Making the Compilation “Pipeline” Explicit: Dynamic Compilation Using Trace Tree Serialization

Andreas Gal, Michael Bebenita, Mason Chang, and Michael Franz

Computer Science Department
University of California, Irvine
Irvine, CA, 92697, USA

{gal,bebenita,changm,franz}@uci.edu

Abstract. Trace-based compilers operate by dynamically discovering loop headers and then recording and compiling all paths through a loop that are executed with sufficient frequency. The different paths through each loop form a tree, with the loop header at the root, in which common code is shared up-stream. Such trace-trees can be serialized in a specific manner that allows us to organize the compiler pipeline as a series of filters. We have implemented such a compiler pipeline that has completely linear runtime behavior. Further, it has only two write barriers, meaning that substantial parts of the compilation effort could potentially be parallelized on future multi-core platforms.

1 Introduction

Most just-in-time (JIT) compilers operate at the granularity of methods. Apart from the fact that it happens at run-time, the actual process of compilation is often not all that different from the way this is done in traditional compilers: First, an intermediate representation incorporating a control-flow graph is built for the entire method. Then, a series of optimization steps is applied, until finally a program in the target instruction set is obtained.

We have been working on an alternative compilation strategy in which no control-flow graph is ever constructed, but in which relevant (i.e., frequently executed) control flows are instead discovered lazily during execution. We have built a compiler that dynamically detects “hot” code paths, so-called “traces”, and generates code on-the-fly for exactly these code paths—all other parts of the program are executed by interpretation only [10].

Our compiler first dynamically detects loop headers and then incrementally discovers alternative paths through the resulting loops. For example, an `if` statement inside of a `while` statement corresponds to two different paths through the `while` loop. Each time that a new path through the loop that hasn’t been recorded before becomes sufficiently “hot”, we recompile all known paths through the loop, re-balancing the processor resources expended on the alternatives.

In this paper, we report how re-compiling the different code paths through a loop can be *pipelined* in a compiler and implemented quite elegantly as a series of filters through which a program passes on its way from intermediate representation to target code. Each compiler phase is implemented as a separate filter that manipulates the instruction "stream" by altering, inserting, re-ordering, or removing instructions as the "stream" passes through it. Even more interestingly, the pipeline requires only two synchronization points, meaning that the compilation activity could potentially be highly parallelized on a a future multicore platform in which there are idle processor resources available.

The remainder of this paper is organized as follows. In Section 2, we describe how alternative paths through a program are discovered lazily and recorded in a representation that we call a "Trace Tree". In Section 3, we show how such Trace Trees can be serialized in such a way that correct compilation semantics result from a set of linear passes over the serialized tree. This is the basis for the idea of pipelining these passes, which is discussed in Section 4. In Section 5 we report on a series of benchmarks that demonstrate the performance of our compiler pipeline. In particular, run-time is directly proportional to the length of the serialized tree. Related work is discussed in Section 6. We end with conclusions and future work in Section 7.

2 Trace Trees

Our trace-based JIT compiler targets the Java Virtual Machine (JVM) bytecode language. It is a mixed-mode execution environment in which bytecode is initially interpreted, and in which only frequently executed ("hot") code is ever compiled. To detect such "hot" code regions that are suitable candidates for dynamic compilation, we use a simple heuristic first introduced by Bala et al. [2]. In this scheme, the interpreter keeps track of backward branches; the targets of such branches are often loop headers. For each target of a backward branch, we keep a counter of how often a backward branch has terminated at this location. When the counter exceeds a certain threshold, a loop header has been discovered.

The interpreter then records the instruction path subsequently taken, starting with the loop header. If that path leads back to the loop header within a certain observation window and fulfills certain other criteria, then we have recorded an initial trace through the loop. There may be other paths through the loop (recall the example of an `if` statement inside of a `while` statement) that may be disovered later. The trace is then compiled and the resulting target code is executed.

Forward branches inside of a trace indicate alternative paths that may not yet have been discovered by our compiler. For example, in the case of our `if` statement inside of a `while` statement, when we initially discover the loop, we only follow one path through the loop. If indeed the `if` condition is so skewed that only one of the two alternatives is ever taken sufficiently often, then our compiler will never generate code for the rare alternative. If the second path is common, then it will eventually be compiled as well.

Branches to paths that have not yet been seen by our compiler are compiled into *guard* instructions. When such a guard fails, that means that we have arrived at a path that has not yet been compiled. This is called a "side exit" from the loop. In this case, the compiled code hands control back to the interpreter. Since this can be expensive, our JIT compiler attempts to compile the alternative path that led to the side exit as well.

For this, at every side exit we resume interpretation, but at the same time we restart the trace recorder to record instructions starting at the side exit point. These secondary traces are completed when the interpreter revisits the original loop header. This results in a tree of traces, spanning all observed paths through the "hot" code region. The rest of this paper explains how we compile such Trace Trees.

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

3 Compiling Traces Trees with Tree Serialization

During trace recording, we perform stack deconstruction and generate an equivalent three address code intermediate representation of JVM bytecode. Each instruction points to the instructions that generated the values of its operands and also points to the instruction immediately preceding it. If a side exit occurs, we link the first instruction in the side trace to the guard instruction that caused the side exit. Thus, a guard instruction can connect at most two traces and each node in a Trace Tree can have at most two successors linking to it. Effectively, a Trace Tree is structured like a binary tree with reversed edges. Trace Trees are always in dominator order, which follows naturally from the way traces are recorded. A preorder depth first traversal of the Trace Tree guarantees that all instruction definitions are seen before any of the uses while the reverse of this guarantees that all uses are seen before any of the definitions. This reversed preorder traversal allows us to serialize arbitrary Trace Trees without losing any information. Since a preorder depth first traversal is not possible using our reversed binary tree data structure, we maintain a back pointing linked list of tail instructions as shown in Figure 1 which allows us to traverse the tree in reversed preorder. This is the order in which Trace Trees are serialized for the compilation pipeline.

3.1 Compilation Pipeline

The compilation pipeline is organized as a series of filters that are chained together. At each point in the pipeline, a filter can remove instructions, modify already existing instructions, re-order instructions, or simply introduce new instructions. Trace Trees are serialized and piped through the first filter in the compilation pipeline. Each filter receives one instruction at a time and forwards it to the next filter until the instruction reaches the final compilation filter, where the instruction is removed from the pipeline. Filters that remove instructions from the pipeline simply do not forward them along, while filters that introduce instructions forward extra instructions. This pipelined architecture makes it easy to construct complex instruction assembly lines. Further in the paper we discuss how the pipeline can be extended to provide various levels of parallelism.

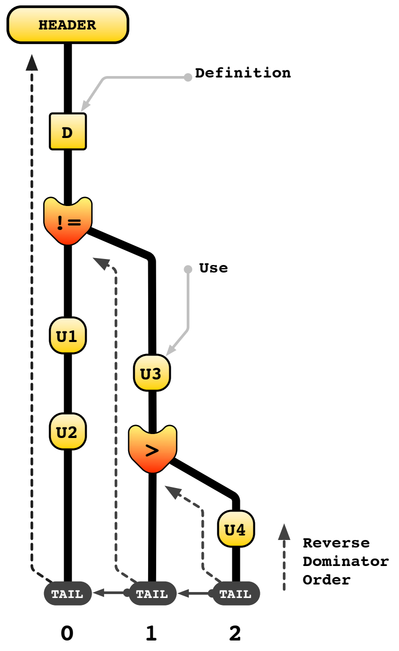


Fig. 1. Traces are recorded in dominator order. The dotted line shows the reversed dominator order used to serialize Trace Trees through the pipeline. This order guarantees that all uses of an instruction are seen before its definition. The reversed dominator order for the tree above is *U4, U3, U2, U1, D*.

3.2 Instruction Class Hierarchy

We make extensive use of the visitor pattern [19, 11] in our compiler architecture. Each compilation filter is in fact a visitor over a sequence of instruction objects. Instructions in our intermediate representation are modeled as class objects and are organized in a class hierarchy based on the instruction's form. The base class of the instruction hierarchy is the abstract `Instruction` class. From this base class, we derive classes based on the instruction's operands such as the `UnaryInstruction` and `BinaryInstruction` classes. The `ADD` instruction, for instance, is derived from the `BinaryInstruction` class as it operates on two operands. This architecture allows us to write optimizations that operate at various levels in the class hierarchy rather than just on leaf classes. It also enables us to neatly separate the implementation of our optimization algorithms from the details of our intermediate representation by making use of the visitor pattern.

The pipeline architecture necessitates a demarcation of individual trees and traces. When Trace Trees are serialized, a `TreeBegin` notification is sent down the pipeline, followed by a `TraceBegin` notification, followed by a sequence of instructions, and then terminating with a `TraceEnd` and `TreeEnd` notification. If a tree has multiple traces, then multiple trace begin/end notifications are sent. These control messages

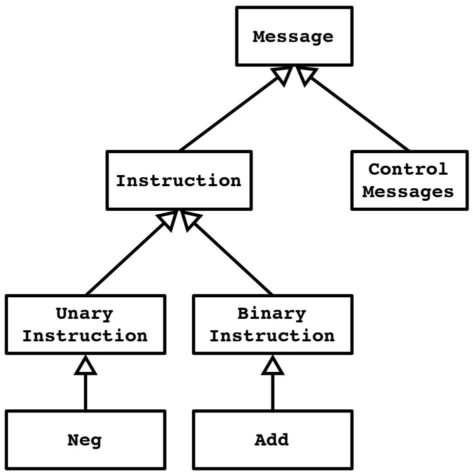


Fig. 2. Partial view of the instruction class hierarchy. Instructions are organized based on instruction form. The root of the hierarchy is the `Instruction` class, however from an implementation standpoint it is convenient to create another level of abstraction, the message class.

```
Instruction i = new Instruction.ADD(leftOperand, rightOperand);
i.accept(new InstructionVisitor() {
  @Override
  void visit(Instruction.BinaryInstruction i) {
    // visited for all binary instructions
  }
});
```

Fig. 3. Example code using the visitor method hierarchy to implement a visitor action for an entire class of instructions at once (here binary instructions).

extend the Instruction hierarchy to include the root `Message` class from which both `Instruction` and each of the individual control message classes inherit (Figure 2).

In our architecture, the `Instruction` class provides an abstract `accept` method with an `InstructionVisitor` object as an argument. Each concrete implementation of the `Instruction` class overrides the `accept` method and invokes the `visit` method on the visitor object using the instruction's own declared type. The default implementation of the `InstructionVisitor` class defines visit methods for all instruction class types.

Figure 3 shows a Java program that uses the visitor method hierarchy to implement a visitor action for an entire class of instructions at once. The `Instruction.ADD` class provides a concrete implementation of the `accept` method declared in `Instruction`. Here, the `visit(Instruction.ADD)` method is invoked on the visitor object that is passed into the `accept` method. The base class implementation of the method `visit(Instruction.ADD)` forwards the invocation to the base class virtual method `visit(Instruction.BinaryInstruction)`, which in turn invokes the catchall method `visit(Instruction)`. This chain of invocations that bubble up to the root of the instruction hierarchy allows us to hook into the instruction hierarchy at any level by extending the `InstructionVisitor` class and overriding a visit method. In this example program, by overriding the implementation of the method `visit(Instruction.BinaryInstruction)` in the anonymous class, we provide an implementation for all binary instructions including the `Instruction.ADD` instruction which derives from `BinaryInstruction`.

3.3 Filter Pattern

In our pipelined compiler architecture, compilation filters are implemented as classes derived from `Filter` (Figure 4), which derives from `InstructionVisitor`. Filters also implement the `Sink` interface which exposes the `receive` method. This is the entry point through which instructions are fed into a filter. Filters accept the incoming instruction and visit it using themselves as an instruction visitor. By default, filters bubble up instructions through a chain of accept method invocations all the way to the root of the instruction hierarchy, at which point instructions are forwarded to the next filter. The code for a compilation filter that performs Common Subexpression Elimination using Value Numbering [20] is shown in Figure 5.

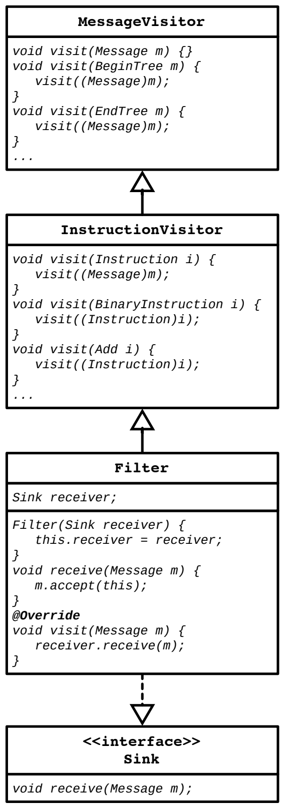
In this example, the CSE class overrides the visit methods for unary, binary, and constant instructions. All other instructions are allowed to pass through the filter untouched. However, for the above mentioned instructions, the CSE filter performs value numbering and keeps a list of previously seen values. If an instruction's value has been seen before, the instruction whose value has been seen takes the place of the current instruction. This is done by keeping a pointer in every instruction to an instruction that replaces it. Further in the pipeline, instructions that are not referenced are eliminated. The CSE filter cannot eliminate redundant instructions because it has no knowledge of future instructions that may reference a redundant instruction. Therefore a forwarding pointer is used instead to indicate that an instruction is substituted with another. Throughout the pipeline, we follow this forwarding chain of instructions every time an operand is accessed.

3.4 Baseline Compiler

Our baseline non-optimizing compiler is assembled from only two filters (Figure 6). A `Clone` filter followed by the `Assembler` filter. The `Clone` filter makes a copy of every instruction it receives and forwards along the copy instead of the original. This is done to ensure that the compilation pipeline does not adversely change the instructions in the original Trace Tree data structure which is constantly extended and recompiled as new traces are added to the tree. The `Assembler` filter performs platform specific register allocation and instruction selection/assembly. The final machine code output is then executed by the interpreter once it reaches the Trace Tree's anchor instruction.

3.5 Optimizing Compiler

The optimizing compiler pipeline extends the baseline pipeline by inserting a sequence of optimization filters between the `Clone` and `Assembler` filters. The optimization



```
void visit(Message m) {}
void visit(BeginTree m) {
   visit((Message)m);
}
void visit(EndTree m) {
   visit((Message)m);
}
...
```

```
void visit(Instruction i) {
   visit((Message)m);
}
void visit(BinaryInstruction i) {
   visit((Instruction)i);
}
void visit(Add i) {
   visit((Instruction)i);
}
...
```

```
Sink receiver;
```

```
Filter(Sink receiver) {
   this.receiver = receiver;
}
void receive(Message m) {
   m.accept(this);
}
@Override
void visit(Message m) {
   receiver.receive(m);
}
```

```
void receive(Message m);
```

Fig. 4. UML diagram of the filter architecture. The `Filter` class extends the `InstructionVisitor` class and implements the `Sink` interface. Messages received by the filter through the `receive(Message)` method are accepted and visited using the filter itself, i.e. `m.accept(this)`. Messages that bubble up the instruction hierarchy are eventually caught in the `visit(Message m)` method and forwarded to the next filter.

pipeline is partitioned into three stages. These stages are separated by `Barrier` filters. `Barrier` filters are pipeline buffers that collect instructions and flush them back once all instructions have been collected. We detect that all instructions have been collected when a `TreeEnd` message is found. In a pipeline without barriers, each instruction traverses the entire pipeline before the next instruction can be sent down the pipeline. This behavior is undesirable for optimization filters that need a full code analysis before proceeding. The full code analysis, which is implemented as a filter, must complete before the optimization filter can start. For this reason we introduce instruction buffers between filters with this type of dependency. The instruction buffer, or barrier, ensures that the code analysis filter has seen the entire Trace Tree before the optimization filter sees any instructions. The architecture of the optimized compiler pipeline (Figure 7) is presented below.

```
public class CSE extends Filter {
   private IdentityHashMap<Value, Instruction> avail =
           new IdentityHashMap<Value, Instruction>();

   public CSE(Sink receiver) {
      super(receiver);
   }
   private void process(Instruction i) {
      Value v = i.getValue();
      if (avail.containsKey(v))
         i.redirectTo(avail.get(v));
      else
         avail.put(v, i);
      receiver.receive(i);
   }
   public void visit(Instruction.UnaryInstruction i) {
      process(i);
   }
   public void visit(Instruction.BinaryInstruction i) {
      process(i);
   }
   public void visit(Instruction.Constant i) {
      process(i);
   }
}
```

Fig. 5. Example for a compilation filter that performs Common Subexpression Elimination using Value Numbering. Only constants and unary and binary instructions are processed, all other instructions and messages are passed through to the next filter (*receiver*). Redundant instructions are replaced by equivalent instructions that were previously observed.

Most pipeline filters expect Trace Trees to be serialized in forward or reverse order, with the exception of the `Clone`, `Reverse`, `Fork`, `Barrier` and `Proxy` filters which operate on trees serialized in any order. The `Reverse` filter reverses the order in which Trace Trees are serialized. Trace Trees are always serialized in reverse order. By having trees serialized in reverse order, the use of an instruction is always seen before its definition. This property is useful for the assembler which performs register allocation, while most optimization and analysis filters need to see the definition of an instruction before its use. By reversing the default Trace Tree serialization order we can ensure this property. Therefore we apply the `Reverse` filter right after the `Clone` filter at the beginning of the optimization pipeline. At the very end of the pipeline we reverse the order again since our assembler performs bottom up code generation. The `Fork` filter connects the previous filter to two receiving filters. Thus any instruction that is received by a `Fork` filter will be forwarded to the two receiving filters. This is useful for debugging purposes, as we can send all instructions to a filter which prints all instructions in the pipeline, while sending the instructions to compilation filters. The `Proxy` filter is described later in the paper.

4 Parallel Compilation and Parallel Pipelining

Two techniques can be used to take advantage of machine-level parallelism to speed up compilation: by either having a parallel compilation, a parallel pipeline, or both

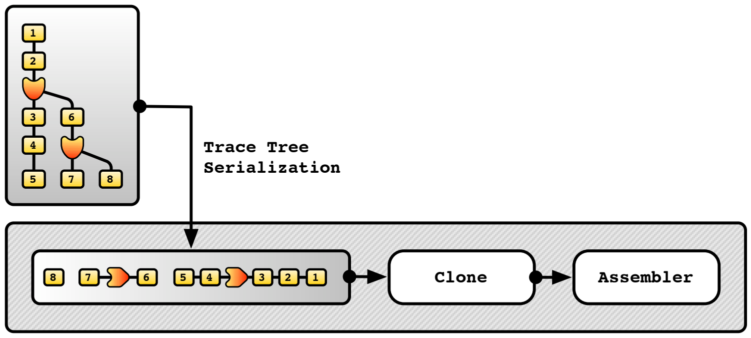


Fig. 6. The Baseline Compiler pipeline is assembled from only two filters. The Trace Tree is serialized in reverse order through a clone filter, which makes a copy of the Trace Tree data structure, and then through the `Assembler` filter which assembles machine code.

simultaneously. These are completely independent of each other. Parallel compilation refers to the use of a separate thread to compile Trace Trees while the main thread continues to interpret the program. A parallel pipeline splits up sections of the pipeline and each section runs its own thread *in parallel*.

In each of these implementations, the threads are controlled via a thread pool. Every compiler phase that requires a thread implements the `Runnable` interface, and asks for a thread from the thread pool to execute.

In case of parallel compilation, the Java interpreter continues executing the main program while the Trace Tree is optimized and compiled. For this, the compiler requests a new thread from the thread pool and begins to compile the recorded trace in that new thread. At the same time, the interpreter continues to interpret the program. Once the final compilation result is available, the compilation thread (or the last compilation thread in case of pipelined compilation) notifies the interpreter that a compiled version of the frequently executed bytecode region is available by annotating the loop header bytecode instruction with a pointer to the compiled code.

In case of a parallel pipeline, we split sections of the pipeline into separate independent threads. Each of these segments are connected via a proxy filter. This proxy filter implements the `Runnable` interface, and uses a message queue to forward incoming messages (instructions) to a new thread that then invokes the receive method of the receiver filter (which is the next filter in the pipeline). It is possible to have a proxy filter between compiler phases. Our current prototype pipeline runs in parallel in 19 different threads. This is, however, not very efficient due to the communication overhead of the message queue. Instead, we split the pipeline into segments, each segment containing multiple filters (Figure 8).

Parallel compilation and the parallel pipeline, can be enabled/disabled independently. We can utilize this to scale the trace compiler to use as many cores as the system provides, by tweaking how many threads are used in the pipeline. Our trace compiler can also use as few as two threads by parallelizing the compilation process itself or by paralellizing the pipeline.

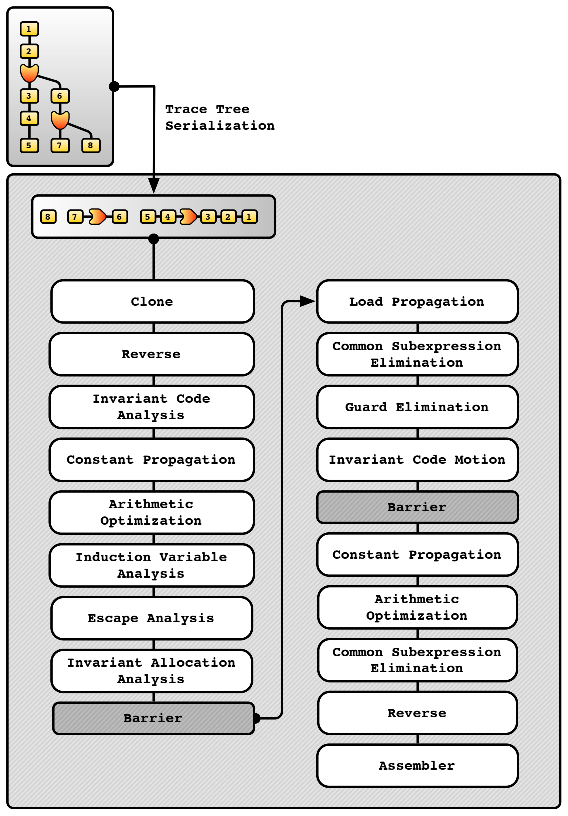


Fig. 7. The optimizing compiler pipeline is assembled from 19 filters, two of which are pipeline barriers. The Trace Tree is serialized in reverse order through a clone filter, and then through a series of basic optimization and analysis filters which make up the first stage of the pipeline. In the second stage, we perform four complex optimizations. Finally in the last stage, we re-apply basic optimizations and assemble machine code.

5 Benchmarks

We have implemented a prototype dynamic compiler that compiles Trace Trees using a compiler pipeline. The prototype compiler compiles Trace Trees recorded for Java programs. The compiler and the compiler pipeline are also implemented in Java [1, 18]. Our system runs on top of Apple's Java VM 6 (Developer Preview). All benchmarks were performed on a Apple MacBook 2.16 Ghz Intel Core 2 Duo with 2 GB RAM. To compensate for measurement errors, each test was run ten times. Our graphs aggregate the results for all ten samples.

Figure 9 shows the results for compiling Trace Trees with increasing size using the prototype compiler pipeline. The compilation time increases linearly with the size of the Trace Tree. The upper part of the figure shows a graph that was recorded by running our

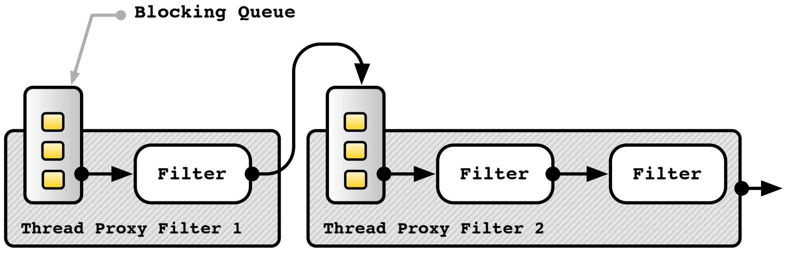


Fig. 8. Visual representation of a parallel pipeline. A proxy object sits between two connecting segments of the pipeline. The proxy itself utilizes and controls the thread it is given. When the proxy receives a new message, it forwards the instruction through all the filters in its segment. Once the instruction has passed through its segment, the thread continues to wait for new instructions. The proxy releases its thread when it receives a `TreeEnd` message.

compiler on top of an interpreting Java VM. The slower execution reduces the impact of measurement errors and influences such as garbage collection. The lower graph shows the performance of the compiler when running as compiled code, in which case our compiler as almost 6 times faster and compiles *and optimizes* graphs with more than 27,000 instructions in less than one second.

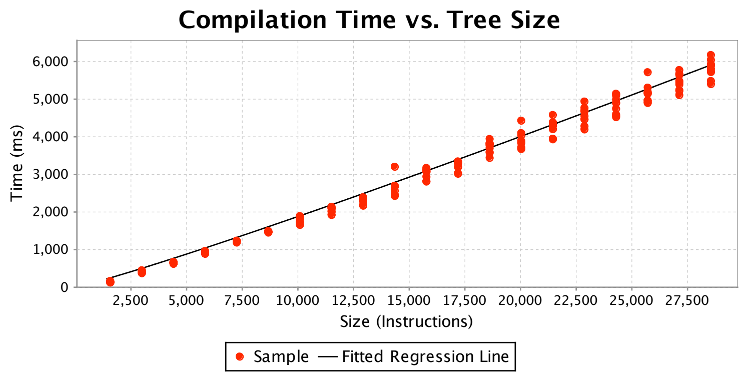
We performed several experiments with different parallelization granularities, executing different parts of the compiler pipeline in parallel using “Proxy” filters. In these experiments we were not able to show a significant compilation time improvement. We believe that the reason for this is twofold.

On the one hand, our compiler is extremely fast and scales very well. Due to the linear nature of traces (and trees of traces), very little work has to be done in each compiler filter. Thus there is little to be gained from parallelizing the pipeline because it is very fast to begin with.

On the other hand, the overhead of using a message queue between two threads running different parts of the pipeline seems to outweigh the performance gain from overlapping those parts of the pipeline. Splitting the pipeline in 4 threads and compiling a large tree with 27,000 instructions, for example, requires sending 108,000 message across the various message queues involved. Amongst others, this requires allocating 108,000 temporary heap objects to transport the messages and the threads involved have to perform a synchronization protocol to avoid race conditions.

6 Related Work

Trace-based dynamic compilation is closely related to *trace scheduling*, which was proposed by Fisher [9] to efficiently generate code for VLIW (very long instruction word) architectures. In Fisher’s system, all code was compiled using trace scheduling. Compiling only partial “hot” program traces was first introduced in the Dynamo system by Bala et al. [2]. Dynamo is a transparent binary optimizer that records frequently executed traces and optimizes instructions in that trace. In contrast to our system, Dynamo



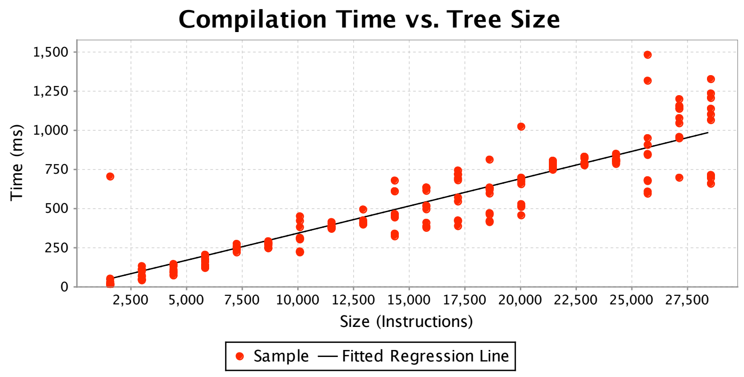


Fig. 9. Linear Regression - Compilation Time vs. Tree Size. The first graph shows the compilation times when executing our compiler through bytecode interpretation. The second graph shows compilation times for running the compiler pipeline as compiled native code.

only optimizes individual traces, whereas in our approach we build tree-shaped hierarchies of traces that represent a (potentially nested) loop region and can be globally optimized.

The trace-tree representation is also closely related to superblocks. In particular, Chang et al.'s work on tail duplication [8] produces an intermediate representation that closely resembles trace-trees. However, in contrast to most superblock approaches that use static analysis to form superblocks [13], our representation is built dynamically at runtime.

Our method of selecting only certain "compilation-worthy" parts of methods is closely related to region-based compilation by Suganuma et al. [21]. Region-based compilation 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. Whaley [23] also found that excluding rarely executed code when compiling a program significantly increases compilation performance and code quality. Our approach further reduces the cost for analysis and optimization by eliminating complex control-flows altogether, merely operating on linear sequences of instructions.

Path profiling was first proposed by Ball et al. [3]. To collect profiling information about program paths, the author use a path encoding that produces unique index numbers for program paths. Profiling code is injected into the code such that a unique index number is generated when a certain sequence of basic blocks (path) is executed, and a profiling counter is associated with each such index number that counts the execution frequency. Our trace-based compilation approach is similar to path profiling in that we perform path specific optimization along each trace. Also, the iterative extension of Trace Trees can be seen as a form of dynamic hot path graph construction. The most significant difference is that we do not incur the actual path profiling runtime overhead, because path specificity is an inherent property of our intermediate representation. Thus, we also do not have to deal with compensation code generation, because we already implicitly perform tail duplication as we record traces.

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 [7] to produce efficient machine code from a prototype-object based language. Feedback directed optimization was subsequently also used in conjunction with other highly dynamic languages such as Scheme [6], Smalltalk [12], and Java.

Hölzle later extended the SELF system to not only use profile-guide compilation, but also adaptive code re-compilation [15]. Kistler et al. [17] further extended this idea and proposed a *continuous* program optimizer for the Oberon system that continuously re-schedules program instructions and dynamically rearranges the layout of objects during execution to maximize performance.

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.

Berndl et al. [4] 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 those traces.

Bradel et al. [5] 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.

Off-loading dynamic compilation to another execution unit while continuing interpretation is frequently used in commercial virtual machines such as Sun's Hotspot Virtual Machine [22] and IBM's J9 virtual machine [16]. Our compilation differs significantly, because it is not merely parallelizable as a whole. Instead, its parallelizable in itself and individual compiler passes can execute overlappingly on different execution units.

7 Conclusions and Future Work

We have presented a novel architecture for dynamic compilers using a pipeline of "filters" implementing individual compiler phases. In this architecture, a Trace Tree is dynamically recorded and extended at runtime and is compiled by serializing the traces in the tree into a single linear sequence of instructions. We have shown that it is possible to serialize trees in this fashion because the dominance information between traces is preserved if the serialization occurs in reverse recording order of traces. In the resulting serialized stream, every definition occurs before all of its uses.

To compile such a serialized tree, it is sent through the pipeline. Each filter in the pipeline can manipulate the instruction stream by modifying, inserting, removing, and re-ordering instructions. The optimization and compilation filters can be put together in different configurations. By directly connecting the final instruction assembly filter to the tree serialization output, for example, we built a non-optimizing compiler. Without modifying either of these filters, we can insert optimization filters in between to obtain an optimizing compiler.

The architecture of our compiler pipeline lends itself very well for parallelization. "Proxy" filters can be inserted to execute parts of the compiler pipeline in a separate thread. The compiler pipeline in our prototype compiler only requires two synchronization points that cannot be overlapped with the rest of the pipeline.

In the benchmarking section we provided a preliminary evaluation of the runtime performance of our compiler. In particular, the compilation cost of the pipeline architecture increases linearly with the size of the compiled Trace Trees. This result is remarkable, considering that we implement several aggressive compiler optimizations that have highly non-linear performance characteristics in traditional compilers. Key to the linearity of our compiler's performance is the linearity of the underlying data structure.

We were not able to show a performance gain from parallelizing the compiler pipeline. We attribute this to the communication and synchronization overhead of sending instructions and messages across the message queues that connect the threads processing the individual pipeline parts. However, we believe that one of the main contributions of our paper is the fact that our compiler pipeline architecture is parallelizable at all. We are not aware of any other dynamic compiler architecture that has the potential to reduce the compilation time for individual compilation units using parallelism. Whether such a parallelized compiler pipeline will actually be able to outperform single-threaded compiler pipelines is subject to future research, and depends on creating a more efficient message queue implementation that reduces the message sending and synchronization overhead.

References

1. K. Arnold and J. Gosling. *The Java programming language (2nd ed.).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
2. V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. *Hewlett Packard Laboratories Technical Report HPL-1999-78. June 1999*, 1999.

3. T. Ball and J. Larus. Efficient path profiling. *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, 1996.
4. M. Berndl and L. Hendren. Dynamic profiling and trace cache generation for a java virtual machine. Technical report, McGill University, 2002.
5. B. Bradel. *The Use of Traces in Optimization*. PhD thesis, University of Toronto, 2004.
6. R. Burger. *Efficient compilation and profile-driven recompilation in scheme*. PhD thesis, Department of Computer Science, Indiana University, 1997.
7. 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.
8. P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Wen-mei. IMPACT: an architectural framework for multiple-instruction-issue processors. *Proceedings of the 18th annual international symposium on Computer architecture*, pages 266–275, 1991.
9. J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.
10. A. Gal, C. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 144–153, 2006.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.
12. A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.
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. *Adaptive optimization for self: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, Department of Computer Science, 1994.
16. IBM. WebSphere Everyplace Custom Environment J9 Virtual Machine. `http://www-306.ibm.com/software/wireless/wece/`, October 2006.
17. T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, 2003.
18. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.
19. J. Palsberg and C. Jay. The essence of the Visitor pattern. *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pages 9–15, 1998.
20. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27, New York, NY, USA, 1988. ACM Press.
21. T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):134–174, 2006.
22. Sun Microsystems. The Java Hotspot Virtual Machine v1.4.1, Sept. 2002.
23. J. Whaley. Partial method compilation using dynamic profile information. *ACM SIGPLAN Notices*, 36(11):166–179, 2001.