HotpathVM: An Effective JIT Compiler for Resource-constrained Devices

Andreas Gal
Donald Bren School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
gal@uci.edu

Christian W. Probst
Informatics and Mathematical Modelling
Technical University of Denmark
2800 Kongens Lyngby, Denmark
probst@imm.dtu.dk

Michael Franz
Donald Bren School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
franz@uci.edu

Keywords Dynamic Compilation, Embedded and Resource-constrained Systems, Mixed-mode Interpretive/compiled Systems, Software Trace Scheduling, Static Single Assignment Form, Virtual Machines

Abstract

We present a just-in-time compiler for a Java VM that is small enough to fit on resource-constrained devices, yet is surprisingly effective. Our system dynamically identifies traces of frequently executed bytecode instructions (which may span several basic blocks across several methods) and compiles them via Static Single Assignment (SSA) construction. Our novel use of SSA form in this context allows to hoist instructions across trace side-exits without necessitating expensive compensation code in off-trace paths. The overall memory consumption (code and data) of our system is only 150 kBytes, yet benchmarks show a speedup that in some cases rivals heavy-weight just-in-time compilers.

1. Introduction

A decade after the arrival of Java, great progress has been made in improving the run-time performance of platform-independent virtual-machine based software. However, using such machine-independent software on resource-constrained devices such as mobile phones and PDAs remains a challenge, as both interpretation and just-in-time compilation of the intermediate VM language run into technological limitations.

Running virtual-machine based code strictly in interpreted mode brings with it severe performance overheads, and as a result requires to run the device's processor at a much higher clock speed than if native code were executed instead. This in turn leads to an increased power consumption, reduced battery autonomy, and may require the overall use of more expensive processors vs. a pure native-code solution. Just-in-time compilation, on the other hand, produces more efficient native code as an end result, but the process of getting to that native code may be very costly for a resource-constrained device to perform in the first place.

VEE'06 June 14–16, 2006, Ottawa, Ontario, Canada.

For example, Sun's Java HotSpot Virtual Machine 1.4.2 for PowerPC includes a just-in-time compiler that achieves an impressive speedup of over 1500% compared to pure interpretation. However, this comes at the price of a total VM size of approximately 7MB, of which about 90% can be attributed to the just-in-time compiler. Clearly, such resources (that don't yet include dynamic memory requirements) are not available on most current embedded devices.

As a consequence, distinct *embedded just-in-time compilers* have emerged, in which trade-offs are being made between resource consumption of the just-in-time compiler and the ultimate execution performance of the code being run on top of the VM. As a representative of this class of VM-JIT compilers, Insignia's Jeode EVM [16] achieves a speedup of 600% over pure interpretation [14].

Embedded just-in-time compilers achieve their results using significantly fewer resources than their larger counterparts mostly by using simpler algorithms. A commonly cited example is the use of linear-scan register allocation instead of a graph-coloring approach, which not only reduces the run-time of the algorithm, but also greatly diminishes the memory footprint. Embedded just-in-time compilers also tend to use far less ambitious data structures than "unconstrained" compilers—for example, while the use of Static Single Assignment form [6] is fairly standard in just-in-time compilers, the time and memory needed to convert just the 10% most frequently executed methods to SSA form using traditional techniques would far exceed the resources of most embedded computers.

In this paper, we present a just-in-time compiler that pursues a new dynamic-compilation approach. Our compiler is an add-on to the JamVM [17], a virtual machine for embedded devices. Unlike other just-in-time compilers that are "intertwined" with the virtual machine hosting them, ours requires changing no more than 20 lines of JamVM's source code. The first prototype of our compiler was in fact designed as an add-on for Sun's KVM [23, 24] virtual machine. Porting the compiler to JamVM only required minimal changes to both our JIT compiler as well as the JamVM source base. Our JIT compiler runs in a total footprint of 150 kBytes (including code and data) while for regular code still achieving speedups similar to those of heavyweight JIT compilers.

Key to the success of our approach is trace-based compilation using SSA form. Similar to other systems before, the HotpathVM JIT compiler dynamically identifies execution traces that are executed frequently—we build dynamic traces from bytecode (which would have been interpreted anyway) rather than from native code, so that the relative overhead of trace recording is much less critical. But the real novelty of our system comes to bear *after* a hot

trace has been identified: it is then dynamically compiled into native code via a nontraditional application of SSA form, which we call Trace SSA (TSSA).

In the classical use of SSA form, a control-flow graph is translated into SSA form in its entirety, and ϕ nodes are placed in control-flow join nodes. Our approach differentiates between the values used in a trace being compiled, which are in SSA form, and values in the rest of the VM, which are not. The VM explicitly moves data from the stack and local variables into dedicated SSA variables before any generated native code is called, and explicitly moves non-dead SSA results back onto the stack and into local variables on every exit from such an optimized trace (including side exits). This approach enables the just-in-time compiler to perform aggressive optimizations on the trace, including moving operations on SSA values across side exit points. Because instruction traces are essentially linear (they may contain only internal back edges) liveness analysis and placement of ϕ nodes are straightforward. Our system supports fairly sophisticated merging of multiple traces that have a common ancestor.

The remainder of this paper is organized as follows. In Section 2, we describe our approach to identify and extract linear code sequences from non-sequential programs. Section 3 describes our trace compiler, which is a specialized JIT compiler that translates bytecode traces to PowerPC machine code. In Section 4 we detail our approach to extend primary traces with secondary traces to cover irregular control-flow scenarios. Then (Section 5), we give a brief overview of the current state of our prototype implementation. Related work is discussed in Section 6. Our paper concludes in Section 7.

2. Trace Selection and Recording

When loading bytecode programs into memory, traces are not readily visible in the code. In fact, most general purpose code is not even purely sequential. The Java bytecode format groups code into methods that are associated with classes, with all code for a class stored in individual class files.

To transform such non-sequential code into linear sequences of instructions, we use an approach called software trace scheduling [9]. Software trace scheduling is based on the observation that a physical CPU does not actually execute a code graph, but merely follows linear instruction sequences (traces) along the edges of that graph. To obtain a trace from a code graph, we record instructions at runtime as they are executed. As not all code is equally worthy of compilation and optimization, we limit our efforts to loops and only start recording bytecode traces once we have identified a frequently executed loop header.

This section gives an overview of the main steps—identifying loop headers, recording traces, and conditions for ending the recording.

2.1 Identifying Loop Headers

To identify loop headers, we use a simple heuristics that first appeared in Dynamo [3], 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 recorded as a potential loop header. The rationale of this approach is that 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. Only after the execution frequency of a potential loop header exceeds a certain threshold, our VM starts recording a bytecode trace (Figure 1). To reduce the overhead for code that is

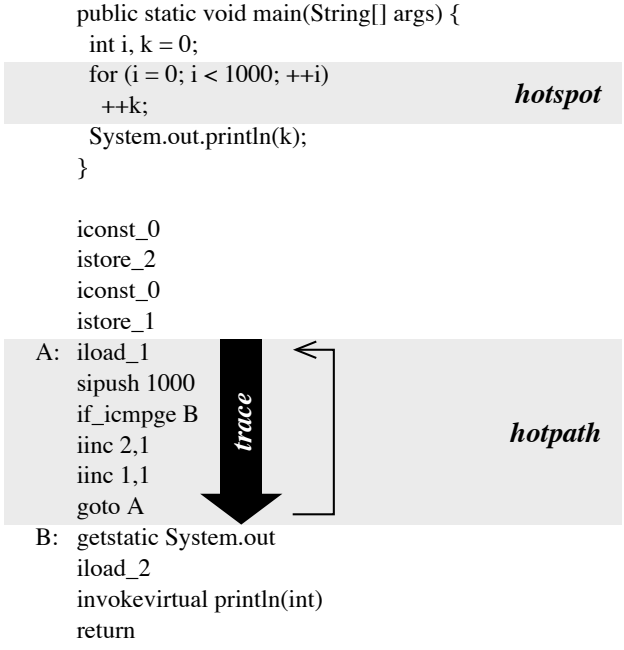

Figure 1. Recording bytecode traces. The source program (upper part of the figure) contains a frequently executed loop that we want to compile (a hotspot). The other parts of the program are executed infrequently and compiling them would likely not much benefit overall performance. The bottom part of the figure shows the resulting bytecode program. A frequently executed backward branch to A triggers the recording of a trace. The trace is complete when execution returns to the loop header. We also refer to recorded traces as "hotpaths", as they are the results of tracing hotspots.

not suitable for trace-based JIT compilation, the execution monitor disconnects itself from the virtual machine if no usable traces can be found and a second threshold is exceeded.

As branch instructions appear frequently in Java bytecode, it is essential to minimize the housekeeping overhead for keeping track of the invocation frequency of backward branches. In our first KVM-based implementation we used a relatively small (977 entries) closed hash table, consisting of 16-bit integer counters, for this purpose. Every time a backward-branch instruction is executed, the corresponding counter in the hash table is incremented. Collisions in the hash table are intentionally tolerated as their impact on code generation is relatively limited. Collisions can merely lead to overestimation of the "hotness" of a code region, triggering code generation for "cold" code. In the worst case this overestimation will cause a slight performance degradation as the VM might be unable to recover the compilation cost in terms of cycles spent on code optimization vs. cycles recovered from more efficient program execution.

Using a hash table is well suited for interpreters such as KVM, which interpret the native bytecode format as it exists in Java class files. JamVM, in contrast, translates the bytecode format into its own intermediate representation prior to execution. Amongst others, relative branch addresses are resolved to absolute target addresses and constant values are retrieved from the constant pool and stored directly in the intermediate representation. To further reduce the overhead of the branch monitoring approach described above, we directly embed the profiling information in the intermediate representation. This allows us to avoid the additional hash table lookup per branch instruction.

2.2 Recording Traces

Traces are recorded by manipulating the threaded code branch target stored in the intermediate representation of each instruction as it is executed by JamVM's direct threaded interpreter [4]. The entire interpreter of JamVM is located in a single C method. Each Java Virtual Machine Language (JVML) instruction is implemented as a code block that is preceded by a label. The address of this label is stored in the intermediate representation, and JamVM uses GCC's [12] computed goto statement to jump to the corresponding label of each instruction as it is executed.

To record a trace, the monitor manipulates the threaded code branch target in the instruction immediately following the first instruction of the trace to redirect execution to a recording code block. This special code block can be understood as a meta instruction that is (virtually) inserted after each executed instruction. The recording block records the current value of the program counter, the opcode of the executed instruction, and the value on top of the stack in a trace structure. Every time the recording block is invoked it reverses the change to the previously executed instruction that caused the recording block to be invoked and patches the next instruction label instead. Using this technique the recording block essentially chases the execution, always making sure that it is one step ahead of the program counter.

While our approach triples the number of hard to predict (and thus costly) indirect dispatches during trace recording, the overhead is essentially zero when the trace recorder is disconnected and is not recording traces. The only noteworthy overhead is incurred for JVML branch instructions, because we have to increment the invocation frequency counter and compare it to a threshold value. To eliminate this overhead we redirect branch instructions to code blocks that do no longer perform profiling once a second stop-loss threshold is exceeded. In this case we assume that no suitable traces could be extracted from the bytecode in question.

While recording traces, the direction (taken or not taken) of every conditional branch is recorded. These decision points in the original program become guard instructions in the recorded trace. At runtime, these guard instructions ensure that the control flow still follows the previously encountered path. Depending on whether the original branch was taken or not, a guard instruction is emitted that either checks for the branch condition itself or its complement.

A taken branch instruction that branches if the value is equal zero (`jeq`), for example, becomes a *guard if not equal zero* instruction (`gne`), because the compiled trace has to be aborted if the original assumption (value is equal zero) does not hold at the time the trace is executed. The same branch instruction (`jeq`) would be converted into a *guard if equal zero* instruction (`geq`) if it was not taken in the initial code, because now the compiled trace has to be guarded against the opposite condition.

Lookup table dispatches (JVML instructions `lookupswitch` and `tableswitch`) are used in Java to efficiently implement the `switch`/`case` construct of the Java high-level language. When encountering a lookup table dispatch instruction, the virtual machine determines the bytecode address where to continue execution based on a table of address and value pairs. We treat lookup table dispatches as if they were simple conditional branches that compare a value with a constant. When the trace is executed, we expect the same value to be encountered by the guard instruction, or the trace is terminated through a side exit.[1]

Optimizing virtual method invocations is a non-trivial task for traditional compilers. Since our compiler always follows the actually executed instruction stream, we only have to insert a guard instruction that checks that the predicted target method is still valid. For this, the guard instruction compares the actual type of the object the virtual method is invoked on with the type that was encountered when the trace was recorded. If the two types still match, execution of the trace continues with the already recorded and compiled instruction stream.

If any guard instruction fails at runtime, control is transfered back to the VM to resume execution at the bytecode level. For this, the mapping between machine code registers and Java stack and local variables has to be preserved. Our intermediate representation embeds this information in every guard instruction and the code generator creates side exit stubs that write back the values into the appropriate local variable locations.

Our initial implementation maintained mapping information for both, the Java operand stack as well as Java local variables. Profiling of our first prototype showed that the code generator never actually had to generate write-back code for stack locations. While in principle legal in JVML, the Java compiler does not generate basic blocks that have more than one predecessor and a non-empty stack at the entry point. Instead, the stack is always empty after branch instructions, except for some special constructs such as the "? :" operator. However, the latter can never be a loop header and thus will never trigger the recording of a trace and is thus irrelevant in this context. Based on this observation we removed support for traces that start with a non-empty stack or contain side-exits with a non-empty stack.

The last map is recorded at the very end of the trace where execution returns to the loop header (trace entry point). This map is special as it indicates which Java local variables are altered during each loop iteration. To ensure proper semantics, all values altered during a loop iteration have to be written back at all side exists, because the value could still be pending from a previous loop iteration (Figure 2).

2.3 Stop Conditions

Recording ends in two cases—either we decide to abort the recording due to certain circumstances, or we successfully finish the recording.

As we are only interested in the loop body itself, we define a number of abort conditions which terminate the recording of a trace. If an exception is thrown, for example, we immediately terminate recording, because exceptions are by definition rare events and by its very nature JIT compilation only focuses on frequently executed code areas. Similarly, we also abort tracing when we encounter a native method invocation. The rationale of this design decision is that native method invocations are already fairly expensive, no matter if compiled or interpreted, and we expect them to happen infrequently. Also, by not having to generate code for native code invocation, our backend compiler is simplified significantly.

When execution returns to the original loop header where a trace started, the recording of the trace ends and the newly recorded trace is sent to the JIT compiler for translation to optimized machine code.

3. Compiling Traces

Once a trace has been recorded, our trace compiler translates it to directly executable machine code. As mentioned previously, our compiler is specialized in compiling traces only and expects its input to be free of branch instructions except for a single, final, unconditional branch at the end of the trace, which jumps back to the loop header. The code generated for the trace is optimized with the assumption that the trace will be executed repeatedly (loop), until it is left through a side exit (guard instructions). In our system, traces do not have a generic exit point, as they always

[1] We will discuss in the next section how to merge several traces so we can compile lookup table dispatches that have several hot paths.

```
int i = 10, a;
while (i-- > 0) {
  if (b[i] == 0) [R1 => i]   /* add: R2 => a */
    break;
  a = 1;
} [ R1 => i, R2 => a ]
```

Figure 2. To ensure proper semantics, any values altered during a loop iteration have to be written back at all side exists, because the value could still be pending from a previous loop iteration. The map recorded for the *if* instruction, for example, initially only contains the necessary information to update the loop counter (*i*). The tail map at the end of the loop in contrast indicates that *a* has to be updated as well. This information is subsequently merged into all side exits to ensure that the *if* instruction not only updates *i* but also *a* in case of a side exit. This is necessary, because as we will discuss in the next section we will perform register coalescing on the tail map, ensuring that the next loop iteration finds all values in the corresponding register without actually writing back any values into the Java local variable locations. Thus each side exit is responsible for this step, even for values that were carried over from the previous iteration and have not been redefined yet in this iteration.

unconditionally branch back to the loop header once an iteration has been completed.

As we record traces across method calls, the input of our compiler is also free from method invocations, which significantly simplifies code generation.

The work of our trace compiler can be divided into three phases: a) stack deconstruction and transformation into SSA form, b) analysis, and c) code generation. Each phase is accomplished in a single sweep over the trace and thus executes roughly in linear time.

3.1 Stack Deconstruction

During the initial phase, we deconstruct the Java stack and replace references to stack locations and local variables with the index number of the instruction that defined the corresponding value. By doing so, we effectively also transform the trace into SSA form (Figure 3). For this, we track for each stack location and local variable the index of the instruction that defined the corresponding value, and update on the fly each operand reference. To import the initial context which consists of values stored on the Java stack and in local variables when the trace starting point is reached, the virtual machine automatically reports `read` pseudo instructions for all occupied local variable slots.

At the end of the loop, a renaming table and the tail map are used to decide which incoming context values are loop invariant. The renaming table maps local variables to their SSA names and is updated as each instruction is recorded and transformed on-the-fly into SSA. Each time a value is written into a local variable, a new SSA name is assigned and the renaming table is updated to reflect the new SSA name. For each use the renaming table is consulted to obtain the current SSA name for the corresponding local variable, which is recorded instead of the actual local variable index.

Each mapping in the tail map consists of the location where the value was defined and the local variable index where the value has to be written back to. A mapping that refers to a definition other than the `read` pseudo instruction we initially filled in for a given local variable slot indicates that the local variable context was redefined in the loop body. For these instructions the `read` pseudo instruction is transformed into a ϕ pseudo instruction.

ϕ pseudo instructions are similar to `read` pseudo instructions in that they read a value from the VM stack or a local variable before entering the loop. Along the loop edge, however, ϕ pseudo

```
    iconst_0
    istore_2
    iconst_0
    istore_1
A: iload_1
    sipush 1000
    if_icmpge B
    iinc 2,1
    iinc 1,1
    goto A
B: getstatic System.out
    iload 2
    invokevirtual println(int)
    return

0: read L1          (phi L1,4)
1: read L2          (phi L2,5)
2: sipush 1000
3: if_icmpge bail-out
4: iinc 1,1
5: iinc 0,1
6: goto 3
bail-out:
    return to VM
```

Figure 3. Phase 1: Stack deconstruction and transformation into SSA form. In our SSA-based IR, operands refer to the defining instruction in the trace instead of stack locations or local variables. Incoming values are represented through explicit *read* instructions, which in this case read local variables 1 and 2. In a single forward scan through the code, all operand references are updated by tracking the definition point for each stack location and local variable in a renaming table. The table initially contains the mappings (L1:0, L2:1), which causes the two *iinc* instructions to be updated with new operands. At the same time, the renaming table is updated to (L1:4, L2:5) because the *iinc* instructions redefined the values in local variables 1 and 2. At the end of the loop, we check which entries in the renaming table have been modified. All modified entries are *loop variables*, and their corresponding *read* instructions are each replaced with a ϕ instruction that reads the updated value along the loop edge, instead of the initial one. The remaining entries are flagged as *loop invariant*.

instructions return the value produced by a downstream instruction in the previous iteration. As the virtual machine has already inserted `read` instructions for all context values, we do not actually have to *insert* any ϕ pseudo instructions into the trace code. We merely transform the corresponding `read` pseudo instruction into a ϕ pseudo instruction.

It should be noted that we do not have to calculate a dominator tree to place ϕ nodes, because our traces contain at most one merge point, which is the trace entry where the loop edge and the entry edge meet. Furthermore, our initial phase also performs a loop invariant analysis—essentially for free.

3.2 Code Analysis in SSA form

Once we have obtained the SSA-based intermediate representation, we perform a series of data-flow analyses on the code (phase 2). Also these analyses occur during a single forward sweep over the trace. Among others, we propagate the loop invariant information to dependent instructions. In other words, all instructions that only depend on loop-invariant inputs are flagged as loop invariant and will be hoisted out of the loop body.

Initially, only the `read` and ϕ pseudo instructions inserted at the beginning of the trace are considered loop-invariant. `read` pseudo instructions are invariant, because they merely read values from the Java stack and local variables into temporary registers and the

Java stack and local variables are in fact not updated during trace execution. ϕ pseudo instructions, in contrast, are by definition loop variant, however, for our purposes we can still hoist them out of the loop because only the invariant part of their semantics results in actual code to be generated.

ϕ pseudo instructions have a different semantics, depending on whether they are executed during initial loop-entry or following a loop iteration. For the initial loop-entry, similarly to `read` pseudo instructions, ϕ pseudo instructions fetch context information from the Java stack or local variables. In case of a loop iteration, however, they carry over the value from the last iteration to the current one. The latter semantics, which is loop variant, we guarantee through our register allocation algorithm by assigning the same output register to ϕ pseudo instructions as the loop-carrying input register they are supposed to copy in case of a loop iteration. As no code generation is necessary to guarantee this property, we can safely move ϕ instructions out of the loop body.

Starting with the `read` and ϕ pseudo instructions, all instructions that only depend on loop-invariant inputs are hoisted out of the actual loop body, and are executed before the loop code.

To create more opportunity to invariant code motion, the trace recorder splits complex Java bytecode instructions into sub-instructions, some of which might be suitable to be hoisted out of the loop body or at least they can be shared between similar subsequent instructions. In fact, we did not have to introduce a large number of new instructions as most of the sub-instructions are simply JVML instructions with relaxed typing (we use `IADD` for address arithmetic, for example).

A common example for this principle are array access instructions (i.e. `IALOAD`). Each `IALOAD` instruction is emitted as a combination of a shift operation (`ISHL`) to transform the index into an appropriate offset depending on the array element size and an `IADD` instruction to generate an address from the offset and the object reference (array base address). Subsequent array accesses can then share parts or the entire address calculation. This approach also creates additional opportunity for efficient code generation by the backend, which can fold constant address calculations (i.e. constant index expression) and emit more compact code using fewer registers.

Integer modulo instructions (`IREM`) undergo a similar treatment and are emitted as a combination of an integer division, multiplication and subtraction since many architectures do not directly support modulo calculation. For architectures that do (i.e. x86) the backend can easily detect and compact the resulting code pattern, while architectures such as PowerPC that have no modulo machine instruction can optimize the resulting instruction stream, i.e. through common subexpression elimination.

In order to guarantee proper semantics for memory accesses, we only consider memory accesses to be loop invariant if no matching read/write exists with the same base type and field offset (or absolute address in case of static fields). While this approach is rather pessimistic and does not take any high-level type information into account, it still allows us to hoist most memory reads/writes out of the loop without having to invest a lot of resources in proper alias analysis.

We also perform common subexpression elimination (CSE) and flag redundant instructions, which the backend phase in turn will suppress during code generation. The rationale for not immediately eliminating redundant instructions is that at this point the register pressure is still unknown, and it might be beneficial to leave the redundant instruction in the code, instead of having to spill due to excessive register pressure.[2]

[2] This condition rarely occurs in case of the PowerPC architecture, but is more frequent on architectures with few registers such as Intel 386.

One could argue that by performing CSE and loop invariant code motion (hoisting) on a trace, we actually perform profile-guided partial redundancy elimination (PRE), because we perform CSE only along the hot path (trace), utilizing any partial redundancy, which regular CSE may not be able to realize on the entire graph. This observation underlines our claim that compilation of code traces is much simpler to facilitate than code generation in presence of code graphs.

The final register allocation and code generation phase is somewhat special in that it works bottom-up, starting at the last instruction in the trace working its way upstream. Register allocation and code generation are performed simultaneously. The register allocator is pre-initialized by assigning fixed register locations to all loop variables (ϕ instructions), which is necessary to guarantee that values generated during a loop iteration are properly carried over to the next iteration.

We try to coalesce both operands of ϕ pseudo instructions into the same hardware register to avoid unnecessary register moves at the end of the trace (loop). This is of course only legal if the live ranges of the of the two operands are disjunct. An overlap can for example occur when the loop code references the previous value of a loop variable after it has already been updated. A frequent cause for this are the unary operators $i{+}{+}$ and $i{-}{-}$ that increment/decrement a loop counter (in this case i), but return the previous value as result of the expression. To ensure proper semantics the register where the loop expects the incoming value to arrive in the next iteration can only be updated with the new value at the very end of the loop. Thus we have to ensure that the two instances are not coalesced into the same register but the value is explicitly transfered through a move instruction at the end of the loop.

As all other data-flow analyses, live range analysis is trivial on a linear sequence of instructions. We merely record the last use of each instruction during the initial iteration over the trace code. The last-use information is maintained as a simple pointer from each instruction to the last instruction in the trace that uses the value generated by it. This information is collected by continuously updating the last use pointer every time a value defined by an instruction appears as operand of another instruction. Intermediate uses are overwritten by subsequent uses, leaving the final use in the pointer field.

Whether the live ranges of the two operands of a ϕ pseudo instruction (the incoming value, and the value generated during each iteration) overlap, can easily be determined by comparing the last use information of the incoming value (context ϕ pseudo instruction) with the definition of the loop-generated value.

If the loop-generated value of the ϕ pseudo instruction is defined after the last use, the two values can be coalesced into the same register. Otherwise a move instruction is generated at the end of the loop (trace).

3.3 Code Generation

The code generator then starts to emit code, starting at the last instruction, and moving backwards. Correspondingly, the machine code is also generated backwards, which has the subtle advantage that the target address of conditional branches is always encountered before the actual branch instruction, eliminating the need to fix up target addresses in a second pass.

During code generation, we first try to perform constant folding for each operand reference of the instruction that is currently being compiled. Traditionally, constant folding is performed at compile time by executing the dynamic semantics of constant instructions. In a mixed-mode interpreter, this leads to significant code duplication between the compiler and the virtual machine, because both essentially perform the same task (executing bytecode instructions). In our system, instead of folding constants in the compiler, we rely

```
IADD:
  if constant(son[0]) && constant(son[1]) {
    value = OP(son[0],son[1])
    return
  }
  if constant(son[0])
    swap(son[0], son[1])
  if int_const(son[1]) && imm16(son[1])
    emit(addi, reg(son[0]), value(son[1]))
  else
    emit(add, reg(son[0]), reg(son[1]))
```

Figure 4. The backend iterates over the code bottom-up, which allows us to use simple pattern-matching for emitting specialized instruction forms. The generator function for the *IADD* bytecode instruction, for example, first checks whether its left operand is constant, and swaps its operands in that case. This is to ensure that if one operand is constant, this is always the right one, as expected by the *addi* machine instruction. Only operands that cannot be folded into a constant are assigned a register using the *reg* function. The corresponding code for the definition will be emitted later, once the code generator arrives at the defining instruction.

on the bytecode interpreter to record the value calculated by each instruction in the trace. The compiler merely tracks a bit whether operands are constant or not, and if all operands of an instruction are constant, the result previously recorded by the virtual machine is assumed to be the correct result value of the instruction.

The only JVML instruction that warrants special treatment is the `IINC` instruction. In contrast to all other JVML instructions it does not store the value it generates on top of the stack and thus our stack recording mechanism cannot capture the value. Instead, it directly pushes the new value into a local variable. Our recording code is aware of this irregularity and records the value of the local variable instead of the top of the stack for `IINC` instructions.

Only if constant folding fails we assign a register to the instruction defining the operand, which is the main reason why we perform code generation in reverse order. It guarantees that we always encounter all uses of a value before its actual definition. The code for the definition is only generated if not all uses could be resolved through constant folding and specialized instruction patterns. A common example is the `addi` instruction that allows to directly embed one operand as an immediate as long the value is within a certain range. If all uses of the value can be directly embedded in such a fashion, it is not necessary to actually generate that value at the definition site and no register has to be assigned to it. In case of forward code generation we would have to decide whether to emit the code for a definition before we know whether the value will actually be used or not, which in turn would require to analyze each use of the value.

As code generation continues, we will eventually reach the instruction we just assigned a register to, and generate the necessary code to produce a value into that register. Code is only generated for instructions that were previously assigned a register. If we are able to satisfy all uses statically through constant folding, the instruction will have no register allocation when we arrive at it, and it is considered dead.

Besides simplifying register allocation, bottom-up code generation also enables us to perform efficient pattern matching to emit specialized machine instruction forms (Figure 4). As described above, we try to fold all operands into constants, and if we realize that one operand of an *add* instruction is a constant, for example, we emit the specialized *addi* machine instruction (in case of a PowerPC backend).

```
02801008: cmpwi r2,1000
0280100c: bgel- 0x2801034
02801010: addi r3,r3,1
02801014: addi r2,r2,1
02801018: b 0x2801008
```

Figure 5. The final code emitted by our backend for the example introduced above. By assigning fixed registers to the ϕ instructions, we avoid any trailing move instruction at the end of the loop to correct the state of the register allocator before following the loop edge back to the loop header. The branch instruction in the original bytecode was compiled as a guard instruction, which exits the compiled code when execution diverges from the predicted path. We use the link register of the PowerPC to record the location where this bail-out happened, which allows the VM to unwind the execution state accordingly.

The resulting machine code for the example introduced in the previous section is shown in Figure 5.

3.4 Side Exits

The initial version of our compiler did not generate specific machine code for individual side exits. Guard instructions were translated to conditional *branch and link instructions* that all pointed to the same compensation code. Every time a guard instruction is executed (whether it passes or fails), the processor updates the link register with the current value of the program counter, which is essentially the address of the conditional branch instruction that was generated for the guard instruction. We can make such liberal use of the link register, because we don't invoke any methods from within compiled traces. If a guard instruction fails, execution is transfered to the side exit compensation code shared by all guard instructions. The compensation code first has to locate the appropriate register to local variable mapping information by translating back from the machine address recorded by the branch instruction to the address of the branch instruction in the intermediate representation that ultimately contains the pointer to the register mapping information.

Benchmarks have shown that this approach creates a significant overhead for traces that are frequently exited through side exists after only few iterations (or no complete iteration at all). We have thus changed our code generator to generate dedicated compensation code for each side exit.

Each side exit stub consists of a series of memory stores transferring values back from hardware registers into the appropriate local variable storage area.

If the side exit occurred while executing an inlined method, the compensation code has to generate additional stack frames to write back the current state.

In our implementation we actually write back the values first, and then create the new stack frames "around" the already written back stack and local variable information, because most temporary values are held in hardware registers, which in turn would have to be flushed to memory before we can execute the high-level functions of JamVM that are responsible for creating new stack frames. As a consequence we have to ensure that the garbage collector is not triggered while we are writing back the stack frames (or while we execute compiled code). We do so by holding the garbage collector lock at all times when compiled code is executed.

While in principle this blocks the garbage collector from reclaiming unused memory while compiled code is executed, and could cause a memory overflow, in practice this does not seem to be a significant limitation. Performance-critical code such as a loop rarely performs any memory allocations.

The main reason for seeing so few memory allocations inside loop bodies is the inherent cost of the memory allocation opera-

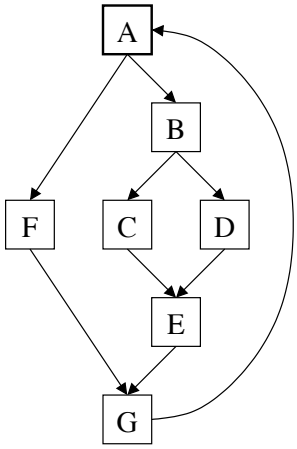

Figure 6. An example for a loop with more than one frequently executed path. If we supported just a single hotpath for each loop and then encountered A, F, G first and recorded a trace for it, all other combinations (A, B, C, E, G and A, B, D, E, G) would result in expensive bail-out operations in which the VM has to recover from failed guard instructions. Our secondary trace approach avoids this problem.

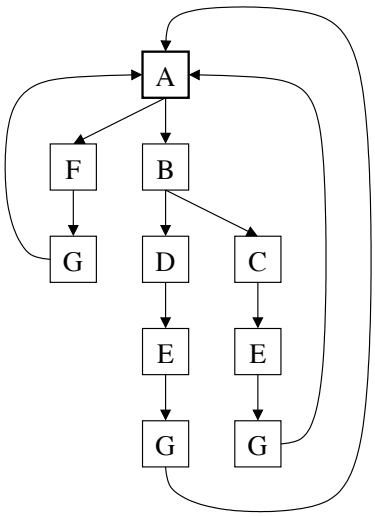

Figure 7. Recording secondary traces. Every time we exit a trace along a non-exception edge, we immediately start recording a new trace. The guard instruction that triggered the recording of the new trace is then patched to point to the newly recorded and compiled trace.

tion. If a loop calls into the memory allocator on every iteration, the overhead of running the memory allocator often by far outweighs any speedup that compilation of the rest of the loop code could achieve. Thus not compiling such loops in the first place is an acceptable initial heuristics.[3] Since the runtime of the loop is dominated by the runtime of the memory allocator, the performance loss will be small.

There is, however, a class of loops that would benefit from compilation despite the presence of memory allocation instructions. Since Java does not allow explicit stack allocation of temporary objects, regular heap allocation instructions are used. As these temporary objects are only live for a single iteration, we could actually bypass the allocator altogether and allocate temporary stack space instead. This would enable us to speed up the loop body without building up any unused heap blocks that the garbage collector would have to reclaim (which it can't since we block garbage collection while executing compiled code.)

We are currently working on a specialized linear-time escape analysis that will allow us to identify such temporary objects at runtime.

4. Trace Merging

Trace-based JIT compilation as presented in the previous two sections works well for very regular code. Unfortunately, code is rarely purely regular. Loops often do not contain a single dominant path, but several frequently executed paths (Figure 6). To achieve good performance in this more general case, we have to support multiple hot paths through a loop.

For this, similarly to Dynamo we record secondary (child) traces every time we exit a trace along a non-exception edge (Figure 7). The guard instruction is updated to jump to the child trace, instead of returning to the VM. The child trace shares the upstream code with its parent trace, and is compiled bottom-up just as its parent trace, with the register allocator initialized to the final state of the parent trace. This is necessary in order for the child trace to know in which register the parent trace stored values that were computed upstream.

If a child trace is left through a side exit, we resume recording but that trace will also be directly associated with the parent trace. In essence we lazily record all actually executed traces through the loop as we encounter them, and connect them to the single loop header of the parent trace.

In certain cases we also have to insert compensation code along the new edge to the child trace, for example in order to compute a value that was dead in the parent trace, but is now active in the child trace. These cases are easily recognizable, because a use refers to a definition in the parent trace that has no register allocation assigned to it.

It should be noted that this approach leads to a certain amount of code duplication, however, it also permits specialized code optimization along each path (Figure 8).

Invariant code motion in the presence of secondary traces also warrants special consideration. By aggressively hoisting computations out of the initially recorded primary trace, a large number of registers would be blocked for further use by secondary traces. To circumvent this problem, we selectively spill certain invariant values into memory when switching from parent to child traces.

An interesting effect can be observed in the presence of nested loops. As shown in Figure 9, in a nested loop construct the inner part of the loop tends to be "hotter" than the outer parts of the nested loop. Thus, the first loop header to be detected as such is likely B, because it is part of the inner loop. Once the trace $B - C$ has been recorded and is executed, a new trace is recorded as soon C exits to D instead of following the loop edge. We record a new child trace $D - A$ and reconnect it to B, which we consider the loop header. The fact that we only consider B as a loop header as far as optimization is concerned means that the primary trace $B-C$ will get more resources allocated than $C-D-A$. However, considering that the inner loop is likely the hotter path, this is actually exactly what we want. Effectively, we have turned the loop inside out, giving maximum consideration to the hottest loop region, and slightly disadvantaging the outer loop parts.

[3] Our compiler currently also rejects such loops since memory allocation requires a callback into the virtual machine and is thus effectively a native method invocation, which in our prototype abort trace compilation.

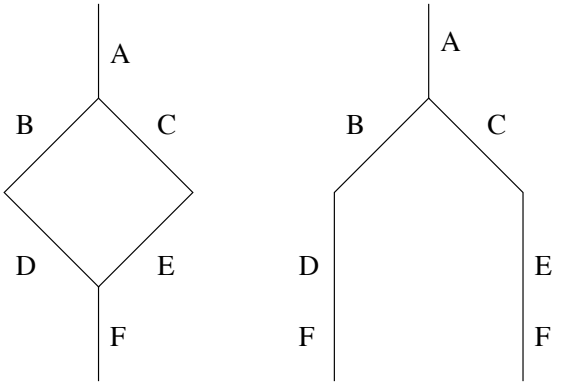

Figure 8. Code duplication resulting from trace-based compilation. A compiler designed to compiler whole control-flow graphs is able to generate shared code for F, while our trace compiler has to generate versions of F separately for each trace. While this is a disadvantage from a code density perspective, it does allow our compiler to individually optimize each version of F, according to its specific pre-context ($B - D$ and $C - E$).

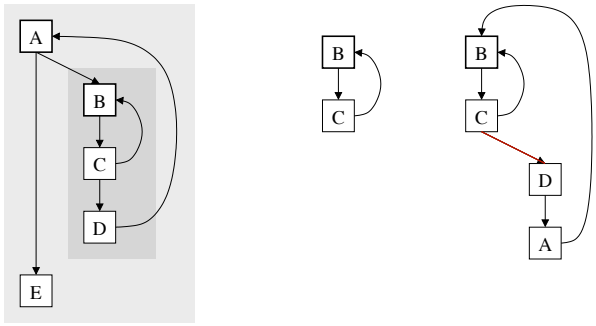

Figure 9. Handling of nested loops. As inner loops tend to be executed more frequently than the outer parts of the loop, the trace through the inner loop is recognized as the primary trace. The loop header of the inner loop B becomes the sole loop header. When C branches to D instead of looping back to B, we start recording a child trace and connect it back to A once the inner loop is entered again.

Once we support trace merging, the question arises how this approach is still different from compilation of whole control-flow graphs. The two most significant advantages over traditional compilation are that paths are split and optimized individually (downstream), and that relevant paths are compiled and optimized only, whereas compilers designed to compile whole control-flow graphs often compile entire ”hot” methods, even though only parts of the method are in fact hot, while certain other parts are rarely or never executed.

5. Benchmarks

We have implemented a prototype of the trace-based HotpathVM JIT compiler. While initially our prototype was built based on Sun's KVM [24, 23] virtual machine, the current implementation we are reporting on in this section has been ported to JamVM [17]. Similarly to KVM, JamVM is a small virtual machine suitable for embedded devices. In contrast to traditional JIT compilers that heavily depend on the internals of the virtual machine they are hosted by, and are deeply interwoven with virtual machine code, for both, KVM and JamVM, we only had to touch approximately 20 lines of the VM source code to add our compiler to the VM. The most relevant change was to invoke our trace recorder every time a branch instruction is encountered, to decide whether to start a new recording, or to execute an already existing trace. The remainder of the changes signal trace abort conditions to the JIT compiler, for example when exceptions are raised or a native method is executed.

The JIT compiler itself consists of roughly 1800 lines of C code, from which 900 lines are used by the frontend (phase 1 and 2), and 900 by the backend (phase 3, PowerPC). Our initial implementation that performed manual constant folding instead of recording constant values through the VM stack was 700 lines large, which is almost a 50% savings over the current size of the JIT compiler.

Compiled to native code, our JIT compiler has a footprint of approximately 50 kBytes, which is noteworthy for a compiler that implements a series of aggressive optimizations that were previously inaccessible to embedded mobile code frameworks. For comparison, JamVM alone consists of over 20,000 lines of C code (180 kBytes of PowerPC native code) [13].

Additionally to the static size of the code, during compilation our JIT compiler requires approximately 48 bytes of heap per instruction in the trace. The space is freed up in between compiler runs. To cache compiled traces, we use a 64 kBytes code buffer, bringing our overall minimum memory consumption to approximately 128 kBytes.

We have measured the performance of our Hotpath Virtual Machine for a set of benchmark programs from the SciMark2 [19] and Java Grande [18] benchmark suites. The set contains programs that are ideal for trace compilation (SOR, LU) but also programs that can be considered the worst-case scenario for trace compilation (NumericSort and MonteCarlo), because they contain branches without clear preferred branch direction (taken and not taken are both equally likely).

The performance was compared to the standard interpreting Java VM, JamVM 1.3.3, which is one of the fastest purely interpreting Java virtual machines, and finally Sun's HotSpot just-in-time compiler (Figure 10). Ideally, we would have preferred to compare our trace-based compiler to a commercial just-in-time compiler such as Sun's CLDC HotSpot VM [15, 20] as well. Unfortunately, however, Sun was unable to license CLDC Hotspot to a non-commercial entity such as the University of California in either binary or source code form.

As expected, our JIT compiler performs well on highly regular and sequential programs such as LU, SOR, and Linpack. FFT and SparseCompRow still yield a speedup of factor two over interpretation but suffer from the lack of long traces that could be exploited. We are unable to record meaningful traces for MonteCarlo and NumericSort because our current prototype does not support trace merging across (inlined) method invocations, which both benchmarks would require.

Our compiler is efficient for pure Java, but fails to optimize a single trace in the presence of native method invocations (because we abort trace recording when encountering a native method invocation). This points to a significant problem regarding the current Java libraries. Because Java interpreters are generally relatively slow, some performance critical methods (such as `arraycopy`) are implemented in native C. This ”optimization” improves performance in case of interpretation, but inhibits our compiler from performing actual JIT compilation, resulting in very poor performance. To improve performance, we will likely have to reimplement all relevant parts of the Java library that do not rely on unsafe language constructs in pure Java. This would be particularly beneficial in an embedded context since Java code is often

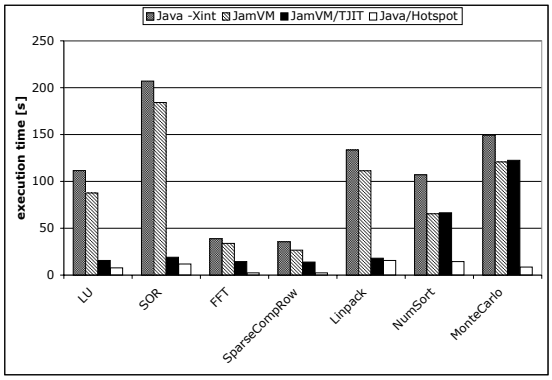

Figure 10. Execution time for a set of benchmark programs from the SciMark2 and Java Grande benchmark suites, executed by for pure interpretation with the standard Java VM (`java -Xint`), JamVM 1.3.3 (`jamvm`), our trace-based JIT compiler (`TJIT`), and finally Sun's Hotspot VM (`Hotspot`). As expected, our JIT compiler performs well on highly regular and sequential programs such as LU, SOR, and Linpack. FFT and SparseCompRow still yield a speedup of factor two over interpretation but suffer from the lack of long traces that could be exploited. We are unable to record meaningful traces for MonteCarlo and NumericSort because our current prototype does not support trace merging across (inlined) method invocations, which both benchmarks would require.

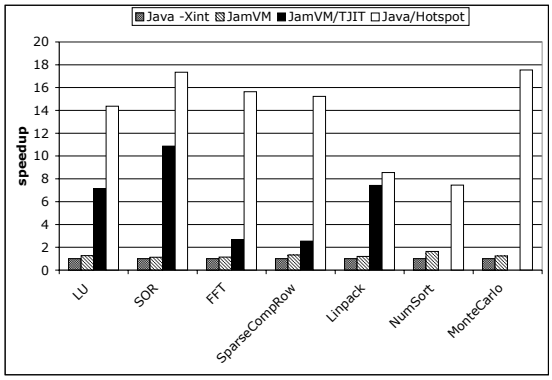

Figure 11. Speedup of trace-based JIT compilation over pure interpretation. For highly regular programs our compiler achieves a speedup of 7-11 over a fast interpreter. For code with significant side exists the speedup drops to factor 3. For highly irregular code our JIT compiler is unable to extract useful traces and does not improve execution time.

significantly more compact than equivalent pre-compiled native code.

Currently, we do not plan on adding support for inlining native methods to our compiler. Previous work in this area [22] has shown that efficient inlining of native code blocks requires translating the machine code instructions inside the native method into the higher-level intermediate representation used internally by the JIT compiler. In essence this amounts to decompiling the native code, so that it can be (re-)optimized, specialized, and interwoven with the surrounding Java code. It is unlikely that this analysis effort would pay off in an embedded context.

6. Related Work

Our research is related to a number of existing works and projects. The main inspiration for our trace selection and recording mechanism was Dynamo [3]. Our system differs from Dynamo, because we record bytecode traces, and then compile them to native machine code, whereas Dynamo performs dynamic optimization on native machine code, emitting optimized native machine code. Our actual trace recording algorithms also differ from Dynamo, because we have more high-level information available than Dynamo, which only sees native machine code instructions.

Our approach of bottom-up code generation is closely related to Burg [11, 10] that extends work by [1, 2]. Just like BEG [8] and Twig [1], Burg works on tree grammars and generates a tree parser, which in turn makes two passes over the tree for code generation. The main difference for all these approaches with respect to our work is that they all need a structured IR to work, while our input is a mere linear sequence of instructions (that is read backwards). Our approach also combines code generation, constant folding, and register allocation, and all three steps are performed in interlock in a single pass over the code. We can perform all three steps in a single pass because we iterate backwards over the code, guaranteeing that we will always see all uses before the actual definition, which we utilize for on-the-fly dead code analysis and register allocation.

The work by Chang et al. [5] on superblock formation and optimization is closely related to our loop compilation mechanism. Similar to Chang, we transform loop code into suitable single-entry traces through re-tracing and duplicating the loop body along each possible execution path. Our work differs from Chang in that we do so at runtime and lazily: we only add additional child traces to the loop construct as we encounter them dynamically at runtime.

A number of existing virtual machines target the same mobile domain our work aims at. Sun has produced its own JIT compiler for the Kilobyte Virtual Machine, called KJIT [21]. KJIT is a lightweight dynamic compiler. Similarly to our work, it only compiles a subset of bytecodes to reduce the size of the compiler. In contrast to our JIT compiler, however, 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 and we have not been able to use it for comparative benchmarks.

Sun's current implementation of an embedded JIT compiler is called CLDC Hotspot VM [15, 20]. 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.

Other VM's for the embedded domain include E-Bunny [7] and Jeode EVM [16]. E-bunny is a simple, non-optimizing compiler for x86, that uses stack-based code generation, which is very fast as far as compile time is concerned, but yields poor code quality in comparison to optimizing compilers. Jeode is an optimizing compiler that uses a simplified form of dynamic compilation. Unfortunately, little is known about its internals.

7. Summary and Conclusion

We have presented an approach to just-in-time compilation that records frequently executed code traces and compiles them to executable native code. Instead of supporting generalized code graphs, our compiler is highly specialized to support only instruction traces, which greatly simplifies the algorithms involved in compilation and code optimization. Whereas traditional systems often compile entire methods, regardless of whether all parts of the method are hot or not, our approach focuses solely on hot code paths and leaves cold regions to the interpreter.

Furthermore, our approach opens up additional optimization potential by not allowing irrelevant code parts (such as rarely executed exception edges) to interfere with the code quality of hot areas. Additionally, compiler analyses and optimizations that are complex and costly when applied to graphs of basic blocks are trivial to implement when the input is restricted to traces.

To support irregular control flow, our system implements a trace merging algorithm that compiles secondary traces when an initial trace is exited along a non-exception edge. Secondary traces are then directly attached to the parent trace and future executions along this new path will be nearly as efficient as continuing along the edges of the original primary trace.

We plan to port our JIT compiler to additional target architectures. Due to the simplicity and small size of the compiler (which is a direct result of limiting the code input to traces), we expect to be able to add additional backends without much effort. A StrongARM backend would be particularly interesting as it would allow a direct comparison with existing embedded virtual machines such as Jeode EVM.

Preliminary performance results from our prototype implementation are highly encouraging, and position our approach favorably among the price/performance trade-offs made by embedded just-in-time compilers. As a simple and effective technique for reducing the size of optimizing just-in-time compilers, and thereby the trusted code base of critical systems, our approach may even be applicable outside of the embedded systems space.

8. Acknowledgment

This research effort was partially funded by the State of California and Microsoft Research under the Microeletronics Innovation and Computer Research Opportunities (MICRO) program, the Bavaria California Technology Center (BacaTec), and the National Science Foundation (NSF) under grants TC-0209163 and ITR-0205712. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation or any other agency of the U.S. Government.

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.

[2] A. Appel. Concise Specification of Locally Optimal Code Generators. Technical Report CS-TR-080-87, Princeton University, 1987.

[3] V. Bala, E. Duesterwald, and S. Banerjia. *Transparent Dynamic Optimization: The Design and Implementation of Dynamo*. Technical Report HPL-1999-78, Hewlett Packard Laboratories, June 1999.

[4] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[5] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software—Practice and Experience*, 21(12):1301–1321, December 1991.

[6] 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.

[7] M. Debbabi, A. Mourad, and N. Tawbi. Armed E-Bunny: a selective dynamic compiler for embedded java virtual machine targeting ARM processors. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 874–878, New York, NY, USA, 2005. ACM Press.

[8] H. Emmelmann, F.-W. Schröer, and L. Landwehr. BEG: A Generator for Efficient Back Ends. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 227–237, 1989.

[9] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[10] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[11] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG: Fast Optimal Instruction Selection And Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, Apr. 1992.

[12] Free Software Foundation. GNU c compiler, Dec. 2005. `http://gcc.gnu.org`.

[13] A. Gal. KVM [24] compiled for PowerPC/Linux with GNU C Compiler 3.2, April 2005.

[14] A. Gal. Measured on a Sharp Zaurus PDA, 206 MHz SA-1110, 64MB RAM, using a modified subset of SpecJVM98, July 2005.

[15] CLDC HotSpot Implementation Virtual Machine, available at `http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf`, Feb. 2005.

[16] Insignia Solutions. Jeode Platform: Java for Resource-constrained Devices, White Paper, 2002.

[17] R. Lougher. JamVM Virtual Machine. `http://jamvm.sf.net/`, Nov. 2005.

[18] 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.

[19] R. Pozo and B. Miller. SciMark2 `http://math.nist.gov/scimark2`, Mar. 2004.

[20] K. Schmid. Jbed Micro Edition CLDC and Jbed Profile for MID. Technical report, Esmertec AG, Dubendorf, Switzerland, 2002.

[21] N. Shaylor. A Just-in-Time Compiler for Memory-Constrained Low-Power Devices. In *Proceedings of the 2nd Java and Virtual Machine Research and Technology Symposium*, pages 119–126, Berkeley, CA, USA, 2002. USENIX Association.

[22] L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining java native calls at runtime. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 121–131, New York, NY, USA, 2005. ACM Press.

[23] SUN J2ME's Homepage. `http://java.sun.com/j2me`.

[24] Sun Microsystems. J2ME Building Blocks for Mobile Devices, White Paper on KVM and the Connected, Limited Device Configuration `http://java.sun.com/products/cldc/wp/KVMwp.pdf`, May 2000.