long as the control flow follows the previous recorded trace $(2, 3, 5)$, the program will execute as native code without further interaction with the VM. In other words, as long as only explicit and implicit edges of the previous compiled trace tree are taken, the compiled code keeps executing.

3.2 Extending Trace Trees

If trace $(2, 3, 5)$ is the sole actual cycle through the loop (i.e. 2 always branches to 3 because its branch condition is invariant), the trace tree does not have to be extended any further to achieve good dynamic compilation results. If $2 \rightarrow 3$ is a true edge, however, we will see a side exit at 2 as soon as this edge is taken.

As discussed in Section 3.1, instructions that can cause a side exit are compiled in such a way that the virtual machine state is restored and interpretation resumed at the corresponding instruction if such a side exit occurs.

Since we detect anchor nodes by monitoring their execution frequency, we know that they are located in *hot* program code that is executed frequently. If we record a trace starting at such an anchor node a, it is likely that the entire trace consists of frequently executed program code. Thus, if a side exit occurs along such a cycle, such a side exit is likely going to occur frequently. Since side exits are expensive to perform it is desirable to extend the trace tree to include such alternative traces. The trace tree is then recompiled

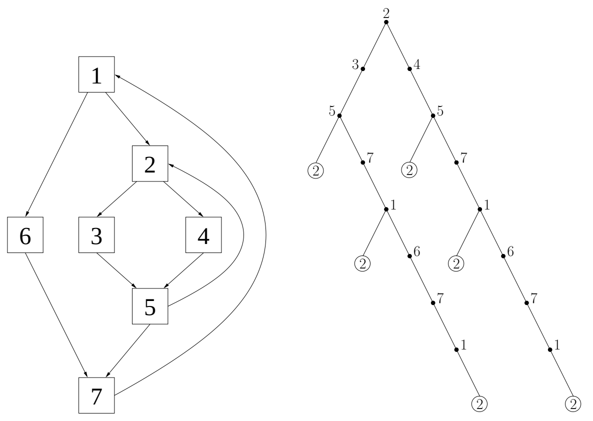

Figure 4. Example of a nested loop and a set of suitable traces through the nested loop. For simplicity, implicit edges are shown as a simple edge going to a copy of the anchor node 2, instead of a dashed edge back to the actual anchor node 2.

and subsequent encounters of the edge that triggered this side exit will no longer result in aborting native code execution.

Upon resuming from a trace tree side exit, the virtual machine immediately re-enters recording mode. The recorded trace is only added to the current trace tree if it cycles back to the anchor node of the trace tree without becoming overly long or executing any instructions and edges that are never included in a trace (such as exception edges).

The right side of Figure 3 shows the extended trace tree resulting from adding the trace $(2, 4, 5)$ to the initially recorded trace tree on the left. The new trace shares all instructions "upstream" (all instructions between the side exit node and the anchor node) of the (former) side exit node 2, which is in this particular example only the side exit node and anchor node 2 itself. Instructions are never shared "downstream". Instead, duplicates of such instructions are added to the tree. Node 5 is an example for this. It appears in both traces, but since it is located after the side exit, it is not shared but duplicated and appears twice in the trace tree. This node duplication gives the trace tree is characteristic *tree*-like shape, and allows it to be transformed into SSA form and analyzed quickly.

The former side exit node 2 is no longer labeled as a side exit in Figure 3, because all possible successor edges are now part of the trace tree. Thus, once this trace tree is recompiled, subsequent executions of the corresponding native code will no longer cause the VM to be resumed, no matter whether $2 \rightarrow 3$ or $2 \rightarrow 4$ is executed at runtime.

3.3 Nested Loops, Topological Trace Ordering

Trace graphs are not restricted to representing simple (non-nested) loops. Two interesting effects can be observed when recording a trace graph for a nested loop. First, the loop header of the inner loop tends to be selected as anchor node for the trace tree. This is intuitive, since the inner loop is executed more frequently than the outer portions of the loop, and thus the inner back-edge returning to the header instruction of the inner loop first crosses the anchor node threshold. And second, since trace trees have only one anchor node, only one loop header is treated as anchor node. The outer parts of the loop—including the outer loop header—are recorded until the control-flow arrives back at the inner loop header (which is the anchor node). Effectively, we have turned the loop inside out, treating the inner loop as the actual trace loop, and the outer loop as mere cycles originating from the anchor node and going back to it. This dramatically simplifies loop analysis, since our representation automatically simplifies all nested loop constructs to simple loops.

Figure 4 shows the control flow graph for a sample nested loop. Traces are shown in the order they would be recorded (left to right). For simplicity, implicit back-edges back to the anchor node $a = 2$ are drawn as simple edges to a copy of the anchor node drawn with a circle around it. This representation is equivalent to showing the implicit back-edge as dashed edge, and neither the copy of the anchor node nor the edge actually appear in the internal representation. The mere fact that the nodes from which the edge originates are labeled as leaf nodes is sufficient to indicate the existence of this implicit back-edge.

In the example shown in Figure 4, the left-hand side of the inner loop was recorded first as trace $(2, 3, 5)$. The next trace is $(2, 3, 5, 7, 1, 6, 7, 1)$. It consists of the continuation of the initial trace at the side exit at node 5, and then two iterations through the outer loop, because the outer loop header branched to 6 instead of 2 which would have terminated the trace earlier. The next trace is $(2, 3, 5, 7, 1)$. It shares all but the final edge to $1 \rightarrow 2$ with the previous trace. This trace must have been recorded after the previous one, because it splits off at node 1, which is part of the previous trace.

We always show the successor instructions of an instruction that was recorded first (i.e. is part of the *primary* trace) as a straight line. $(5, 7, 1, 6, 7, 1)$ for example is a secondary trace continuation that merges with the primary trace $(2, 3, 5)$ in 5. The naming of primary and secondary traces is recursive in the sense that the back-edge $2 \rightarrow 1$ is a secondary trace to $(5, 7, 1, 6, 7, 1)$, because it was recorded following a side exit from it. Such secondary traces that were recorded later following a side exit are shown coming in at an angle.

The trace tree in Figure 4 is extended further with additional cycles through the right side of the inner loop, and a cycle through the outer loop branching of from this alternative path through the inner loop.

Since trace trees are assembled and extended dynamically, the exact order in which traces are added to the tree is not deterministic, but as we have discussed above, secondary traces are always recorder *after* their primary trace, forming a topological ordering of the traces.

3.4 Bounding Trace Trees

Trace trees can obviously grow indefinitely. Instead of branching back to the anchor $a = 2$, the outer loop in Figure 4 could for example enter a cycle $(6, 7, 1)$ and never return to 2, or only after a large number of iterations. In essence we would repeatedly inline and unroll the outer loop, hoping that at some point the control flow returns to the anchor node $a = 2$.

To limit the growth of such malformed trace tree, in addition to a maximal trace length we also limit the number of allowed backward branches during trace recording. Since each such back-edge is most likely indicative of an iteration of some outer loop, by limiting backward branches we essentially limit the number of times we permit the outer loop to be unrolled before the control must return to the anchor node or we abort recording. In our current prototype implementation we permit 3 back-edges per trace, which is sufficient to generate trace-trees of triply-nested loops as long as the outer loops immediately return to the inner anchor. This is usually the case for most regular loops.

If a trace recording is aborted due to an excessive number of back-edges, we record this in the side-exit node and allow a certain number of additional attempts to record a meaningful trace. The rationale behind this is that potentially we only encountered an unusual irregular loop iteration, and future recordings might reveal a direct path back to the anchor node. Our prototype permits a second record attempt, but we have observed only one loop in our benchmarks that ever required such a second recording attempt.

3.5 Method Calls

Similar to outer loops, method calls are inlined into a trace tree instead of appearing as an actual method invocation instruction. Whereas a *static* method invocation has exactly one target method that will be inlined, *virtual* and *interface* method dispatches can invoke different methods at runtime, depending on the actual type of the receiver object.

During trace recording, we convert static and virtual method invocations into a conditional branch depending on the type of the receiving object. While in theory such a conditional branch instruction sometimes could have hundreds of potential successor nodes, most programs rarely invoke more than two or three specific implementations at single virtual method call site. Each of these target methods can potentially be inlined into the trace graph, as long as the abort conditions described above are not violated (i.e. not too many back-edges are encountered).

Experiments have shown that simple methods can easily be inlined into the trace tree of the invocation site since such methods often contain no or very few conditional branches. Method calls that invoke a method that itself contains a loop, however, are frequently rejected due to an excessive number of back-edges, in particular if the invoked method contains a nested loop. Especially for the latter, we believe that this is not really a problem. Instead of inlining the method into the surrounding scope, the method itself will at some point be recognized as a hot spot and a trace tree will be recorded for it. The slowdown resulting from interpreting the outer scope will not significantly impact overall performance, since the method does contain an expensive nested loop construct and thus optimizing the method by far outweighs the cost of interpreting the surrounding scope (if this was not the case, the method would have been inlined in the first place).

An additional restriction we apply to inlining method calls is that we only permit downcalls, i.e. the anchor node must always be located in the same or a surrounding scope as all leaves (which correspond to tails of traces). This means that we do not follow `return` statements and abort traces that encounter them in scope 0. This does not restrict the maximal trace tree coverage, because for every trace tree we disallow (i.e. a trace tree with the anchor node growing outwards) there is always another possible trace tree that does grow downward (with an anchor outside the method and only the traces reaching inside the method).

This restriction has the added benefit that it simplifies the handling of side exits inside inlined methods. Each side exit node is annotated with an ordered list of scopes that have to be rebuilt in case the traces abruptly ends at that point. By limiting the growth of trace trees in one direction, we always only have to add scopes to the top of the virtual machine stack when recovering from a side exit, and we never have to deal with removing method frames from the stack because a side exit happens in a scope further out than the anchor node (where the trace was entered).

The treatment of side exits in our system significantly differs from trace-based native-code to native-code optimization systems such as Dynamo [1]. When inlining code from invoked methods we do not create a stack frame for the invoked method call at each iteration. Instead, the additional local variables are allocated into machine registers, just as local variables in the outermost scope of the trace tree. When a side exit occurs inside code inlined from a method, we materialize the additional method frames on the stack before writing back the stack and local variable state.

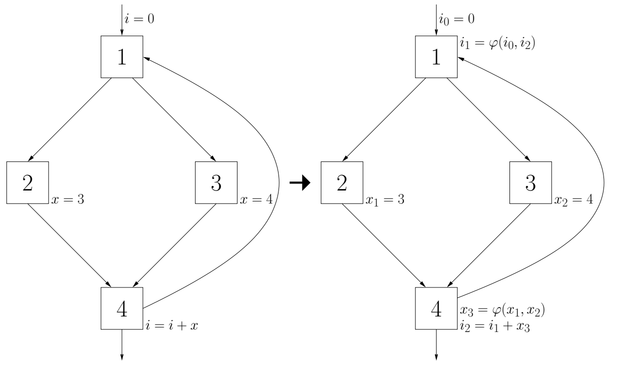

Figure 5. Traditional Static Single Assignment form.

4. Compiling Trace Trees

To compile trace trees, we use a variant of Static Single Assignment (SSA) form [8]. In traditional SSA, multiple definitions of variables are renamed such that each new variable is written to exactly once. ϕ-instructions are inserted in control-flow merge points where multiple values of an original variable flow together. Figure 5 shows an example for transformation into SSA form in case of a loop containing an `if-then-else` statement. A ϕ-instruction has to be inserted in the basic block following the `if-then-else` statement to merge the values of x, which now has been split up into x_1 and x_2 for the left and right branch respectively. Another ϕ-instruction is inserted into the loop header (node 1) to merge the values for i, which can either be the initial value of i when the loop was entered (for the first iteration), or the value from the previous iteration for all subsequent cycles through the loop.

4.1 Trace Static Single Assignment Form

To transform traces into SSA, we perform stack deconstruction during trace recording. References to the Java stack are replaced with direct pointers to the instruction that pushed that value onto the stack. Since each trace corresponds to a cycle through a non-nested loop[1], such pointers either point to an instruction upstream in the trace (closer to the anchor node), or they point to placeholder pseudo-instructions that represent values that were on the stack when the trace started at the anchor node, and local variable references. These pseudo-instructions are generated for all live stack locations and local variables, and are shared by all traces in a trace tree.

Since traces only follow exactly one control flow, they do not contain ϕ-instruction *except* for the anchor node which has two potential predecessor states: the initial state at trace entry, and the state from the previous trace iteration in case of returning to the anchor node from a completed iteration. To distinguish this special form of SSA from its common form, this form is also called *Trace Static Single Assignment* (TSSA) form [12].

The left side of Figure 6 shows the TSSA form for the example trace in Figure 3. If a variable x is assigned a different value in left side of the `if-then-else` statement (3) than the right side (4), we still would not have to insert a ϕ-statement, because this trace only consists of the instructions following the left side of the `if-then-else` statement. A ϕ-statement has to be inserted for the loop variable i, however, to update it after a successful iteration through the trace.

It is important to note that TSSA can be applied here only because we consider each trace separately, even thought traces often

[1] Even in case of nested loops a trace is always a single cycle through it, inlining the outer scopes into the non-nested innermost loop.

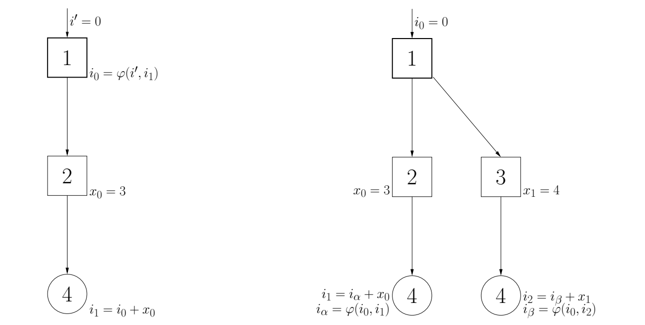

Figure 6. Trace Static Single Assignment (TSSA) form for the initial trace shown in Figure 3 (left), and the Trace Tree Static Single Assignment (TTSSA) form for the entire trace tree (right).

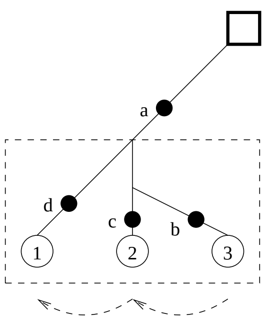

Figure 7. Bottom-up traverse of a trace tree during code generation in reverse recording order of the leaves of the tree.

share instructions "upstream". Thus, we do not insert a ϕ-statement after branching nodes that have several potential successor instruction. Instead, each of those traces is considered a mere linear sequence, and is converted into TSSA separately.

4.2 Trace Tree Static Single Assignment Form

Treating each trace individually for SSA-form purposes of course creates a collision of their respective ϕ-instructions in the anchor node, since each trace would try to place a ϕ-instruction into the anchor node for each loop variable or value that it uses from the surrounding virtual machine content. To avoid this collision, when constructing SSA form for an entire trace tree—which we refer to as *Trace Tree Static Single Assignment Form* (TTSSA)—, ϕ-instructions are always attached to the leaf node of the trace. Since each trace has its unique leaf node, no collision can ever occur.[2]

The right side of Figure 6 shows the final TTSSA form for the trace tree we constructed in the previous section. References to the surrounding context (which are represented by placeholder pseudo-instruction) are always labeled with a zero index. Variable i_0 thus corresponds to the initial value of loop variable i when the trace tree was entered. Since all traces share the same pseudo-instructions when accessing the surrounding context, such *context* references are also shared by all traces. Variable assignments inside traces are denoted with a unique numeric index, whereas ϕ-instructions are denoted with a unique letter index (i.e. i_α and i_β).

4.3 Code Generation and Register Allocation

Every time a trace tree is created or extended, we recompile the entire tree and re-generate the associated native code from scratch. While this seems to be wasteful and slow at first, it is important to realize that we only have to recompile a set of sequences of linear instructions, which can be done quickly and in linear time. Thus, a compiler using a trace tree representation is extremely fast.

When compiling a trace tree, the first step is to identify loop variables. A loop variable is any value in the surrounding context (stack or local variables) that is modified inside the loop. The presence of a ϕ-instruction updating a context variable indicates a loop-variable. If only one trace has a ϕ-instruction for a variable, its sufficient to make it a loop-variable, because all other traces have to respect the register assignment made for that variable, otherwise its value could be overwritten unwittingly.

The rationale for recompiling the entire tree every time a trace is added is that when a new trace is added, it might reveal additional loop variables, which for efficiency reasons we might want to allocate into a dedicated hardware register. For this effect, that hardware register must not be used along any possible trace path. In addition, such new traces can also reveal additional instructions that can be hoisted out of the loop, and thus again need a dedicated register, potentially invalidating previous register assignment assumption.

When compiling traces, we want to emit shared code for instructions that are shared "upstream" between traces. For this, we start compiling traces at their leaves, and ascend the trace tree until we hit an instruction for which we are *not* the primary successor, which means that the instruction we just came from was not the instruction that initially followed that instruction when it was added to the tree. A similar bottom-up code compilation approach for traces was proposed by Gal et al. [12], and has a number of advantages for code generation and instruction pattern matching. For the propose of this paper, we will only focus on the register allocation, since it is unique for our trace tree representation.

If we start compiling at leaf 1 in Figure 7, for example, we would not stop emitting code until we hit the anchor node, because we only follow the primary trace (which is symbolized here through a straight line). When compiling leaf 2, however, we would stop the compilation run when hitting the primary trace (which started at 1), because we leave the upstream code to be compiled when ascending from the leaf of the primary trace itself.

When ascending the trace tree along a trace path, we assign registers to all operands of every instruction we encounter. Due to the structure of our tree, all instructions that execute before the instruction being compiled are located "upstream" in the tree. In this sense the trace tree could amongst others also be considered a dependence graph. Thus, it is guaranteed that for a single trace, when compiling from bottom, we will always see all uses of an instruction before we see its actually definition.

Every time we approach an actual definition, we check whether a register was assigned to it previously. If this is the case, some "downstream" instruction uses the value generated by it, and we generate code for the definition. If no "downstream" instruction assigned a register to the definition (and its side-effect free), we do not generate code for it, which in essence is equivalent to performing dead code elimination.

The order in which we compile traces is crucial, however. While it is guaranteed that in any particular *trace* we will see all uses of a definition before we encounter a definition, this guarantee does not hold for the entire trace tree. Consider the example in Figure 7. If we start compiling at leaf 1 and then compile leaves 2 and 3, we would encounter the definition site (a) right after passing over the downstream use d, but before we have visited uses b and c. This is the case because when starting at leaf 1 we do not stop ascending the tree at the merge point between a and d since the corresponding merge point is the primary continuation of the path coming from 1.

[2] Traces can contain inlined copies of leaf nodes of other traces, but they still have their own leaf node further "downstream".

To guarantee that *all* uses (b, c, d) are seen before the definition (a), we have to find an ordering of the leaves such that all code downstream of a is generated before a itself is visited. In Figure 7, this is symbolized by a dashed box around the corresponding trace parts.

While not the only valid ordering, the reverse recording order of traces always fulfills this requirement, because traces can only be extended once they have been recorded. Thus, in the recording order of traces, a primary trace always occurs before any trace that depends on it. A secondary trace could never be inserted into the tree if its corresponding primary trace isn't present, because there is no place to attach it to.

Reversing this ordering guarantees that all dependent traces are visited before we consider the primary edge into the final merge point closest to the definition. In Figure 7, this means that compilation would start at leaf 3, stop when hitting its primary trace that originates at leaf 2, at which point code generation continues starting at leaf 2 until it is stops again at the merge point with the primary trace 1, which then finally is compiled in one sweep without interruption (since we are ascending the trace along its primary edge in the merge point).

When compiling the tree by ascending it in reverse recording order, registers are always blocked by the register allocator as soon as use is encountered, and the register is not freed again until the definition is reached (which means that all uses were visited). This implicitly also correctly blocks loop variables for the entire code, since as soon as the first use of a loop variable is encountered, that register will be blocked until the anchor node is compiled—which always happens last.

An important restriction for registers assigned to loop variables is that they cannot be used *anywhere* in the generated code *except* for a loop variable as long as the loop variable is not dead along a trace. This is the case because if we would use a register r_1 in a trace and then subsequently assign r_1 to a loop variable further up in the tree, the already generated code would unknowingly overwrite r_1 because when it was initially compiled, r_1 wasn't blocked yet.

To prevent this problem, every time a hardware register is used within a trace tree, we mark it as *dirty* and this flag is sticky even after the register is released again. While such dirty registers can be re-used for non-loop variables, loop-variables are only allocated from the pool of *virgin* registers that have never been used before. Our prototype implementation assigns registers in ascending order $(1, 2, 3, \ldots)$ to non-loop variables, and in descending order to loop variables $(31, 30, 29, \ldots)$. If there are not sufficient hardware registers (the compiler runs out of virgin registers), the compilation run is restarted and virtual registers which are mapped to memory through spilling are added to the pool. A more sophisticated approach would involve splitting live ranges and making loop registers available for the general pool once they become dead along a particular trace.

5. Results: Trace Trees in Practice

We have implemented a prototype trace tree-based dynamic compiler for the JamVM [21] virtual machine. JamVM is a small virtual machine suitable for embedded devices. The JIT compiler itself consists of roughly 2000 lines of C code, of which 800 lines are used by the front end that records traces and deconstructs the stack. TTSSA transformation, liveness analysis, common subexpression elimination and invariant code analysis are contained in another 200 lines of C code. This remarkable code density is possible, because linear sequences of instruction traces are much simpler to analyze than corresponding control-flow graphs. The remaining 800 lines of code consist of the PowerPC backend, which transforms our trace tree into native PowerPC machine code.

Compiled to native code, our JIT compiler has a footprint of approximately 150 kBytes (code plus static and dynamic data, including the code buffer), which is noteworthy for a compiler that implements a series of aggressive optimizations. To evaluate the performance of our prototype dynamic compiler, ideally we would like to compare it to existing commercial embedded Java virtual machines. Unfortunately, all commercial embedded JVMs (such as Sun's CLDC Hotspot VM system [27]) are available to OEMs only, and we were unable to obtain a license to perform comparative benchmarks. A small number of free JVM implementations exist that target embedded systems, including the JamVM virtual machine that we used as basis platform. However, most of these do not have a dynamic compiler and thus are of limited use for a direct performance comparison.

An additional complication arises from the fact that embedded systems are not nearly as homogenous as the desktop computing market. A number of different processor architectures compete for the embedded systems space, including StrongARM/XScale, PowerPC, SPARC, SH and others. Not all embedded VMs are available for all platforms, further complicating comparative benchmarks. In addition to the interpreter-only JamVM, we use Kaffe [29] and Cacao [18] as representatives for embedded virtual machines with just-in-time compilation. While both systems were not primarily designed for embedded use, both contain a baseline JIT-compiler that does not perform aggressive optimization, which is very similar to the state of the art in embedded dynamic compilation. Thus, even though a direct comparison with commercial embedded VMs would be preferable, the comparison with Kaffe and Cacao should provide an estimate on how our trace-based compiler performs in comparison to traditional space-saving and complexity-saving dynamic compilers.

The third VM suitable for embedded use that we used in our benchmarks is SableVM [11], which sacrifices some portability aspects of the interpreter concept in favor of execution performance. SableVM is small and space efficient, and faster than pure interpeters, but its performance still lags behind optimizing dynamic compilers.

In addition to embedded VMs (or to be more precise two VMs representative for this class of systems), we also compare our prototype to Sun's heavyweight JVM Hotspot systems. JVM Hotspot is clearly not designed for embedded use, and the code size of the JIT compiler alone is over 3.5 MBytes, which is more than 20 times the total size of our system. The goal of our system is to deliver performance close to JVM Hotspot, at a much lower cost.

Figure 8 shows the benchmark results for running the Java Grande [22] benchmark suite on a 1.8 GHz PowerPC G5 with 1GB RAM using the virtual machines outlined above. All virtual machines are compared against the JVM Hotspot virtual machine running in interpretation-only mode (`-Xint`).

With a speedup of 7 and 10 respectively, our Trace Tree Compiler (TTC) and JVM Hotspot significantly outperform all other VMs in all benchmarks. Only Cacao is able to achieve comparable performance for two benchmarks (LUFact and SOR). We have excluded the Series kernel from this comparison. It measures mostly the performance of trigonometric functions, which for our platform (PowerPC) all VMs implement as native library code.

While JVM Hotspot still out outperforms our system (speedup 10 vs speedup 7), this additional performance cost comes at a runtime price that is unaffordable for most embedded system. The runtime compilation costs of our system in comparison to the Hotspot compiler are shown in Figure 9 and Figure 10. On average, our JIT compiler is 350 times faster than the Hotspot compiler, and emits 30 times less native code.

In addition to the native code buffer our system also has to maintain the intermediate representation in memory across com-

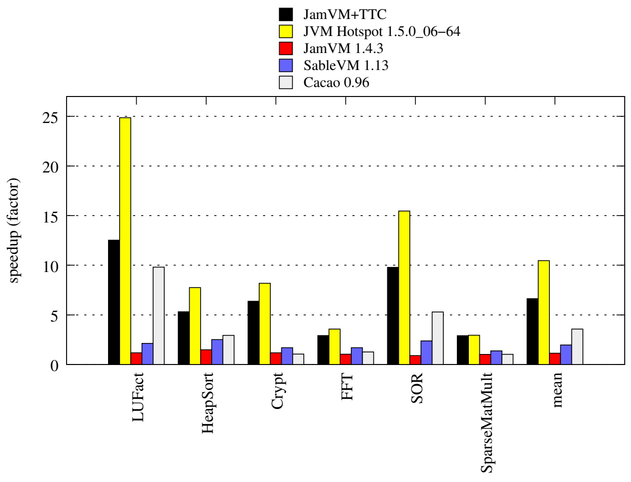

Figure 8. Speedup. Compiled code performance relative to pure interpretation for several dynamic compilers.

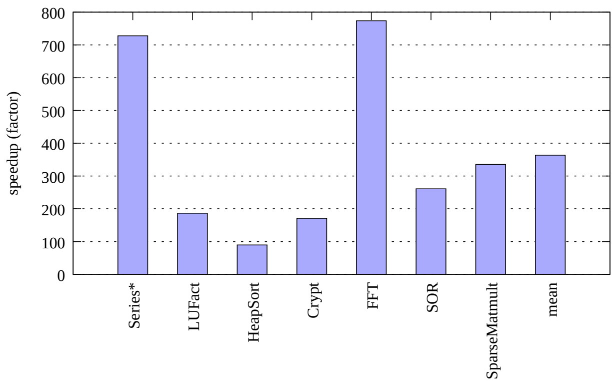

Figure 9. Compile Time: Time required for just-in-time compilation (of all paths that eventually get compiled) in our system relative to JVM Hotspot.

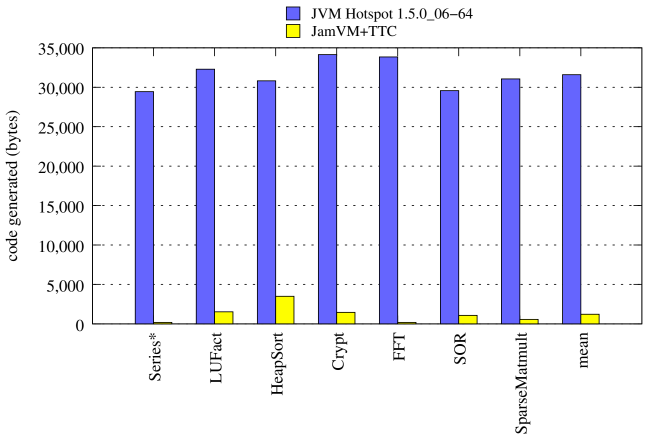

Figure 10. Code Size: Cumulative size of native code (of all paths that eventually get compiled) emitted by our system compared to JVM Hotspot.

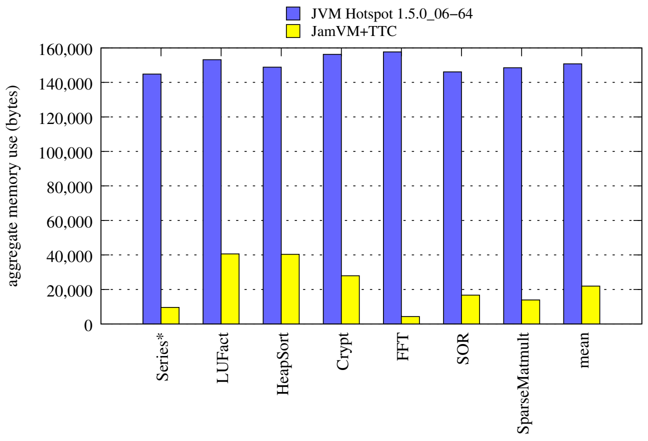

Figure 11. Memory Usage: Aggregate size of all intermediate data structures (including intermediate representation) for our system compared to JVM HotSpot.

piler runs since each additional trace has to be merged with the previously built trace graph. Figure 11 shows the aggregate memory consumption for the intermediate representation in comparison to Hotspot. Our system is 7 times more memory efficient than Hotspot.

It is important to note that the intermediate representation has to be held in memory only for a very short period of time. Benchmarks have shown that traces are added to traces in rapid succession, quickly reaching a point where additional traces do not significantly improve performance anymore. We have observed a less than 5% performance penalty when limiting the number of traces in a tree to 5, which allows all trees to be completed in less than 100ms. Once the tree is complete, the intermediate representation can be discarded.

Disposing of the IR, however, does not automatically preclude further optimization of the code area. It only means that any additional optimization has to start "from scratch" by recording a fresh initial trace, followed by subsequent extensions of the trace tree. It is conceivable that program phase shifts cause excessive side exits after a tree was completed, and the prototype system could be extended to discard the optimized code and re-record the code area. Even repeated recompilations should not cause a significant performance problem considering the vast performance advantage of our system in comparison to existing heavy weight compilers.

6. Related Work

Trace-based dynamic compilation is closely related to *trace scheduling*, which was proposed by Fisher [10] to efficiently generate code for VLIW (very long instruction word) architectures. The trace scheduling approach differs from our system in that a trace scheduling compiler always compiles all code in a program, whereas our dynamic compiler exclusively focuses on frequently executed "hot" program regions, and leaves it to the virtual machine interpreter to execute compensation code and rarely executed code regions.

When a trace scheduling compiler selects a trace, it disregards all but the favored execution path through the basic blocks involved in that trace. While the favored path is the most frequently executed path (if the predictions are accurate), the control flow can still unexpectedly diverge from a trace (*side exit*), or unexpectedly branch into a trace (*side entry*). Side exits can be dealt with with little overhead. If code was scheduled past a side exit, for example, the same code can be duplicated along the side exit path to ensure that it is always executed–even in case of a side exit. Side entry points on the other hand require a significant bookkeeping overhead to generate the required compensation code, since the compensation code can differ depending from where the side entry originates.

Based on this observation, Hwu et. al proposed superblocks as an alternative to trace scheduling. A superblock is a trace (or a set of traces) that has no side entries. A simple method for creating superblocks is to first collect regular traces, and then eliminate side entries along the trace path using tail duplication [7]. Static program analysis has been proposed as well for superblock formation [13].

Similar to superblocks, our trace trees have only a single entry point and no side entries. When extending trace trees, we follow side exits if they reconnect to the trace tree entry point, which could be considered an extension of Chang et al.'s tail duplication algorithm.

Compiling only partial "hot" program traces was first introduced in the Dynamo system by Bala et al. [1]. Dynamo is a transparent binary optimizer that records frequently executed traces and optimizes instructions in that trace. While the Dynamo work differs significantly in its target domain and implementation (Dynamo both records and emits machine code traces, while our system records bytecode traces and compiles them to machine code using SSA), its trace selection and recording mechanism is very closely related to our work.

Abandoning the traditional use of methods as compilation units for dynamic compilation in embedded systems was first proposed by Bruening et al. [4, 5]. The authors made the observation that compiling entire methods can be inefficient even when only focusing on "hot" methods. Instead, they propose compiling based on hot traces and loops.

Whaley [28] also found that excluding rarely executed code when compiling a program significantly increases compilation performance and code quality. The author discusses two examples of code optimizations that benefit from such rare path information: a partial dead code elimination path, and a rare-path-sensitive pointer and escape analysis. Compensation code is inserted to fall back to the interpreter when a rarely taken path (which has not been not compiled) is executed. A direct performance comparison of Whaley's work and our system is not possible, because Whaley didn't implement an actual dynamic compiler but instead simulated the results using profiling and off-line code transformation to benchmark the expected performance of the proposed system.

Berndel et al. [2] also investigated the use of traces in a Java Virtual Machine. However, while providing a mechanism for trace selection, the authors do not actually implement any code compilation or optimization techniques based on these traces.

Bradel et al. [3] propose using traces for inlining methods calls in a Java Virtual Machine. Similar to our work, they use trace recording to extract only those parts of a method that are relevant for the specific caller site.

Dynamic compilation with traces uses dynamic profile information to identify and record frequently executed code traces (paths). By dynamically adding them to a trace tree, and thus iteratively extending that trace tree as more traces are discovered, we perform a form of feedback directed optimization, which was first proposed by Hansen [14]. Feedback-directed optimization was used heavily in the SELF system [6] to produce efficient machine code from a prototype-object based language, and subsequently also used in conjunction with other highly dynamic languages such as Java.

A problem frequently encountered by feedback-directed re-optimizers is how to substitute a newly compiled code region for its currently running counterpart. A long-running loop inside a method, for example, can only be replaced with an optimized version if the runtime system knows how to replace the currently running code including all the associated state with the newly compiled version. Such an exchange of code versions is often called *on-stack replacement* [15].

Similar to traditional feedback-oriented and continuous compilers, our trace compiler compiles an initial version of a trace tree consisting of a single trace, and then iteratively extends the trace tree by recompiling the entire tree. A key advantage of our trace-based compiler over traditional frameworks is the simplicity of the on-stack replacement mechanism. Instead of having to handle machine-code to machine-code state transitions (which are highly complex), trace-trees always write back all state onto the Java stack when a side-exit is encountered. Recording restarts at such side exit points, and once the trace tree has been extended it is compiled and entered the next time execution arrives at the header node. In essence, in our system on-stack replacement comes for free because we never deal with methods in the first place. We can change the code layout at every side-exit from a loop.

Recently, the GNU C Compiler (GCC) framework introduced a tree-shaped SSA-based intermediate representation [23, 24]. Similar to our work, the control flow is simplified to reduce optimization complexity. However, in contrast to our work, the authors use static analysis to generate this Tree SSA representation, whereas we dynamically record and extend trace trees. The Tree SSA approach also does not transform nested loops into simple loops, forcing the compiler to deal with the added complexity of SSA in the presence of nested loops and complex control-flows. This might be appropriate in a static compilation setting such as GCC, but would be too time consuming in the context of dynamic compilation.

A number of existing virtual machines target the embedded and mobile systems domain. The Mate system by Levis et al. [19] provides a Java VM for very small sensor nodes. Sun's Kilobyte Virtual Machine (KVM) [26] targets 32-bit embedded systems with more than 256kB RAM. KVM is not compatible with the full Java standard and only supports a subset of the Java language and runtime libraries. JamVM [21] is not strictly an embedded VM, but it is small enough to be used on small embedded devices. In contrast to KVM, it does support the full Java standard.

Sun has produced a research JIT compiler for the Kilobyte Virtual Machine, called KJIT [25]. KJIT is a lightweight dynamic compiler. In contrast to our trace tree-based JIT compiler, KJIT does not perform any significant code optimization but merely maps bytecode instructions to machine code sequences. Also, KJIT seems to be an internal research project only. We have not been able to obtain it for comparative benchmarks.

Sun's current implementation of an embedded JIT compiler is called CLDC Hotspot VM [27]. Unfortunately, very little is known about the internal details of this compiler. According to Sun's white papers, CLDC Hotspot performs some basic optimizations including constant folding, constant propagation, and loop peeling, while our compiler also applies common subexpression elimination and invariant code motion. Just like KJIT, CLDC Hotspot is unavailable for comparative benchmarks.

Other VM's for the embedded domain include E-Bunny [9] and Jeode EVM [17]. E-bunny is a simple, non-optimizing compiler for x86 that uses stack-based code generation. It is very fast as far as compile time is concerned, but yields poor code quality in comparison to optimizing compilers. Jeode EVM is an optimizing compiler that uses a simplified form of dynamic compilation. Unfortunately, just as with CLDC Hotspot, little is known about its internals.

IBM's J9 Virtual Machine [16] is a modular VM that can be configured to run on small embedded devices, workstation computers, and servers. While in its smallest configuration (which is used for cell phones and PDAs) it can run on embedded platforms, that configuration only supports a very limited set of code optimizations.

Kaffe [29] and Cacao [18] are free virtual machines for Java that are small enough to run on mobile devices. Both contain JIT compilers, but neither performs aggressive code optimization. We use Kaffe and Cacao in our comparative benchmarks because they are suitable for embedded use, and freely available.

7. Conclusions and Outlook

Our work closes the remaining code-quality gap that has existed between CFG-based and trace-based compilation. 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. Using a topological ordering inherent to trace trees, we are able to compile traces individually and then merge the individual compilation results into one coherent and globally optimized native code block. This eliminates the trace-to-trace transition overhead that has existed in previous approaches. The end result is surprisingly good code quality produced by a surprisingly small and frugal compiler.

Where dynamic compilers so far have been quite similar to their static counterparts, "pretty much doing the same things, just at runtime", our approach is markedly different. 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.

As for future work, Sun has announced that it is considering to open source its CLDC Hotspot implementation. Once this code is freely available, we plan on porting our JIT compiler to Sun's embedded virtual machine. This would finally allow meaningful comparative benchmarks with a commercial-strength embedded dynamic compiler. Since Sun's implementation does not support PowerPC, we will also have to retarget our system for StrongARM or MIPS. Due to the small size of our backend, and the overall reduced complexity of our system, this task should be comparatively easy.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2000.

[2] M. Berndl and L. Hendren. Dynamic profiling and trace cache generation for a Java virtual machine. Technical report, McGill University, 2002.

[3] B. Bradel. *The Use of Traces in Optimization*. PhD thesis, University of Toronto, 2004.

[4] D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 13–20, 2000.

[5] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[6] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *Higher-Order and Symbolic Computation*, 4(3):243–281, 1991.

[7] P. Chang, S. Mahlke, and W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[9] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, N. Tawbi, H. Yahyaoui, and S. Zhioua. A dynamic compiler for embedded Java virtual machines. In *PPPJ '04: Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, pages 100–106. Trinity College Dublin, 2004.

[10] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[11] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. *Compiler Construction, 12th International Conference, Jan*, 2003.

[12] 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.

[13] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. mei W. Hwu. Superblock formation using static program analysis. In *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 247–255, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[14] G. J. Hansen. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, March 1974.

[15] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. 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.

[16] IBM. WebSphere Everyplace Custom Environment J9 Virtual Machine. `http://www-306.ibm.com/software/wireless/wece/`, October 2006.

[17] Insignia Solutions. Jeode Platform: Java for Resource-constrained Devices, White Paper. `http://www.insignia.com/`, 2002.

[18] A. Krall and R. Grafl. CACAO: A 64 bit Java VM just in time compiler. In *PPoPP Workshop on Java for Science and Engineering Computation*, 1997.

[19] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[21] R. Lougher. JamVM virtual machine 1.4.3. `http://jamvm.sf.net/`, May 2006.

[22] 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.

[23] D. Novillo. Tree SSA, a new optimization infrastructure for GCC. *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[24] D. Novillo. Design and implementation of Tree SSA. *Proceedings of the GCC Developer's Summit*, 2004.

[25] N. Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 119–126, San Francisco, CA, August 2002.

[26] Sun Microsystems. KVM-Kilobyte virtual machine white paper. Technical report, Sun Microsystems, Palo Alto, CA, 1999.

[27] Sun Microsystems. CLDC HotSpot Implementation Virtual Ma-

chine. http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf, Feb. 2005.

[28] J. Whaley. Partial method compilation using dynamic profile information. *ACM SIGPLAN Notices*, 36(11):166–179, 2001.

[29] T. Wilkinson. Kaffe–a Java virtual machine. http://www.kaffe.org/, October 2006.