The Jalapeño Dynamic Optimizing Compiler for Java^{TM}*

Michael G. Burke Jong-Deok Choi Stephen Fink David Grove Michael Hind
Vivek Sarkar Mauricio J. Serrano V. C. Sreedhar Harini Srinivasan
John Whaley
IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

Abstract

The Jalapeño Dynamic Optimizing Compiler is a key component of the Jalapeño Virtual Machine, a new Java[1] Virtual Machine (JVM) designed to support efficient and scalable execution of Java applications on SMP server machines. This paper describes the design of the Jalapeño Optimizing Compiler, and the implementation results that we have obtained thus far. To the best of our knowledge, this is the first dynamic optimizing compiler for Java that is being used in a JVM with a compile-only approach to program execution.

1 Introduction

This paper describes the Jalapeño Optimizing Compiler, a key component of the Jalapeño Virtual Machine, a new JVM being built at IBM Research. A distinguishing feature of the Jalapeño JVM is that it takes a *compile-only* approach to program execution. Instead of providing both an interpreter and a JIT compiler as in other JVMs, bytecodes are always translated to machine code before they are executed. Jalapeño has three different compilers to provide such translation: an optimizing compiler for computationally intensive methods (which is the subject of this paper), a "quick" compiler that performs a low level of code optimization (primarily register allocation), and a "baseline" compiler that mimics the stack machine of the JVM specification document [29]. The compile-only approach makes it easier to mix execution of unoptimized and optimized compiled methods in the Jalapeño JVM, compared to mixing interpreted execution and JIT-compiled execution as in other JVMs.

A primary goal of the Jalapeño JVM is to deliver high performance and scalability of Java applications on SMP server machines. Some previous high performance implementations of Java (e.g., [9, 25, 22, 18]) have relied on static compilation, and have therefore disallowed certain features such as dynamic class loading. In contrast, the Jalapeño JVM supports all features of Java [29], and the Jalapeño Optimizing Compiler is a fully integrated dynamic compiler[2] in the Jalapeño JVM.

The Jalapeño project was initiated in December 1997 at the IBM T. J. Watson Research Center and is still work-in-progress. This paper describes the design of the Jalapeño Optimizing Compiler and the implementation results that we have obtained thus far. To the best of our knowledge, this is the first dynamic optimizing compiler for Java that is being used in a JVM with a compile-only approach to program execution.

The rest of the paper is organized as follows. Section 2 provides the context for this work by describing key features of the Jalapeño Virtual Machine. Section 3 outlines the high-level structure of the Jalapeño Optimizing Compiler and how it is invoked within the Jalapeño Virtual Machine. Section 4 describes the intermediate representation (IR) used in the Jalapeño Optimizing Compiler. Sections 5 and 6 describe the "front-end" and "back-end" respectively of the Jalapeño Optimizing Compiler; the front-end describes a (mostly) single-pass translation of Java bytecodes to an optimized high-level IR (HIR), and the back-end describes how HIR is lowered and translated into optimized machine code accompanied by exception tables and GC stack-maps. Section 7 summarizes our framework for efficient flow-insensitive optimizations for single-assignment variables. Section 8 describes our framework for inlining method calls. Section 9 presents performance results obtained from the current im-

[*] *To appear in the 1999 ACM Java Grande Conference, San Francisco, California, June 12–14, 1999.*

[1] Trademark or registered trademark of Sun Microsystems, Inc.

[2] Though dynamic compilation is the default mode for the Jalapeño Optimizing Compiler, the same infrastructure can be used to support a hybrid of static and dynamic compilation, as discussed in Section 2.

plementation of the Jalapeño Optimizing Compiler (as of March 1999). Section 10 describes two interprocedural optimizations that are in progress as extensions to the current implementation — interprocedural optimization of register saves and restores, and interprocedural escape analysis. Finally, Section 11 discusses related work and Section 12 contains our conclusions.

2 The Jalapeño Virtual Machine

The subsystems of the Jalapeño JVM include a dynamic class loader, dynamic linker, object allocator, garbage collector, thread scheduler, profiler (on-line measurements system), three dynamic compilers, and support for other runtime features, such as exception handling and type testing. Among the three dynamic compilers, the baseline compiler was implemented first. It is used to validate the other compilers, for debugging, and as the default compiler until the quick compiler is fully functional. The class loader supports dynamic linking via backpatching for classes that were loaded after compilation.

Memory management in the Jalapeño JVM consists of an object allocator and a garbage collector. The Jalapeño JVM supports type-accurate garbage collection. With a view to future experimentation to determine which garbage collection algorithm will be best suited for SMP execution of multithreaded Java programs, the Jalapeño JVM contains a variety of type-accurate garbage collectors (generational and non-generational, copying and non-copying) [24].

In the Jalapeño JVM, each object has a two-word header: a pointer to a *type information block*, and a *status word* for hashing, locking, and garbage collection. Since threads in Java are objects, the Jalapeño JVM creates a distinct object for each Java thread. One of the fields of this thread object holds a reference to the thread's stack, which contains a contiguous sequence of variable-size stack frames, one per method invocation. These stack frames are chained together by "dynamic links".

Another distinguishing feature of the Jalapeño JVM is that all its subsystems (including the compilers, run-time routines, and garbage collector) are implemented in Java and run alongside the Java application. Although it is written in Java, the Jalapeño JVM is self-bootstrapping; i.e., it does not need to run on top of another JVM. One of the many advantages of a pure Java implementation is that we can dynamically self-optimize the Jalapeño JVM.

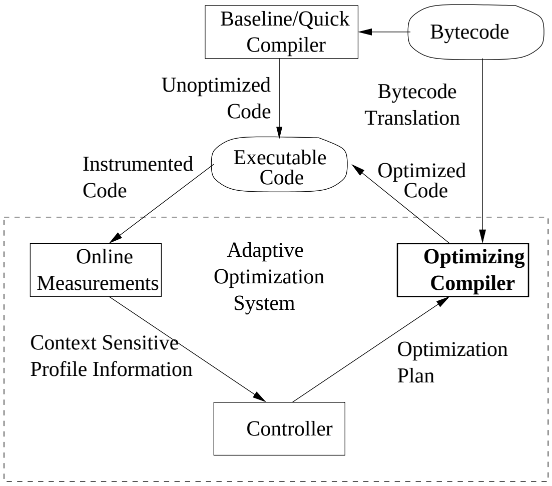


Figure 1: Context for Jalapeño Optimizing Compiler

3 Structure of the Jalapeño Optimizing Compiler

The Jalapeño Optimizing Compiler is *adaptive* and *dynamic*. It is invoked on an automatically selected set of methods while an application is running. The goal of the Jalapeño Optimizing Compiler is to generate the best possible code for the selected methods for a given compile-time budget. In addition, its optimizations must deliver significant performance improvements while correctly preserving Java semantics with respect to exceptions, garbage collection, and threads. Reducing the cost of synchronization and other thread primitives is especially important for achieving scalable performance on SMP servers. Finally, it should be possible to retarget the Jalapeño Optimizing Compiler to a variety of hardware platforms. Building a dynamic optimizing compiler that achieves all these goals is a major challenge.

Figure 1 shows the overall design for how the Jalapeño Optimizing Compiler is used in the Jalapeño Virtual Machine. The Optimizing Compiler is the key component of Jalapeño's Adaptive Optimization System, which also includes an On-Line Measurements (OLM) subsystem and a Controller subsystem. (The OLM and Controller subsystems are currently under development.) The OLM system is designed to monitor the performance of individual methods in the application by using software sampling and profiling techniques combined with a collection of hardware performance monitor information, and to maintain context-sensitive profile information for method calls in a Calling Context Graph (CCG) similar to the Calling Context Tree introduced in [3]. The Controller subsystem will be invoked when the OLM subsystem detects that a certain performance threshold is reached. The controller uses the CCG and its associated profiling information to build an "optimization

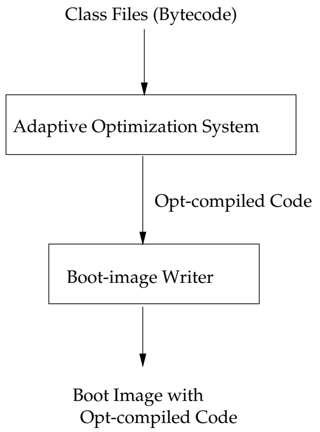

Class Files (Bytecode)
Adaptive Optimization System
Opt-compiled Code
Boot-image Writer
Boot Image with Opt-compiled Code

Figure 2: The Jalapeño Optimizing Compiler as a static compiler

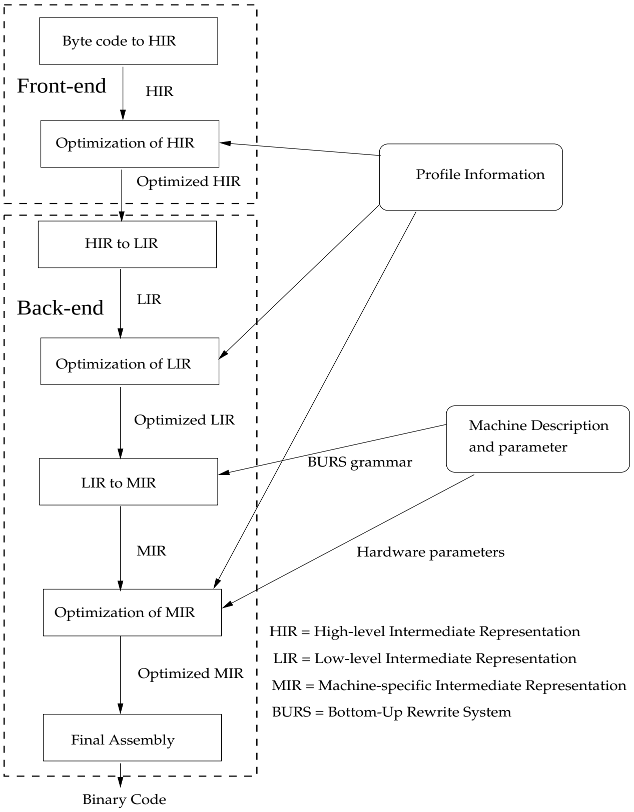


Figure 3: Internal structure of Jalapeño Optimizing Compiler for compiling a single method

plan" that describes which methods the optimizing compiler should compile and with what optimization levels. The OLM subsystem will continue monitoring individual methods, including those already optimized by the optimizing compiler, to trigger further optimization passes as needed.

The optimizing compiler can also work as a static compiler by saving the generated binary code in a file for later execution. In fact, as part of Jalapeño's bootstrapping procedure, the optimizing compiler currently compiles selected methods from the Jalapeño JVM, and stores the resulting binary code in a "boot image". Similarly, the optimizing compiler could also compile selected methods from a user application and store them in a custom boot image tailored to the application. When doing so, the optimizing compiler would essentially function as a static compiler (as shown in Figure 2).

When the Jalapeño Optimizing Compiler functions as a pure dynamic compiler, it must generate the best possible code for a given compile-time budget. The compile-time budget is less important when the Jalapeño Optimizing Compiler functions as a static compiler or as a static bytecode-to-bytecode optimizer.[3]

Figure 3 shows the internal structure of the Jalapeño Optimizing Compiler when compiling a single method. At the highest level, the Jalapeño Optimizing Compiler consists of an optimizer *front-end* (described in Section 5) and an optimizer *back-end* (described in Section 6).

4 Intermediate Representation

This section outlines some essential features of the register-based intermediate representation (IR) used by the Jalapeño Optimizing Compiler. Compared to a stack-based IR, a register-based IR better matches the load-store architectures that we target. Thus, it enables more effective machine-specific optimizations, as well as greater flexibility in code motion and code transformation.

An instruction in our IR is an n-tuple (a generalization of quadruples and three-address code [1]) consisting of an *operator* and some number of *operands*. The most common type of operand is the *register* operand, which represents a symbolic register. There are also operands to represent constants, branch targets, method signatures, types, etc. A key difference between the Jalapeño HIR and Java bytecodes is the addition of separate operators to implement explicit checks for several common run-time exceptions, e.g., NULL_CHECK and BOUNDS_CHECK operators to test for null pointer dereferences and out-of-bounds array accesses respectively. These additional operators facilitate optimization.

Instructions are grouped into basic blocks, delimited in the instruction stream by LABEL and END_BBLOCK instructions. In our IR, method calls and potential trap sites do not end basic blocks. The basic blocks and the control

[3] Another project at IBM is using the front-end of the Jalapeño Optimizing Compiler as the foundation for building a static bytecode optimizer.

flow graph (CFG) of the procedure are constructed as a byproduct of BC2IR's generating the HIR instruction stream (see Section 5.1). The IR also includes space for the caching of optional auxiliary information, such as reaching definition sets, a data dependence graph, or an encoding of the procedure's loop nesting structure.

5 Jalapeño Optimizing Compiler Front-end

The front-end contains two parts: (1) the BC2IR algorithm that translates bytecodes to HIR and performs on-the-fly optimizations during the translation, and (2) additional optimizations performed on the HIR after BC2IR. This section contains a description of the BC2IR algorithm with bytecode to HIR translation outlined in Section 5.1, and on-the-fly optimizations summarized in Section 5.2. Examples of optimizations that are performed on the HIR can be found later in Section 7.

5.1 The BC2IR Algorithm

Figure 4 shows an overview of the BC2IR algorithm. The algorithm contains two parts: (1) the *Main Loop* that selects a basic block (BB) from a worklist, called the *basic block set* (BBSet) and (2) the *Abstract Interpretation Loop* that interprets bytecodes within a BB. The algorithm maintains a *symbolic state* during the translation process, which corresponds to abstract values of stack operands and local variables.[4] The *initial state* of a BB is the symbolic state of the machine at the start of the BB. Initially, certain candidate BBs that can be put in the BBSet are identified (for example, the BB beginning at bytecode 0 with an empty initial stack or exception handler blocks).

After the initial BBSet is identified, BC2IR enters the main loop, and selects a BB such that its initial state is fully known and no HIR has been generated for it. For each BB, the bytecode in it is abstractly interpreted, the current state is updated, and new BBs may be generated. The BBs thus generated will be added to the BBSet. During this phase the compiler constructs the CFG and performs other analyses and optimizations. The abstract interpretation process essentially interprets the bytecodes based on the Java bytecode specification defined in [29].

Bytecodes that pass Java verification have an important property that we exploit: "When there are two execution paths into the same point, they must arrive there with exactly the same type state" [4]. At a control flow join, the values of stack operands may differ on different incoming edges, but the types of these operands must match. An element-wise meet operation is used on the stack operands to update the symbolic state [38]. When a backward branch whose target is the middle of an already-generated basic block is encountered, the basic block is split at that point. If the stack is not empty at the start of the split BB, the basic block must be regenerated because the initial states may be incorrect. The initial state of a BB may also be incorrect due to as-of-yet-unseen control flow joins. To minimize the number of a times HIR is generated for a BB a simple greedy algorithm is used for selecting BBs in the main loop. When selecting a BB to generate the HIR, the BB with the lowest starting bytecode index is chosen. This simple heuristic relies on the fact that, except for loops, all control-flow constructs are generated in topological order, and that the control flow graph is reducible. Surprisingly, for programs compiled with current Java compilers, the greedy algorithm can always find the optimal ordering in practice.[5]

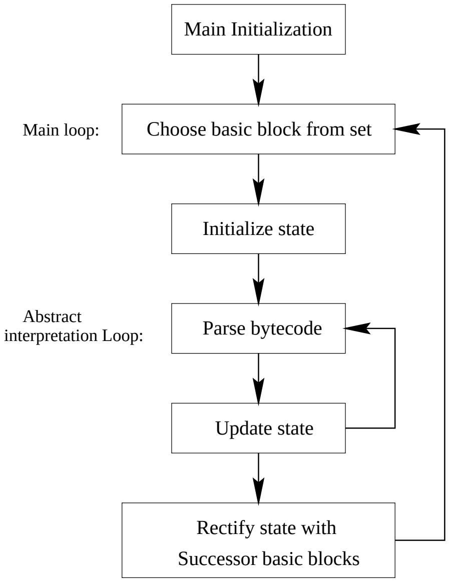

Main Initialization
Main loop:
Choose basic block from set
Initialize state
Abstract interpretation Loop:
Parse bytecode
Update state
Rectify state with Successor basic blocks

Figure 4: Overview of BC2IR algorithm

```
class t1 {
    static float foo(A a, B b, float c1, float c3)
    {
      float c2 = c1/c3;
      return(c1*a.f1 + c2*a.f2 + c3*b.f1);
    }
}
```

Figure 5: An example Java program

[4] Abstract values of local variables are needed during on-the-fly optimizations.

[5] The optimal order for basic block generation that minimizes number of regeneration is a topological order (ignoring the back edges). However, because BC2IR computes the control flow graph in the same pass, it cannot compute the optimal order a priori.

Example: Figure 5 shows an example Java source program of `class t1`, and Figure 6 shows the HIR for method `foo` of the example. The number on the first column of each HIR instruction is the index of the bytecode from which the instruction is generated. Before compiling `class t1`, we compiled and loaded `class B`, but not `class A`. As a result, the HIR instructions for accessing fields of `class A`, bytecode indices 7 and 14 in Figure 6, are **`getfield_unresolved`**, while the HIR instruction accessing a field of `class B`, bytecode index 21, is a regular **`getfield`** instruction.

Also notice that there is only one **`null_check`** instruction that covers both **`getfield_unresolved`** instructions; this is a result of BC2IR's on-the-fly optimizations.

```
0   LABEL0              B0@0
2   float_div           l4(float)  = l2(float), l3(float)
7   null_check          l0(A, NonNull)
7   getfield_unresolved t5(float)  = l0(A), < A.f1>
10  float_mul           t6(float)  = l2(float), t5(float)
14  getfield_unresolved t7(float)  = l0(A, NonNull), < A.f2>
17  float_mul           t8(float)  = l4(float), t7(float)
18  float_add           t9(float)  = t6(float), t8(float)
21  null_check          l1(B, NonNull)
21  getfield            t10(float) = l1(B), < B.f1>
24  float_mul           t11(float) = l3(float), t10(float)
25  float_add           t12(float) = t9(float), t11(float)
26  float_return        t12(float)
    END_BBLOCK          B0@0
```

Figure 6: HIR of method `foo()`. *l* and *t* are virtual registers for local variables and temporary operands, respectively.

5.2 On-the-Fly Analyses and Optimizations

To illustrate our approach to on-the-fly optimizations we consider copy propagation as an example. Java bytecode often contains sequences that perform a calculation and store the result into a local variable (see Figure 7). A simple copy propagation can eliminate most of the unnecessary temporaries. When storing from a temporary into a local variable, BC2IR inspects the most recently generated instruction. If its result is the same temporary, the instruction is modified to write the value directly to the local variable instead.

Other optimizations such as constant propagation, dead code elimination, register renaming for local variables, method inlining, etc. are performed during the translation process. Further details are provided in [38].

```
Java bytecode   Generated IR          Generated IR
                (optimization off)   (optimization on)
-------------   ------------------   -----------------
iload x         INT_ADD tint, xint, 5   INT_ADD yint, xint, 5
iconst 5        INT_MOVE yint, tint
iadd
istore y
```

Figure 7: Example of limited copy propagation and dead code elimination

6 Jalapeño Optimizing Compiler Back-end

In this section, we describe the *back-end* of the Jalapeño Optimizing Compiler.

6.1 Lowering of the IR

After high-level analyses and optimizations are performed, HIR is lowered to low-level IR (LIR). In contrast to HIR, the LIR expands instructions into operations that are specific to the Jalapeño JVM implementation, such as object layouts or parameter-passing mechanisms of the Jalapeño JVM. For example, operations in HIR to invoke methods of an object or of a class consist of a single instruction, closely matching the corresponding bytecode instructions such as **`invokevirtual/invokestatic`**. These single-instruction HIR operations are lowered (i.e., converted) into multiple-instruction LIR operations that invoke the methods based on the virtual-function-table layout. These multiple LIR operations expose more opportunities for low-level optimizations.

```
0   LABEL0              B0@0
2   float_div           l4(float)  = l2(float), l3(float)   (n1)
7   null_check          l0(A, NonNull)                       (n2)
7   getfield_unresolved t5(float)  = l0(A), <A.f1>           (n3)
10  float_mul           t6(float)  = l2(float), t5(float)   (n4)
14  getfield_unresolved t7(float)  = l0(A, NonNull), <A.f2>(n5)
17  float_mul           t8(float)  = l4(float), t7(float)   (n6)
18  float_add           t9(float)  = t6(float), t8(float)   (n7)
21  null_check          l1(B, NonNull)                       (n8)
21  float_load          t10(float) = @{  l1(B), -16 }        (n9)
24  float_mul           t11(float) = l3(float), t10(float)  (n10)
25  float_add           t12(float) = t9(float), t11(float)  (n11)
26  return              t12(float)                           (n12)
    END_BBLOCK          B0@0
```

Figure 8: LIR of method `foo()`

Example: Figure 8 shows the LIR for method `foo` of the example in Figure 5. The labels (n1) through (n12) on the far right of each instruction indicate the corresponding node in the data dependence graph shown in Figure 9.

6.2 Dependence Graph Construction

We construct an instruction-level dependence graph, used during BURS code generation (Section 6.3), for each basic block that captures register true/anti/output dependences,

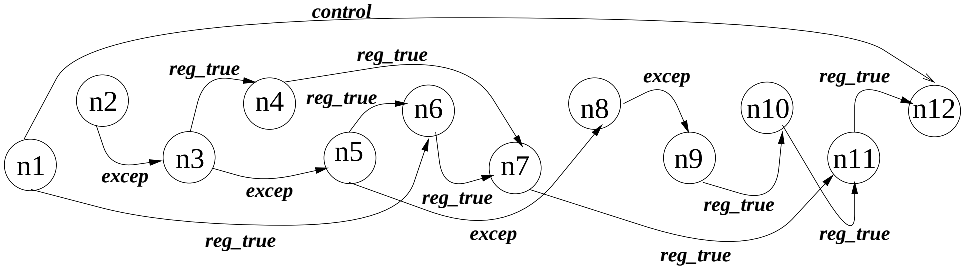


Figure 9: Dependence graph of basic block in method `foo()`

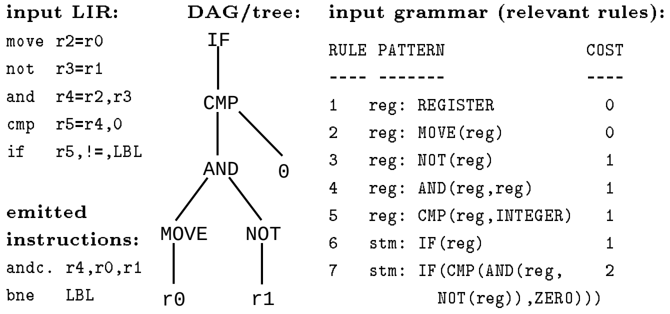
memory true/anti/output dependences, and control dependences. The current implementation of memory dependences makes conservative assumptions about alias information.

Synchronization constraints are modeled by introducing *synchronization dependence* edges between synchronization operations (**`monitor_enter`** and **`monitor_exit`**) and memory operations. These edges prevent code motion of memory operations across synchronization points. Java exception semantics [29] is modeled by *exception dependence edges*, which connect different exception points in a basic block. Exception dependence edges are also added between register write operations of local variables and exception points in the basic block. Exception dependence edges between register operations and exceptions points need not be added if the corresponding method does not have catch blocks. This precise modeling of dependence constraints allows us to perform more aggressive code generation.

Example: Figure 9 shows the dependence graph for the single basic block in method `foo()` of Figure 5. The graph, constructed from the LIR for the method, shows register-true dependence edges, exception dependence edges, and a control dependence edge from the first instruction to the last instruction in the basic block. There are no memory dependence edges because the basic block contains only loads and no stores, and we do not currently model load-load input dependences[6]. An exception dependence edge is created between an instruction that tests for an exception (such as **`null_check`**) and an instruction that depends on the result of the test (such as **`getfield`**).

6.3 BURS-based Retargetable Code Generation

In this section, we address the problem of using tree-pattern-matching systems to perform retargetable code generation after code optimization in the Jalapeño Optimizing Compiler [33]. Our solution is based on partitioning a basic block dependence graph (defined in Section 6.2) into trees that can be given as input to a BURS-based tree-pattern-matching system [15]. Unlike previous approaches to partitioning DAGs for tree-pattern-matching (e.g., [17]), our approach considers partitioning in the presence of memory and exception dependences (not just register-true dependences). We have defined legality constraints for this partitioning, and developed a partitioning algorithm that incorporates code duplication.


input LIR:

```
move r2=r0
not  r3=r1
and  r4=r2,r3
cmp  r5=r4,0
if   r5,!=,LBL
```

emitted instructions:

```
andc. r4,r0,r1
bne   LBL
```

DAG/tree:

input grammar (relevant rules):

RULE	PATTERN	COST
1	reg: REGISTER	0
2	reg: MOVE(reg)	0
3	reg: NOT(reg)	1
4	reg: AND(reg,reg)	1
5	reg: CMP(reg,INTEGER)	1
6	stm: IF(reg)	1
7	stm: IF(CMP(AND(reg, NOT(reg)),ZERO)))	2

Figure 10: Example of tree pattern matching for PowerPC

Figure 10 shows a simple example of pattern matching for the PowerPC. The data dependence graph is partitioned into trees before using BURS. Then, pattern matching is applied on the trees using a grammar (relevant fragments are illustrated in Figure 10). Each grammar rule has an associated cost, in this case the number of instructions that the rule will generate. For example, rule 2 has a zero cost because it is used to eliminate unnecessary register moves, i.e., coalescing. Although rules 3, 4, 5, and 6 could be used to parse the tree, the pattern matching selects rules 1, 2, and 7 as the ones with the least cost to cover the tree. Once these rules are selected as the least cover of the tree, the selected code is emitted as MIR instructions. Thus, for our example, only two PowerPC instructions are emitted for five input LIR instructions. Figure 11 shows the MIR for method `foo` in Figure 5, as generated by our BURS code generator.

[6] The addition of load-load memory dependences will be necessary to correctly support the Java memory model for multithreaded programs that contain data races.

```
    LABEL0              B0@0
2   ppc_fdivs           l4(float)  = l2(float), l3(float)
7   getfield_unresolved t5(float)  = l0(A, NonNull), < A.f1>
10  ppc_fmuls           t6(float)  = l2(float), t5(float)
14  getfield_unresolved t7(float)  = l0(A, NonNull), < A.f2>
17  ppc_fmuls           t8(float)  = l4(float), t7(float)
18  ppc_fadds           t9(float)  = t6(float), t8(float)
21  ppc_lfs             t10(float) = @{ -16, l1(B, NonNull) }
24  ppc_fmuls           t11(float) = l3(float), t10(float)
25  ppc_fadds           t12(float) = t9(float), t11(float)
26  return              t12(float)
    END_BBLOCK          B0@0
```

Figure 11: MIR of method `foo()` with virtual registers

6.4 Register Allocation

Our register allocator framework supports different allocation schemes, according to the available time that can be spent in optimizing a method. We currently employ a linear scan register allocator [32].

The LIR that reaches the register allocator contains two types of symbolic registers: temporaries, obtained from converting stack simulation into registers, and locals, obtained from Java locals specified in the bytecode. We give higher priority to allocating physical registers to those temporaries whose live range does not span a basic block.

The linear scan algorithm is not based on graph coloring, but allocates registers to variables in a single linear-time scan of the variables' live ranges in a greedy fashion. This algorithm is several times faster than algorithms based on graph coloring, and results in code that is almost as efficient as that obtained using more complex allocators [32].

Example: The virtual registers, used by MIR, will be converted into physical registers by the register allocator, as shown in Figure 12. The output of the register allocator also includes prologues and epilogues at the beginning and end of each method, as shown in the figure. Note that no **null_check** instructions appear in the MIR; this is because the Jalapeño JVM's object model allows null-pointer exceptions to be caught without the need for explicit checking.

6.5 Final Assembly

The final phase of the Jalapeño Optimizing Compiler is the assembly phase that emits the binary executable code of an opt-compiled method into an instruction array of `int`. The assembly phase also finalizes the exception table and the stack map of the instruction array, by converting offsets in the IR to offsets in the machine code. The handle of the optimized instruction array, a Java array reference, is stored into a field of the object instance for the method. In addition to the baseline compiled instruction array, the object instance of a method can concurrently hold multiple opt-compiled instruction arrays, each of which is specialized based on factors such as the call-site contexts or the values of the parameters. Selection of a particular instruction array to be invoked at a particular invocation site can be made during compile-time when LIR is generated or at the actual invocation time via back patching.

```
0  LABEL0              B0@0
0  ppc_stwu            FP(int),    @{-24, FP(int) }
0  ppc_ldi             R0(int)   = 4021
0  ppc_stw             R0(int),    @{  4, FP(int) }
0  ppc_mfspr           R0(int)   = LR(int)
0  ppc_stw             R0(int),    @{ 32, FP(int) }
2  ppc_fdivs           F3(float) = F1(float), F2(float)
7  getfield_unresolved F4(float) = R3(A, NonNull), < A.f1>
10 ppc_fmuls           F1(float) = F1(float), F4(float)
14 getfield_unresolved F4(float) = R3(A, NonNull), < A.f2>
17 ppc_fmuls           F3(float) = F3(float), F4(float)
18 ppc_fadds           F1(float) = F1(float), F3(float)
21 ppc_lfs             F3(float) = @{ -16, R4(B, NonNull) }
24 ppc_fmuls           F2(float) = F2(float), F3(float)
25 ppc_fadds           F1(float) = F1(float), F2(float)
0  ppc_lwz             R0(int)   = @{ 32, FP(int) }
0  ppc_mtspr           LR(int)   = R0(int)
0  ppc_addi            FP(int)   = FP(int), 24
26 ppc_blr             LR(int)
   END_BBLOCK          B0@0
```

Figure 12: MIR of method `foo()` with physical registers

6.6 Generation of Exception Tables and GC Stack-Maps

An exception table for an opt-compiled method is constructed during BC2IR using the information in the class file. The entries in the table are in terms of the HIR instructions, and the table is updated as high-level optimizations applied to the HIR result in modifications in the HIR. The table is also updated as the HIR is converted into LIR, as optimizations are applied to the LIR, and as LIR is converted into MIR (machine-specific IR). Since different activation records on the run-time stack may be generated by methods compiled by different compilers (baseline and optimizing), a common interface among compilers is used, making the Jalapeño exception handler unaware of which compiler is providing this exception table.

When a garbage collection occurs, the type-accurate garbage collector needs mapping information describing which registers and stack locations (register spills) hold references (object pointers). As these locations vary among program points, a different map could be generated for each program point. However, since garbage collection can only occur at certain predefined points called GC points, maps are only stored for these points. Jalapeño employs a common interface among compilers, analogous to the one for exception tables, making the garbage collector unaware of which compiler generates this stack map information.

7 Flow-Insensitive Optimizations

Based on profiling feedback, a dynamic compiler can reserve the most time-consuming optimizations for "hot spots" in the code, and rely on quicker optimizations for other sections. We focus on a few quick, flow-insensitive optimizations here. Fast and effective flow-sensitive algorithms remain topics for future work.

The optimizing compiler performs several types of flow-insensitive optimizations. Clearly, when optimizing the HIR or LIR, the compiler can quickly perform transformations local to a basic block, such as local common subexpression elimination and elimination of redundant local exception checks. Furthermore, some semantic expansion transformations of standard Java library classes do not require flow-sensitive information [39].

To optimize across basic blocks, we can exploit the JVM specification which ensures that "Every variable in a Java program must have a value before it is used" [29]. Using this rule, if any variable has only one definition, then that definition reaches every use of the variable. For such variables, we can build def-use chains and perform copy propagation and dead code elimination, without any expensive analysis. This technique will conservatively catch many optimization opportunities, but will miss some cases that flow-sensitive analysis would detect.

Threads, exceptions, and garbage collection constrain transformations such as code motion, redundancy elimination, loop optimizations, and locality-enhancing optimizations. Our future flow-sensitive optimization algorithms must consider the semantics of these language features. Section 6.2 addressed these issues for modeling exceptions and synchronization when building a data dependence graph.

8 Inline Expansion of Method Calls

Our current implementation performs inlining at two stages during the translation process: during front-end BC2IR translation and during HIR optimization.

8.1 Inlining in BC2IR

The optimizing compiler performs top-down inlining during BC2IR. To inline a call site, the BC2IR implementation processes the basic blocks of the callee method as if they belong to the caller. This approach has the advantage that the front-end's top-down optimizations, such as constant folding and constant propagation, naturally extend into the inlined method body. Additionally, the front-end translation process automatically links thrown exceptions in the inlined method to catch blocks in the caller.

We currently use static code size and depth heuristics to decide whether or not to inline. The top-down on-the-fly inlining hinders the efficacy of static heuristics, since at the time we must decide to inline, we have not yet seen the full call graph. To bypass this limitation, in future work, the controller will make inlining decisions based on profiling information, and pass an "inlining plan" to BC2IR.

Inlining of Java static and final methods is always safe. Inlining virtual methods presents some complications, and is postponed until the HIR optimization phase, as described next.

8.2 Inlining in HIR Optimization phase

During HIR optimization, we wish to perform analysis and transformations based on semantics of some special Java bytecodes, such as `monitorenter` and `new`. We preserve these bytecodes during HIR optimization, and expand them into inlined method calls immediately prior to conversion from HIR to LIR. To inline at the HIR level, we generate a new HIR for the inlined call, and patch it into the caller HIR. The patching process updates the control flow graph as necessary, including setting up links from the callee to exception handlers in the caller.

Inlining virtual methods is more complicated than inlining static and final methods. We currently inline selected virtual methods during HIR optimization, predicting the receiver of a virtual call to be the declared type of the object, and rely on static bytecode-size heuristics. We guard each inlined virtual method with a run-time conditional test to verify that the receiver is predicted correctly, and default to a normal virtual method invocation if it is not. In future work, we will examine guards with run-time trap instructions, which may run faster on current processors for correct predictions, but necessitate more complex and costly recovery for incorrectly predicted methods (see Section 10.3).

9 Performance Results

9.1 Implementation Status

As shown in Figure 1, the Jalapeño Adaptive Optimization System consists of three major subsystems: Online Measurements (OLM), Controller, and Optimizing Compiler. To date, our main implementation focus has been the Optimizing Compiler subsystem. Our initial implementation of the optimizing compiler targets the PowerPC and correctly supports all JVM bytecodes, including support for exceptions and threads. For the experimental results reported below, a conservative non-copying garbage collector was used because the optimizing compiler's generation of GC stack-maps is not yet robust enough to run all of the larger benchmarks.

Although a prototype version of the OLM subsystem has been built, work on the controller subsystem is still in the design phases. Therefore, the optimizing compiler can currently only be invoked as either a static compiler or dynamically by the Jalapeño JVM class loader to optimize all methods of all dynamically loaded classes.

9.2 Experimental Methodology

The performance results in this section were obtained on an IBM F50 Model 7025 with four 166MHz PPC604e processors running AIX v4.3. The system has 1GB of main memory. Each processor has split 32KB first-level instruction and data caches and a 256KB second-level cache. Because of the incomplete implementation of the Controller and OLM subsystems, in this paper we only present results for time spent in program execution; time spent in dynamic compilation was not instrumented or measured.

In the next subsection, we compare results using the following four Java environments:

- *JDK w/o JIT:* The IBM enhanced port of the Sun JDK 1.1.6 interpreter (without the JIT).
- *JDK w/ JIT:* The IBM enhanced port of the Sun JDK 1.1.6 with v3.0 of the IBM JIT compiler [23]. This product compiler performs an extensive set of optimizations, including inlining of math library methods, virtual methods, recursive calls, field privatization, constant propagation, dead store elimination, elimination of redundant numerical type-casts, elimination of redundant exception checks, common subexpression elimination, optimized loop generation, register allocation, and instruction scheduling.
- *Jalapeño Baseline:* The Jalapeño Virtual Machine configured to use the Jalapeño Baseline Compiler as a JIT for all classes dynamically loaded by the application.
- *Jalapeño Optimizer:* The Jalapeño Virtual Machine configured to use the Jalapeño Optimizing Compiler as a JIT for all classes dynamically loaded by the application. The following optimizations were performed: inlining of static and final methods, semantic inlining of selected library routines, limited static class prediction to safely inline virtual methods, linear scan register allocation, limited constant propagation, type propagation, unreachable code elimination, local common subexpression elimination, flow-insensitive copy propagation and dead code elimination, and local redundant bounds check elimination.

Test	JDK w/o JIT	Jalapeno Baseline	JDK w/ JIT	Jalapeno Optimizer
BSort	77.19	34.26	3.20	3.94
Bi BSort	67.93	30.49	2.32	3.10
Qsort	15.27	6.10	1.11	0.78
Sieve	11.47	4.74	0.34	0.42
Hanoi	17.84	7.90	1.00	1.54
Dhrystone	7.12	2.33	0.65	0.68
Tree	9.87	14.49	2.44	3.40
Fibonacchi	20.23	11.58	1.75	0.98
Array	4.95	10.15	1.01	0.84
Compress	85.67	46.08	5.86	7.23
DB	7.18	3.89	1.73	2.94
Javac	7.52	4.21	2.29	7.63
Jack	31.36	23.46	7.06	10.54

Table 1: Execution times (seconds)

In both Jalapeño configurations, the "boot image" containing the Jalapeño JVM itself was created by using the Jalapeño Optimizing Compiler as a static compiler performing all the optimizations listed above.

9.3 Micro-benchmark Programs

To evaluate code quality, Figure 13 and Table 1 compare the performance on these four Java environments for nine micro-benchmarks developed by Symantec Corporation. For the micro-benchmarks, we report the mean wall-clock execution time for the last ten of eleven runs; standard deviations were negligible.

The results show that on three of the nine tests (QSort, Fibonacci, Array), the optimizing compiler delivers better performance than the product JIT compiler. This is encouraging because the product JIT compiler performs many more optimizations than the current implementation of the optimizing compiler. Performance on Dhrystone is roughly equivalent, and on the remaining five tests the optimizing compiler performance is within a factor of 1.6 of the product JIT compiler.

9.4 Macro-Benchmark Programs

To evaluate system performance on medium-sized benchmarks, we present performance results on several codes from the SPECjvm98 suite [14]. The system currently runs four of the seven tests (`_201_compress`, `_209_db`, `_213_javac`, `_228_jack`); the others do not yet run due to incomplete library support.

We ran the tests using the SPEC driver program, configured to run each test between two and four times, and report the best wall-clock time. This methodology factors

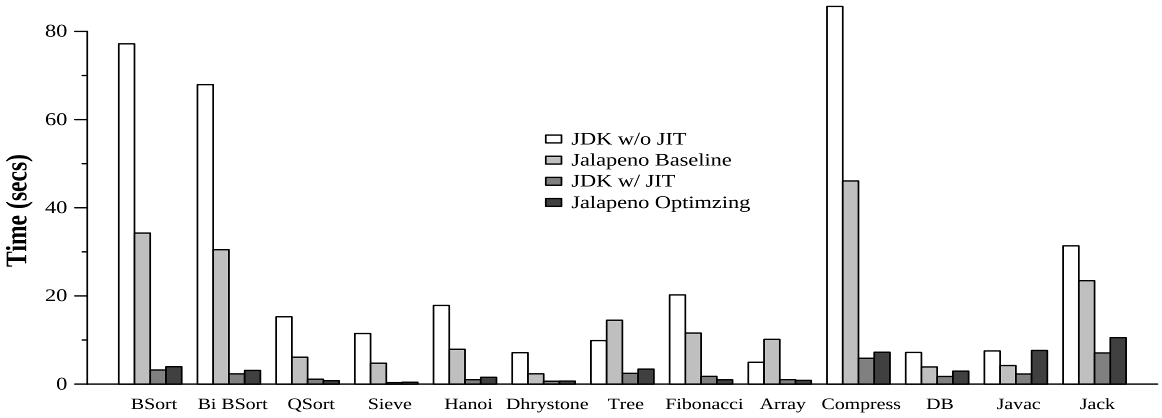


Figure 13: Execution time (in seconds) for micro and macro benchmarks

out compile-time. We run the benchmarks using the SPEC problem size parameter set to 10, for medium-size input parameters. Note that these results do *not* follow the official SPEC reporting rules, and therefore should not be treated as official SPEC results.

Figure 13 and Table 1 show the results in the four Java environments enumerated above. The results show that the current optimizing compiler runs these codes between 1.2 to 3.3 times slower than the product JIT. We believe that the performance inversion of the Jalapeño Baseline and Optimizing configurations for `javac` can be attributed to performance problems in the conservative GC subsystem. Given the current immature state of our JVM and compiler, we are encouraged that our performance is within an order of magnitude of the best current commercial technology. These results suggest that with much more tuning and further optimization, a JVM written entirely in Java may achieve performance competitive with a state-of-the-art JVM implemented in C.

10 Interprocedural Optimizations — Extensions to Current Implementation

This section describes two interprocedural optimizations that are in progress as extensions to the current implementation — interprocedural optimization of register saves and restores (Section 10.1), and interprocedural escape analysis (Section 10.2). In addition, Section 10.3 discusses issues related to interprocedural optimization in the presence of dynamic class loading.

10.1 Interprocedural Optimization of Register Saves and Restores

To optimize register saves and restores at call sites [12, 34], we first perform interprocedural register usage analysis. Interprocedural register usage is a backward analysis performed over the call graph of a program to determine the register requirements across method boundaries. Without interprocedural analysis, all caller save registers have to be saved and restored at call sites even if they are not used in callee methods. For interprocedural analysis we first construct a call graph of all methods that are compiled by the optimizing compiler. The call graph accommodates virtual method call sites. We then process the methods in the call graph in reverse topological order.[7] The analysis assumes that registers are allocated contiguously, starting from the first available register. For each method we first compute the number of registers used intraprocedurally. This information is propagated back to caller(s) of the method. The value at each node gives the number of registers that have to be saved and restored at each call site which invokes that node. If the callee is compiled with the baseline compiler, then we propagate $\perp$ to the caller ($\perp$ indicates that all live caller save registers have to be saved and restored at that call site). In the presence of cycles, we can either compute a fixed point or propagate $\perp$.

To illustrate our approach consider the call graph shown in Figure 14. The numbers on the left of each node correspond to the register requirements within each method (computed intraprocedurally). The numbers enclosed in square brackets are the register requirements computed during our backward interprocedural analysis. The meet of two values is the maximum value of the two values, and meet of a $\perp$ with any value is a $\perp$.

10.2 Escape Analysis

Escape analysis is a technique for determining whether an object that is created in a method may escape a call to the method. The most well-known application of escape analysis

[7] We ignore the back edges while determining the topological order.

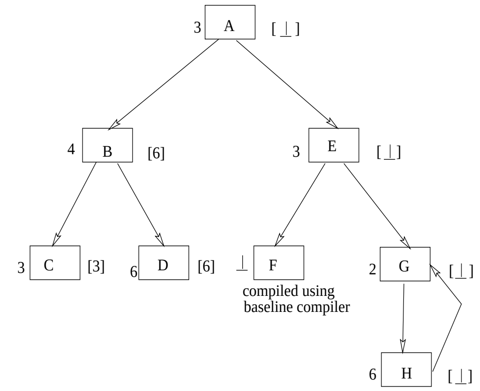


Figure 14: An example for interprocedural register saves and restores

is to allocate non-escaping objects on the stack instead of the heap. This leads to (i) reduced overheads of object allocation and deallocation (i.e., garbage collection, for languages that support it), and (ii) usually, improved data locality [6]. In the context of Java, escape analysis can also be used to identify objects that are local to a thread. This leads to another important benefit – reduced synchronization overhead, as we can eliminate the locking operations on thread-local objects.

We have developed several algorithms for performing escape analysis. Among them, we have implemented an approach based on a *connection graph* abstraction, in the context of a static compiler to evaluate its potential benefits. Our results are encouraging, especially for reducing synchronization overheads [11]. We are currently implementing escape analysis in the Jalapeño Optimizing Compiler.

10.3 Handling Dynamic Class Loading

Interprocedural optimizations and those optimizations that depend on the structure of the class inheritance graph can become invalid in the presence of dynamic class loading. When an optimization applied to a method becomes invalid in this manner, all optimized versions of the method currently on the call stack must be replaced. One option is to replace them by the unoptimized version of the method. We maintain a "resolution dependence graph" during the optimization phase that indicates which methods could be affected when a new class is loaded.

Consider the interprocedural optimization of register saves and restores described in Section 10.1. Assume, for example, that a new class can be loaded at the point where C is called (in Figure 14). Also, assume that we have a new call C' in place of C. Methods C', B, and A are affected. Now assume that the intraprocedural register requirement for C' is 6. Only the register requirement of C' is affected, since the register requirement of B is already 6. In summary, we essentially incrementally update only affected nodes. For this purpose we can use techniques from incremental data flow analysis and compilation [7, 26].

The invalidation mechanism for dynamic class loading has not yet been implemented.

11 Related Work

Dynamic compilation, also called dynamic translation or just-in-time compilation, has been a key ingredient in a number of previous implementations of object-oriented languages. Deutsch and Schiffman's high performance implementation of Smalltalk-80 dynamically translated Smalltalk bytecodes to native code [16]; their compiler was quite similar to our baseline compiler. Implementations of the Self language also relied on dynamic compilation to achieve high performance [8]. All three generations of Self compilers utilized register-based intermediate representations that are roughly equivalent to the one used by the Jalapeño Optimizing Compiler. Recently, a number of just-in-time compilers have been developed for the Java language [2]. Some of these compilers translate bytecodes to a three-address code, perform simple optimizations and register allocation, and then generate target machine code.

A number of previous systems have utilized more specialized forms of dynamic compilation to selectively optimize program hot spots by exploiting "run-time constants" [13, 5, 31, 19]. In general, these systems emphasize extremely fast dynamic compilation, often performing extensive off-line precomputations to avoid constructing any explicit representation of the program fragment being compiled at dynamic compile-time.

Implementing a Java virtual machine and its related subsystem (including the optimizer) in Java opens several challenges. Taivalsaari [36] also describes a "Java in Java" implementation to examine the feasibility of a high quality virtual machine written in Java. One drawback of this approach is that it runs on another Java virtual machine, which adds performance overhead because of the two-level interpretation process. Our approach avoids the need for another JVM by bootstrapping the system. Compared to Taivalsaari's system we have also implemented several optimizations to improve the performance of the overall system and Java applications.

A large collection of work addresses optimizations specific to object-oriented languages, such as class analysis, both intraprocedural [10] and interprocedural (see related work in [20]), class hierarchy analysis and optimizations [37, 35], receiver class prediction [16, 21, 9], method specialization [37], and call graph construction (see related work in [20]). Other optimizations relevant to Java include bounds check elimination [30] and semantic inlining [39].

12 Conclusions and Future Work

The use of Java in many important server applications depends on the availability of a JVM that supports efficient execution of such applications on server machines. Jalapeño is one such JVM. Our ability to correctly execute a wide range of large Java programs has validated the soundness of Jalapeño's compile-only approach to program execution. In addition, our preliminary performance results show that, even with its current limited set of optimizations, the Jalapeño Optimizing Compiler is capable of delivering performance that is comparable to the performance delivered by a production-strength JIT compiler. The fact that the Jalapeño run-time system (and the rest of the JVM) is implemented in Java makes this achievement all the more remarkable. To the best of our knowledge, the Jalapeño Optimizing Compiler is the first dynamic optimizing compiler for Java that is being used in a JVM with a compile-only approach to program execution.

There are many challenging directions for future research based on the Jalapeño Optimizing Compiler. In the area of optimizations, we already described two interprocedural optimizations in Section 10 that are currently in progress. In addition, we have begun work on flow-sensitive optimizations using Array SSA form [27, 28] and context-sensitive profile-directed inlining of method calls based on the Calling Context Graph.

Acknowledgments

We would like to thank other past and present members of the Jalapeño JVM team — Bowen Alpern, Dick Attanasio, John Barton, Perry Cheng, Anthony Cocchi, Brian Cooper, Susan Hummel, Derek Lieber, Vassily Litvinov, Mark Mergen, Ton Ngo, Igor Pechtchanski, Jim Russell, Janice Shepherd, Steve Smith, Peter Sweeney — for their contributions in building the rest of the JVM and for various suggestions in the design of the Jalapeño Optimizing Compiler. We also thank Laureen Treacy for her proofreading assistance.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Ali-Reza Ald-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[3] Glen Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997.

[4] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[5] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, May 1996.

[6] B. Blanchet. Escape analysis: Correctness, proof, implementation and experimental results. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 25–37, January 1998.

[7] Michael Burke and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[8] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, March 1992. Published as technical report STAN-CS-92-1420.

[9] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. Technical Report UW-CSE-96-06-02, University of Washington, Department of Computer Science and Engineering, June 1996.

[10] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizating dynamically-typed object-oriented programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 150–164, 1990.

[11] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape analysis for Java. Technical report, IBM T.J. Watson Research Center, 1999.

[12] Fred C. Chow. Minimizing Register Usage Penalty at Procedure Calls. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, July 1988. *SIGPLAN Notices, 23(7).*

[13] Charles Consel and Franccois Noël. A general approach for run-time specialization and its application to C. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 145–156, January 1996.

[14] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998.

[15] R.R. Henry C.W. Fraser and T.A. Proebsting. Burg — fast optimal instruction selection and tree parsing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[16] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Annual ACM Symposium on the Principles of Programming Languages*, pages 297–302, January 1984.

[17] M. Anton Ertl. Optimal code selection in DAGs. In *26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1999.

[18] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimzing compiler for java. submitted for publication, draft at http://www.research.microsoft.com/apl/, October 1998.

[19] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, to appear.

[20] Dave Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, October 1997.

[21] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994. *SIGPLAN Notices, 29(6)*.

[22] IBM. IBM's high performance compiler for java. White paper at http://www.alphaworks.ibm.com.

[23] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganama, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *ACM Java Grande Conference*, San Fransisco, CA, June 1999.

[24] Richard Jones and Rafael Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[25] Jove. Jove, super optimizing deployment environment for Java. White paper at http://www.instantiations.com/javaspeed/jovereport.htm.

[26] Michael Karasick. The architecture of Montana: An open and extensible programming environment with an incremental c++ compiler. In *Sixth International Symposium on Foundations of Software Engineering*, pages 131–142, November 1998.

[27] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in Parallelization. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1998.

[28] Kathleen Knobe and Vivek Sarkar. Conditional constant propagation of scalar and array references using array SSA form. In Giorgio Levi, editor, *Lecture Notes in Computer Science, 1503*, pages 33–56. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium.*

[29] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1996.

[30] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.

[31] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 109–121, June 1997.

[32] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM TOPLAS*, 1999. To appear.

[33] Vivek Sarkar, Mauricio J. Serrano, and Barbara B. Simons. "Retargeting Optimized Code by Matching Tree Patterns in Directed Acyclic Graphs", Patent Application, submitted in December 1998.

[34] Peter A. Steenkiste and John L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, 1989.

[35] Peter F. Sweeney and Frank Tip. A study of dead data members in C++ applications. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 324–332, June 1998. *SIGPLAN Notices, 33(5)*.

[36] Antero Taivalsaari. Implementing a Java virtual machine in the java programming language. Technical Report SMLI TR-98-64, Sun Microsystems, March 1998.

[37] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1997.

[38] John Whaley. Dynamic optimization through the use of automatic runtime specialization. M.eng., Massachussetts Institute of Technology, May 1999.

[39] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *ACM Java Grande Conference*, San Fransisco, CA, June 1999.