APRIL 2000

WRL Research Report 2000/2

The Swift Java Compiler: Design and Implementation

Daniel J. Scales
Keith H. Randall
Sanjay Ghemawat
Jeff Dean

COMPAQ **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL), located in Palo Alto, California, is part of Compaq's Corporate Research group. Our focus is research on information technology that is relevant to the technical strategy of the Corporation and has the potential to open new business opportunities. Research at WRL ranges from Web search engines to tools to optimize binary codes, from hardware and software mechanisms to support scalable shared memory paradigms to graphics VLSI ICs. As part of WRL tradition, we test our ideas by extensive software or hardware prototyping.

We publish the results of our work in a variety of journals, conferences, research reports and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes, conference papers, or magazine articles. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

You can retrieve research reports and technical notes via the World Wide Web at:

http://www.research.compaq.com/wrl

You can request research reports and technical notes from us by mailing your order to:

Technical Report Distribution
Compaq Western Research Laboratory
250 University Avenue
Palo Alto, CA 94301 U.S.A.

You can also request reports and notes via e-mail. For detailed instructions, put the word "Help" in the subject line of your message, and mail it to:

wrl-techreports@pa.dec.com

The Swift Java Compiler: Design and Implementation

Daniel J. Scales, **Keith H. Randall**, **Sanjay Ghemawat**, and **Jeff Dean**
Western Research Laboratory and System Research Center
Compaq Computer Corporation

Abstract

We have designed and implemented an optimizing Java compiler called Swift for the Alpha architecture. Swift translates Java bytecodes to optimized Alpha code, and uses static single assignment (SSA) form for its intermediate representation (IR). The Swift IR is relatively simple, but allows for straightforward implementation of all the standard scalar optimizations. The Swift compiler also implements method resolution and inlining, interprocedural alias analysis, elimination of Java run-time checks, object inlining, stack allocation of objects, and synchronization removal. Swift is written completely in Java and installs its generated code into a high-performance JVM that provides all of the necessary run-time facilities.

In this paper, we describe the design and implementation of the Swift compiler system. We describe the properties of the intermediate representation, and then give details on many of the useful optimization passes in Swift. We then provide some overall performance results of the Swift-generated code for the SpecJVM98 benchmarks, and analyze the performance gains from various optimizations. We find that the Swift design is quite effective in allowing efficient and general implementations of numerous interesting optimizations. The use of SSA form, in particular, has been especially beneficial, and we have found very few optimizations that are difficult to implement in SSA.

1 Introduction

Though originally used mainly in web applications, the Java programming language is becoming popular as a general-purpose programming language, because of its simple design and its properties that improve programmer productivity. Its object-oriented features promote modularity and simplify the coding of many kinds of applications. Its type-safe design detects or eliminates many programming bugs that can occur in other languages. In addition, its automatic memory management takes care of a time-consuming aspect of developing code for the application programmer.

We have developed a complete optimizing compiler for Java called Swift. Because we believe Java will be a popular general-purpose programming language, we are interested in developing an intermediate representation and a set of advanced optimizations that will enable us to generate highly efficient code for Java programs. Most existing commercial Java systems do just-in-time (JIT) compilation that can do only limited optimizations, but we are interested in studying more expensive optimizations that can improve the relative efficiency of code generated from Java, as compared to other languages, such as C.

Java presents many challenges for a compiler attempting to generate efficient code. The required run-time checks and heap allocation of all objects can introduce considerable overhead. Many routines in the standard Java library include synchronization operations, but these operations are unnecessary if the associated object is only being accessed by a single thread. Virtual method invocation is quite expensive if it cannot be transformed to a direct procedural call. In addition, if a virtual method call cannot be resolved, the unknown call can reduce the effectiveness of various interprocedural analyses. Many operations can cause exceptions because of run-time checks, and the possibility of these exceptions further constrains many optimizations.

On the other hand, the strong typing in Java simplifies much of the compiler analysis. In particular, Java semantics ensure that a local variable can only be modified by explicit assignment to the variable. In contrast, the creation of a pointer to a local variable in C allows a local variable to be modified at any time by any routine. Similarly, a field of a Java object can never be modified by an assignment to a different field. Because of these properties that make most data dependences explicit in Java programs, we believe that static single assignment (SSA) form is highly appropriate as an intermediate representation for Java. SSA graphs make most or all of the dependences in a method explicit, but provide great freedom in optimizing and scheduling the resulting code.

We have therefore adopted SSA form in Swift, and have used Swift to study the effectiveness of SSA form in expressing various optimizations. Though Swift is a research platform, it is currently a complete Java compiler with all of the standard optimizations and numerous advanced optimizations. Swift is written completely in Java and installs its generated code into a high-performance JVM [12] that provides all of the necessary run-time facilities.

In this paper, we describe the design and implementation of the Swift compiler system, and provide performance results. The organization of the paper is as follows. We first describe the intermediate representation in detail, and dis-

cuss some of its advantages. We next describe the structure of the compiler and the many interprocedural and intraprocedural optimizations in Swift. We then provide some overall performance results of the Swift-generated code for the SpecJVM98 benchmarks, and analyze the performance gains from various optimizations. Finally, we discuss related work and conclude.

2 Swift Intermediate Representation

In this section, we describe the major features of the Swift intermediate representation (IR). A method is represented by a static single assignment (SSA) graph [3, 15] embedded in a control-flow graph (CFG). We first describe the structure of the SSA graph, the structure of the CFG, and the relationship between the two. Next we describe the type system used to describe nodes in the SSA graph. We then describe our method of representing memory operations and ensuring the correct ordering among them. We also describe the representation of method calls. Finally, we discuss some of the advantages of the Swift IR.

2.1 Static Single Assignment Graph

As we mentioned above, the basic intermediate form used in Swift is the SSA graph. We consider the SSA form of a method to be a graph consisting of nodes, which we call *values*, that represent individual operations. Each value has several inputs, which are the result of previous operations, and has a single result, which can be used as the input for other values. Since a value has only a single result, we will frequently identify a value with the result that it produces.

In addition to its inputs, each value has an operation field and an auxiliary operation field. The operation field indicates the kind of operation that the value represents. For example, if the operation field is **add**, then the value represents an operation that adds the two incoming values to produce its result. The auxiliary operation field is used to specify any extra static information about the kind of operation. For example, if the operation field is **new**, then the value represents an operation that allocates a new object, and the auxiliary field specifies the class of the object to be allocated. If the operation field is **constant**, the value represents a numeric or string constant, and the auxiliary field specifies the actual constant.

The Swift IR contains about 55 basic operations, as summarized in Figure 1. These operations include the usual arithmetic, bitwise-logical, and comparison operators. They also include operations for controlling program flow, merging values (phi nodes [15]), invoking methods, accessing the contents of arrays and objects, and allocating new objects and arrays from the heap. Some object-oriented operations include instanceof computations, virtual method calls, and interface calls. There are also a few primitives that are particularly Java-specific, such as operations modeling the `lcmp` or `fcmpl` bytecodes.

Numeric operations
- constant
- add, sub, mul, div, rem, negate
- shl, shr, ushr
- and, or, xor, not
- lt, leq, eq, neq, geq, gt
- lcmp, fcmpl, fcmpg
- convert

Control operations
- if, switch, throw
- arg, return
- phi
- invoke_virtual, invoke_special, invoke_static, invoke_interface

Memory operations
- get_field, put_field
- get_static, put_static
- arr_load, arr_store, arr_length

Run-time checks
- instanceof
- null_ck, bounds_ck, cast_ck
- init_ck

Miscellaneous
- nop
- select
- new, new_array
- monitor_enter, monitor_exit

Figure 1: Summary of Swift IR operations

As with most other Java compilers, the Swift IR breaks out the required run-time checks associated with various Java bytecodes into separate operations. The Swift IR therefore has individual operations representing null checks, bounds checks, and cast checks. These operations cause a run-time exception if their associated check fails. A value representing a run-time check produces a result, which has no representation in the generated machine code. However, other values that depend on the run-time check take its result as an input, so as to ensure that these values are scheduled after the run-time check. Representing the run-time checks as distinct operations allows the Swift compiler to apply optimizations to the checks, such as using common subexpression elimination on two null checks of the same array.

As an example, Figure 2 shows the expansion of a Java array load into Swift IR. The ovals are SSA values, and the boxes are blocks in the CFG. (The CFG will be discussed in the next section.) *array* and *index* are the values that are input to the array load operation. Java requires that a null check and a bounds check be done before an element is loaded from an array. The **null_ck** value takes the array reference as input, and throws a `NullPointerException` if the reference is null. The **arr_length** value takes an array reference and the associated **null_ck** value as input, and produces the length of the array. The **bounds_ck** value takes an array length and an array index as inputs, and throws an `ArrayIndexOutOfBoundsException` if the index is not within the bounds of the array. Finally, the **arr_load** value takes an array reference, an index, and the associated null-check and bounds-check values, and returns the specified element of the array.

Swift also has a value named **init_ck** that is an explicit representation of the class-initialization check that must precede some operations. This value checks if a class has been initialized, and, if not, calls the class initialization method. Operations that load a class variable or create a new object both require an initialization check of the associated class. Calls to class methods also require this check, but this check is handled by the initialization code of the class method.

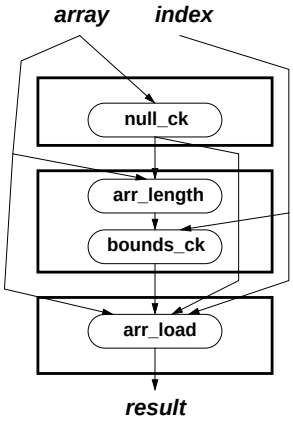

Figure 2: Expansion of Java array load to Swift IR

Swift will eliminate redundant initialization checks during optimization. In addition, our underlying JVM replaces an initialization check by a NOP after the first time that it is encountered.

The Swift IR also has about 100 machine-dependent operations which represent or map very closely to specific Alpha machine instructions. As described below, one pass of the compiler converts many of the machine-independent operations into one or more machine-dependent operations. The conversion to machine-dependent values allows for optimization of the lower-level operations, and allows the instruction scheduling, register allocation, and code generation passes to operate directly on the SSA graph.

Our SSA graph is essentially a factored representation of the use-def chains for all variables in a method, since each value explicitly specifies the values used in computing its result. We have found that many of our optimizations can make efficient use of def-use information as well. When Swift builds an SSA graph, it therefore also builds up def-use information and it updates the def-use chains whenever the graph is manipulated. Thus, an optimization can, at any stage, directly access all the users of a particular value.

2.2 Representing Control Flow

The Swift IR actually represents a method as an SSA graph embedded within a control-flow graph (CFG). Each value is located in a specific basic block of the CFG, although various optimizations may move values among blocks or even change the CFG. A *block* in a CFG has zero or more incoming edges and zero or more outgoing edges. Some of the outgoing edges may represent the control-flow that results from the occurrence of an exception. These edges are labeled with the type of exception that causes flow along that edge.

Each method's CFG has a single entry block, a single normal exit block, and a single exceptional exit. The entry block contains the values representing the input arguments of the method. The exit block contains the value representing the return operation of the method. The exceptional exit block represents the exit of a method that results when an exception is thrown that is not caught within the current method. Because many operations can cause run-time exceptions in Java and these exceptions are not usually caught, there are many blocks which have an exception edge to the exception exit.

We have chosen in the Swift IR to use the standard definition of a basic block. All blocks end whenever an operation is reached that has two or more control exits. An operation that causes an exception therefore always causes the end of a basic block. Because of the large number of operations in Java that can cause run-time exceptions, this design can result in CFGs with large numbers of blocks. Some other Java compilers [8, 22] have used an intermediate form with extended basic blocks, in which an exception-causing operation does not necessarily end a block in the CFG. The use of extended basic blocks may reduce the amount of memory used to represent the CFG. However, we believe that many of the dataflow calculations and interesting optimizations will have to treat the extended basic blocks as separate blocks anyway in order to achieve the desired precision, and therefore will become more complex. In addition, the larger basic blocks are not as important for good scheduling, since Swift uses global code motion and trace scheduling.

Many types of values can affect the control flow of a program. An **if** node takes a boolean value as input and determines the control flow out of the current block based on that input. Similarly, a **switch** node determines control flow based on an integer input. Exception-causing operations include method calls, run-time checks, and object or array allocation. In addition, the **throw** operation explicitly throws an exception.

Each block has a reference to a distinguished value, called the *control value*. For a block which has more than one outgoing edge, the control value is the value that controls the program flow or that may cause an exception. The control value of the normal exit block is the **return** value. Simple blocks with a single outgoing edge have no control value. The control value field of a block provides easy access to the exception-causing or control-flow value of the block. In addition, the set of control values indicates the base set of values in a method that are "live", because they are used in computing the return value and for controlling program flow. Other live values are determined recursively based on the inputs of this base set. Swift uses this definition to do dead code elimination of values that are no longer needed.

All values which are the control value of a block cannot be moved from their block. We say that these values are *pinned*. Phi nodes are also pinned. Operations that write to the global heap (see Section 2.4) are also pinned. All other operations are not pinned, and may be moved freely among blocks, as long as their data dependences are respected.

The Java VM has bytecodes that do light-weight subroutine calls and returns within a method. These bytecodes are used to implement the `finally` statements without du-

plicating bytecodes. However, these subroutines complicate control-flow and data-flow representations. We have therefore chosen to inline these subroutines in the Swift IR, as is done by most other Java compilers [22]. There is a possibility of exponential blowup in the size of the CFG if there are multiply nested `finally` clauses, but in practice this situation is very unlikely.

2.3 Swift Type System

Every value in the SSA graph also has a program type. The type system of the Swift IR can represent all of the types present in a Java program. These types include base types, such as integer, short, or double, as well as array types and object types. We compute the type of each value of a graph as we build the SSA graph from the method's bytecode. As has been noted elsewhere [26], the bytecode for a method does not always contain enough information to recover the exact types of the original Java program. However, it is always possible to assign a consistent set of types to the values such that the effects of the method represented by the SSA graph are the same as the original method. Though Java bytecode does not make use of an explicit boolean type, we assign a type of boolean to a value when appropriate. The boolean type essentially indicates an integer value which can only be 0 and 1, and can enable certain optimizations that don't apply to integers in general.

For some operations, the value type actually further specifies the operation and therefore will affect the specific machine code generated. For example, the generic **add** operation can specify an integer, long, or floating-point addition, depending on its result type. Information about a value's type can also help optimize an operation that uses that value. For example, the compiler may be able to resolve the target of a virtual method call if it has more specific information about the type of the method call's receiver.

The Swift type system contains additional information that facilitates these kinds of optimizations. It allows specification of the following additional properties about a value with a particular Java type T:

- the value is known to be an object of exactly class T, not a subclass of T
- the value is an array with a particular constant size
- the value is known to be non-null

By incorporating these properties into the type system, we can describe important properties of any value in the SSA graph by its type. In addition, we can easily indicate properties for different levels of recursive types, such as arrays. One possible generalization of the type system is to allow union types. However, we have not found this extension to be very useful for the Java applications that we have examined.

2.4 Representing and Ordering Memory Operations

Because of Java's strong typing and lack of pointers, the local variables of a method cannot be read or written by any other method in the same thread or another thread. Therefore, all program dependences on a local variable are explicit in the current method, and SSA form is ideal for representing these dependences. All values in the SSA graph can be considered to be temporary variables stored in a large set of virtual registers. It is only near the end of the compilation process that the register allocator must choose which values are stored in actual registers and which are spilled to the stack.

On the other hand, reads and writes of global variables or locations allocated from the heap must be represented explicitly as memory operations, since there may be many hidden dependencies among these operations. In Java, these memory operations include getting and putting values into the field of a class object or instance object, and getting or putting a value into an array.

In general, the compiler must ensure that the original ordering among memory operations is maintained, even though the memory operations are not connected in the standard SSA graph. For example, a store operation may write into a field that is read by a later load operation. The store operation does not produce any value that is used by the later load operation, so there is no scheduling dependence between the two operations. However, in order to maintain the original program semantics, the compiler must ensure that the store is executed before the load.

The Swift IR solves this problem by actually having store operations produce a result, which is called a *global store* [10]. The global store represents the current contents of global memory. Since a store operation modifies the contents of memory, it takes the latest global store as an extra input and produces a new global store. Additionally, each load operation takes the latest global store as an extra input. With one exception, these connections now correctly represent the data dependences between memory operations, and therefore allow many optimizations to immediately generalize to memory operations. The exception is that the SSA graph does not include required dependences – known as *anti-dependences* – between load operations and following store operations. The Swift compiler enforces these dependences via explicit checks during scheduling. It simply adds extra constraints to ensure that a load operation which takes a global store S as input is not moved down past a store operation that modifies S (i.e. takes S as input). We chose not to include anti-dependences edges in the SSA graph mainly to reduce the number of edges in the graph. Note also that an anti-dependence between a load and a store operation does not indicate that the load is an input of the store, so representing anti-dependences would require a new kind of edge in the SSA graph.

If the current global store is different on two incoming

edges of a block, then a phi node must be inserted in that block to merge the two global stores into a new current store, just as with any other kind of value. The phi node ensures that any memory operations in that block or below do not incorrectly move up one side or the other of the control-flow merge. Similarly, the phi node of global stores at the top of a loop ensures that memory operations do not incorrectly move out of a loop if the loop has side effects. The need for phi nodes for global stores increases the number of nodes in the SSA graph. However, the extra number of phi nodes is not large, and the advantage is the simplicity that results when memory operations are treated like all other values in the SSA graph.

There are a number of necessary modifications to other types of values to handle stores correctly. Method bodies have a global store as an extra input argument. The **return** value at the end of a method takes two inputs, the actual return value of the method (if there is one) and the latest global store. This adjustment ensures that all stores are correctly kept alive by our dead-code elimination passes. Similarly, a **throw** value takes a value to throw and a global store as input.[1] As described in the next section, method calls take a global store as input and produce a new global store as an output, since any method call may modify global memory.

Java provides constructs for synchronizing on an object, which we model via **monitor_enter** and **monitor_exit** operations. The Java memory model essentially requires that memory operations cannot move above or below these synchronization operations. We easily enforce this requirement in this Swift IR by having both **monitor_enter** and **monitor_exit** take a global store as input and produce a new global store as output. Optimization around reads of volatile fields can be similarly limited by having such reads produce a new global store as output.[2]

One shortcut that we take with the current Swift IR is that operations that store into an object or an array (**put_field**, **put_static**, and **arr_store**) are pinned to their original block. This pinning is not strictly required, since these operations do not cause exceptions. However, a write to memory has a control dependence on the immediately preceding control operation that determines whether it is executed. For example, a write to memory should not be moved up above a preceding IF operation. In the Swift IR, we decided not to represent these control dependences, and instead chose to pin store operations to their original block.

Figure 3 shows a full example of the Swift IR for a method that does writes to global memory. In this figure, we use our typical method for representing the Swift IR. Because the SSA graph can have so many nodes and edges, we give names to all the values and specify the inputs to a value as named arguments, rather than via edges between those values. We have indicated the CFG in the usual manner, with one additional shortcut. We have not shown the exception exit block, and instead indicate that a block has an outgoing edge to the exceptional exit block by shading the block. That is, a block is shaded if it may produce an uncaught exception.

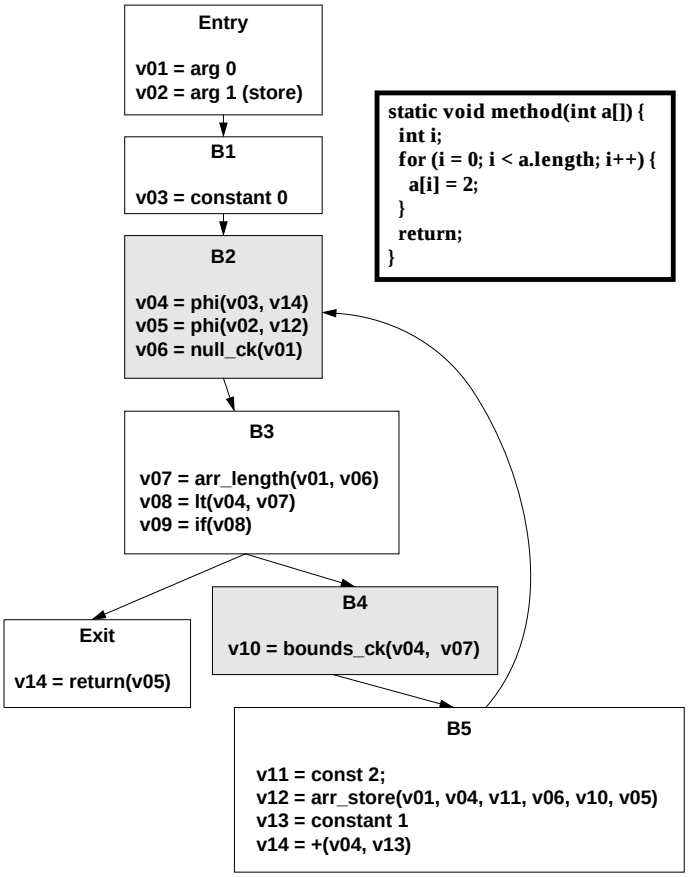

Figure 3: Example of Swift IR for a Full Method

In this example, the entry block contains a value representing the array input argument and a global store input argument. In Block 2, the value **v04** is the phi node required for the loop index `i`, and the value **v05** is the phi node required in the loop for the global store. Because the method has no return value, value **v14** takes only the current global store as an input. Note that some common subexpression elimination has already occurred in this graph, since the **null_ck** and **arr_length** values of the `a.length` expression have already been combined with the corresponding values of the array store.

[1] The exception exit of a method should also contain a return value which takes an exception object and a global store as input. Such a return value would typically require phi nodes with many run-time exception objects and many global stores as inputs. For simplicity, we do not represent this exception return object. This simplification does not greatly affect the compiler, since most of the exception processing is done by the run-time system. The only effect is that we cannot easily do method inlining inside code with an explicit exception handler.

[2] For simplicity, we currently insert a special value before each read of a volatile field that modifies the global store (but generates no actual machine code), instead of creating special versions of the read operations.

2.5 Method Calls

The Swift IR has operations representing the various types of method invocation in Java, including class method calls, instance method calls (either virtual or direct), and interface calls. Class method calls are like normal non-object-oriented procedure calls, since there is no receiving object. Values representing method calls all take the method arguments as inputs, as well as the current global store. In addition, instance method calls take a null check of the receiver object as an extra input, because Java requires that a null check on a receiver object be done before an instance method invocation on that object is done.

Most method calls produce a program result, as well as producing a new global store. In order to follow the rule that each value in the SSA graph produces a single result, method calls produce a tuple, which consists of the method result and the new global store. The two components of the result are accessed by special SELECT values [10], which take a tuple as input and produce a specified component of the tuple as output. Values representing method calls are therefore typically followed by one or two select values in the SSA graph.

If a method call can cause an exception that is caught in the current method, the call actually produces a distinct global store for each normal and exception successor edge, since the state of global memory may be different depending on how the method call completed. In the common case when no exception of the method call is caught, no extra store is required, since the exception exit will not access the global store.

In the representation of a method body, there are corresponding arguments in the entry block for the incoming global store and, if an instance method, the null check on the receiver. In the case of an instance method, the receiver, which is the first argument, is known to be non-null, as indicated by the null-check argument to the method call. When building an SSA graph, the compiler therefore automatically uses the null-check argument when a null check of the receiver is required in the method. The null-check argument does not produce any machine code, so the null checks on the receiver argument have been eliminated.

The above handling of the store and null-check arguments results an especially clean implementation of method inlining. In order to do method inlining, the compiler builds the IR of the original method and the called method. The CFG and SSA graph of the called method are inserted into the CFG and SSA graph of the calling method at the point of the call, with the arguments to the method body of the inlined method replaced by the corresponding arguments to the method call, including the global store and the null check. This matching process establishes all the correct dependences (including those for memory operations and those based on the null check of the receiver object) so that all values that were within the inlined SSA graph can now be moved anywhere allowed by the usual dependence rules.

Note that a method that has no write operations returns its original store argument in the **return** operation. If this method is inlined, then the operations that were below the call to this method will now be using the store from above the method call, and thus optimizations will automatically take advantage of the fact that the inlined method has no side effects.

2.6 Results of the Swift IR

We have found that the design of the Swift IR has worked out especially well, and has allowed us to do simple and general implementations of various optimizations. In this section, we summarize some of the advantages of the IR, as well as some other implications.

First, the rule for determining if values are equivalent, and therefore candidates for common subexpression elimination (CSE) is very simple, yet works for all operations. A general rule in the Swift IR is that values with identical operations (including the auxiliary operation) and equivalent inputs are equivalent. This rule follows because the global store inputs conservatively represent memory dependences, and all other true data dependences are explicit in the inputs to values.[3] With this rule, global CSE can be applied to all values, including, for example, memory reads. Values that load the field of an object will be candidates for CSE only if they access the same field of the same object, *and* they have the same global store input. Because the rule about equivalent values is so simple, it is easy to apply in a limited fashion where useful in many other passes of the compiler. In particular, the Swift compiler does a limited form of CSE while building the SSA graph in order to reduce the number of nodes initially created.

Another advantage of the Swift IR is that almost all data flow and ordering dependences are explicit in the SSA graph. As mentioned in Section 2.4, anti-dependences are not explicitly represented and must be generated during scheduling passes. In addition, we have pinned various operations to their basic blocks so as not to have to represent various control dependences. However, all other values are easily moved around, subject to the straightforward scheduling constraints. Like CSE, scheduling is still simple enough that several passes in Swift do limited forms of scheduling to determine if some other optimization would be useful.

As we explained in Section 2.5, method inlining is quite simple in the Swift IR. In addition, method inlining just produces a slightly larger SSA graph that is like any other graph and only includes the required dependences between values in the inlined method and values in the calling method. All the standard optimizations, such as CSE, code motion, and dead-code elimination, apply after method inlining without any artificial restrictions.

[3] One detail is that the auxiliary field of control operations and phi nodes is set to be the block in which they are located, so that control operations are never combined and phi nodes are only combined if they have identical inputs and are in the same block.

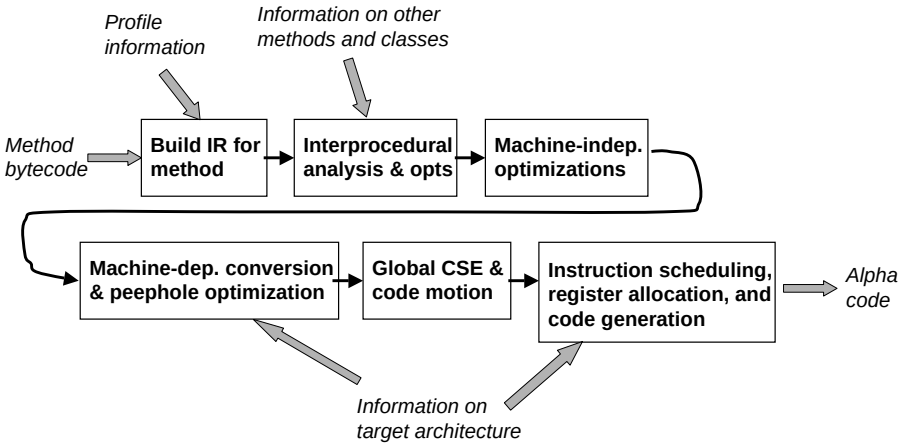

Figure 4: Organization of the Swift Compiler

One property of SSA that became clear to us only after using it in the Swift compiler is that SSA improves the effectiveness of peephole optimizations. SSA form exposes all the users of a value, rather than having some limited window. The Swift compiler therefore applies pattern-based peephole optimizations, where the peephole is essentially the whole method, and values in the pattern can be arbitrarily far apart in the control flow of the method. For example, one peephole optimization says that a length operation applied to an array allocated in the method can be replaced by the size argument to the allocation operation. This optimization can be applied no matter where the allocation and length operations are in the method. In fact, it is frequently useful for the case when an array is allocated at the beginning of a method, and length operations, often generated because of required bounds checks, occur throughout the method.

We should note that the use of SSA form does put some constraints on the Swift compiler. First, and most importantly, the register allocator must be quite effective and general-purpose, since it must assign the numerous values of the SSA graph to registers and stack locations with minimal copies and spills. Otherwise, the optimization advantages of SSA may be outweighed by the cost of copying and spilling values. Similarly, the passes that do the final placement of values in blocks and scheduling within blocks are quite important. The use of SSA form provides much freedom in placing values into a final schedule, so the scheduling passes must make effective use of available information in computing a good schedule.

Finally, some groups [22] have noted that SSA form is difficult to use for restructuring optimizations, since it may be difficult to add required phi nodes correctly as code is restructured and new join points are created. We have not encountered any difficulty with keeping an SSA graph correctly structured for almost all of our optimizations, including many that modify the CFG, such as branch removal, method inlining, method splitting, conditional constant propagation, etc. The only optimization that we found difficult in SSA form was loop peeling (see Section 3.3.5). However, other restructuring optimizations, such as loop unrolling or loop interchange, might also be complex in SSA form.

3 The Organization of Swift and its Optimizations

In this section, we describe the basic organization of the Swift compiler, in terms of the sequence of passes that convert Java bytecode to Alpha machine code, and then we describe individual passes in more detail. Figure 4 shows the overall high-level flow of the compiler. The IR for a method is built based on the bytecode of the method, and any available profile information is used to annotate the CFG. A variety of interprocedural optimizations are applied, which may require accessing information about other methods and classes. These interprocedural optimizations, such as method inlining, can expose many opportunities for later optimizations. Then, a variety of machine-independent optimizations are applied. A tree-matching algorithm is then used to convert many operations to a lower and/or machine-dependent form and to do peephole optimizations. Some further optimizations are applied to the machine-dependent form, mainly global CSE and code motion, to take advantage of opportunities exposed by the machine-dependent form. Finally, actual Alpha code is generated by a sequence of passes that do instruction scheduling, register allocation, and code generation. All of these passes operate on the representation of the method in Swift IR. Figure 5 provides a more detailed list of the analyses and optimizations that occur during compilation and will be discussed below.

Interprocedural analyses
- alias analysis
- class hierarchy analysis
- escape analysis
- field analysis
- type propagation

Interprocedural optimizations
- method resolution
- method inlining
- method splitting
- object inlining
- stack allocation
- synchronization removal

Intraprocedural optimizations
- bounds check removal
- branch removal
- conditional constant propagation
- dead code elimination
- global CSE
- global code motion
- loop peeling
- null check removal
- peephole optimizations
- strength reduction
- type test elimination

Machine-dependent passes
- conversion to machine-dependent operations
- peephole optimizations
- sign-extension elimination
- trace scheduling
- register allocation
  - live range splitting
  - biased graph coloring with rematerialization
- block layout
- code generation

Figure 5: Analyses and Optimizations of Swift Compiler

3.1 Building the IR

Swift generates the IR from a method's bytecode in a manner similar to other Java compilers that use SSA form for their intermediate representation. First, the bytecode is scanned to determine the number of basic blocks and the edges between the basic blocks. Then, a phi node placement algorithm [30] is executed to determine which local variables of the JVM require phi nodes in the various basic blocks. Then the bytecodes of each of the basic blocks are executed via abstract interpretation, starting with the initial state of the local variables on method entry. The abstract interpretation maintains an association between JVM local variables and SSA values, determines the appropriate inputs for new values, and builds the SSA graph.

Swift does numerous simple optimizations during creation of the SSA graph in order to reduce the number of nodes created. As described above, it can replace an **arr_length** operation with the allocated size of an array, if the array was allocated in the current method. It can also eliminate bounds checks if the index and array length are constant. These optimizations are especially important in methods (such as class initialization methods) which initialize large constant-sized arrays.

Swift can also take advantage of profile information that was produced by previous runs. Based on the profile information, Swift annotates the edges of the CFG indicating their relative execution frequency. This profile information is mainly used for decisions about code layout and the choosing of traces by the trace scheduler. If no profile information is available, Swift guesses reasonable execution frequencies based on the loop structure of the CFG.

Swift maintains the invariant that there are no critical edges in the CFG. A *critical edge* is an edge whose source is a block with multiple successors and whose destination is a block with multiple predecessors. Critical edges must be eliminated in order to facilitate the first phase of register allocation, which, as described in Section 3.4.3, places copy operations at the inputs of phi nodes. In addition, the removal of critical edges ensures that global code motion (see Section 3.3.2) has the greatest freedom in placing values so as to minimize unnecessary computations in the final code. A critical edge can be eliminated by introducing an empty block between the source and destination of the edge. Swift eliminates all critical edges immediately after building the CFG, and later phases that modify the CFG are written to ensure that they do not create critical edges.

3.2 Interprocedural Analysis and Optimizations

The Swift compiler does extensive interprocedural analysis and numerous interprocedural optimizations, many of which can individually be turned on and off. The interprocedural analyses include class hierarchy analysis (CHA), type propagation, type- and field-based alias analysis, escape analysis, and a new form of analysis called *field analysis* [24]. Some of the associated optimizations include method resolution, method inlining, elimination of run-time checks, removal of unnecessary data dependence edges, stack allocation, and synchronization removal.

Several interprocedural optimizations are combined into a single pass and are applied repeatedly, because each optimization can enable further optimizations. These optimizations include method resolution, method inlining, type propagation, and determining types based on field analysis. For example, the inlining of one method can clarify the type of a receiver of another method call, which may enable that method call to be resolved and possibly inlined as well.

In the following sections, we describe the various interprocedural analyses done by Swift, and their use in various optimizations.

3.2.1 Class and Method Analysis

Swift contains modules that compute and cache useful pieces of information about classes and methods. By default, these

modules operate on demand and analyze classes and methods only when requested because Swift encounters an opportunity for a possible optimization.

The basic information maintained about classes is the class hierarchy, as determined by the subclassing relationship. By default, this hierarchy is built only as necessary to help resolve method calls. However, Swift can optionally use class-hierarchy analysis (CHA) in resolving method calls. CHA assumes that the set of classes is fixed at compile time, so the entire class hierarchy is known by the compiler [17]. With this assumption, many virtual method calls can be resolved, because the compiler can determine that there is only one possible implementation of a specified method in a class or all of its known subclasses. If CHA is turned on, Swift loads information on all the known classes and builds a representation of the class hierarchy. It can then easily scan all the subclasses of any given class. If CHA is not being used, Swift only assumes that it knows all the subclasses of a class if it is marked `final`.

Swift also maintains a variety of information about methods in a hash table. Simple examples include the size of the method's bytecode (for use in deciding whether to inline a call) and whether the method is reimplemented in a subclass (for use in resolving a method call). In addition, a method entry can store many other useful properties which can only be determined by examining the method bytecode. For example, Swift can determine, when useful for optimization, if a method is guaranteed to return a non-null result. These properties are typically determined on demand by building the Swift IR for the method and then examining the resulting graphs. The use of SSA in the Swift IR naturally makes these properties somewhat flow-sensitive.

3.2.2 Type Propagation

Type propagation is useful for resolving some virtual method calls, especially when CHA is not being used. As we mentioned above, Swift assigns types to all values based on available information in the bytecode and SSA graph. Some values have very specific types. For example, a value that allocates a new object of a particular class C is known to have an exact type of class C, not any subclass of C, even though the static type in the original Java program indicates C or any subclass. Type propagation ensures that this exact information is recorded in the type field of applicable values. The use of SSA form makes the type propagation flow-sensitive, since types are merged correctly at control-flow joins. Given the type propagation, Swift can resolve a virtual method call directly if the receiver object of the call has an exact type.

Exact types can be determined in several ways. First, as mentioned above, exact types are known when an object or array is directly allocated. Second, one of the properties of methods that Swift can compute on demand is whether the method returns an object with an exact type. Third, Swift can use field analysis, as described in the next section, to determine if a load from a field of an object always returns an object with an exact type.

```
public class Plane {
  private Point[] points;

  public Plane() {
    points = new Point[3];
  }

  public void SetPoint(Point p, int i) {
    points[i] = p;
  }

  public Point GetPoint(int i) {
    return points[i];
  }
}
```

Figure 6: Example Class for Field Properties

3.2.3 Field Analysis

Field analysis is an inexpensive kind of interprocedural analysis that determines useful properties of a field of an object by scanning the code that could possibly access that field, as dictated by language access rules. For example, a `private` field in Java can only be accessed by methods in the local class, while a `package` field can only be accessed by methods in classes that are in the same package as the containing class.[4] An example of a field property is more exact information on the type of a field. Field analysis can often show that a field, if non-null, only references objects of class C, and never objects of any of C's subclasses. Similarly, field analysis can often prove that a field, if non-null, always contains an array of a particular constant size.

As an example, consider the code in Figure 6. Because the field `points` is `private`, the compiler only needs to scan the instance methods in class `Plane` to determine its properties. The compiler can prove that `points`, when non-null, must reference an array with base-type `Point` and a fixed size of 3. In addition, the compiler knows that `points` is non-null anywhere outside `Plane`'s constructor.

As mentioned above, Swift uses exact type information from field analysis to help resolve method calls. Information about the size of fixed-sized arrays is used to simplify or eliminate bounds-check computations. Null checks can potentially be eliminated for fields that are known to be non-null. While Swift (like most Java systems) uses page protection to implement null checks without any extra code, eliminating the null check is still useful because it gives the compiler more flexibility in code motion. As with the class and method analysis modules, a property of a field is computed on demand only if required for a potential optimization. For more details on field analysis, see [24].

[4] A Java field without any access modifiers is visible to its entire package and therefore we call it a `package` field.

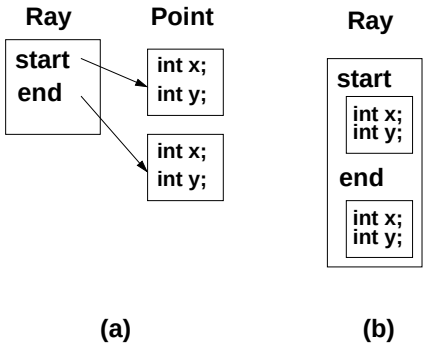

Figure 7: Example of Object Inlining

We have also used the field analysis approach to determine fields that are candidates for *object inlining*. Object inlining [21] is a method of reducing the overhead of accessing objects by allocating the storage of an object within its containing object. If object B is inlined in object A, the cost for accessing object B is reduced by a pointer dereference. Objects A and B are likely to be frequently accessed together, so cache locality may be improved by inlining if A and B are now stored in the same cache line. As an example of object inlining, Figure 7(a) shows the storage layout for a `Ray` object which references two `Point` objects. Figure 7(b) shows the storage layout when the two `Point` objects are inlined into the `Ray` object.

Our analysis is unique in that it finds both objects that can be inlined without a header for the inlined object, and objects that can be inlined, but require with an object header. The object header contains an indication of the type of the object and its method table, and is also typically used for synchronizing on the object. The header information is therefore needed for many operations on the object, including virtual method invocations, synchronization, and type-inclusion checks. If an object is inlined with a header, then a reference to the object can be allowed to escape into the heap. We are not aware of any other implementations of object inlining that allow references to inlined objects to be stored in the heap.

Our field analysis finds fields whose contents can always be inlined. The basic criteria is that a new object of particular class must be allocated and assigned to the field in all constructors for the containing class, and the field must not be reassigned by any other method that can access the field. In addition, the inlined object must include its header, if an operation that might require the header may be executed or if a reference to the object might be stored into the heap.[5] For more details on our analysis and implementation of object inlining, see [24].

[5]For the purposes of object inlining, we use very quick and simple tests to determine whether the object might be stored into the heap, in contrast to the more general escape analysis described in the next section.

3.2.4 Escape Analysis

Escape analysis is used to determine if a reference to an object escapes a thread (i.e. can be accessed by another thread) or a particular method call (i.e. can still be accessed by the current thread after the method call completes). Escape analysis is a necessary component for determining if an object can be allocated on the stack, rather than the heap. If a reference to an object does not escape a particular method call (along with a few other conditions), then the object can be allocated on the stack frame of that call. Escape analysis can also be used for eliminating or reducing the cost of unnecessary synchronization. If a reference to an object does not escape a thread, then synchronization on the object is unnecessary.[6]

Our basic escape analysis assumes that an object escapes if a reference to the object is ever stored into a global variable or a heap object (including arrays). We can then use a simple dataflow analysis [23] to determine if any value in an SSA graph escapes. To increase the precision, we make use of summary information about the effects of called methods. Our method analysis module does an interprocedural dataflow analysis on demand to determine if a method may store or return its arguments, and if it definitely returns an unaliased object (i.e. a new object that has not already been stored). If a method call is not resolvable or analyzable, the conservative assumption is made that the method call returns and stores all parameters and does not return an unaliased object.

We have also extended the simple analysis to take advantage of some information available from field analysis. The main idea is to discover fields which are *encapsulated*, in the sense that the field is initialized with a reference to a new object by methods in the object's class or package, can only be accessed by these methods, and and is never "leaked" by these methods. If we discover an object that does not escape, then the contents of any encapsulated fields of that object do not escape, and so on, recursively.

Given the above escape analysis, stack allocation and synchronization removal are implemented fairly easily. While compiling a method, Swift looks for candidate values, which are either objects that are allocated directly in the method, or are newly allocated, unaliased objects returned by method calls. Swift then reduces its list to the candidates which it can prove do not escape the current method. There are also additional restrictions for objects that will be stack allocated, such as that each object must have a known, exact type and arrays must have small, constant lengths. If an object can be stack allocated, then an operation to allocate the object on the stack is added to the current SSA graph, and the existing

[6]Under the current Java memory model, the synchronization cannot necessarily be removed, since the use of synchronization must cause all updates by the local processor to be committed and all updates by other processors to be seen. However, the Java memory model will likely be revised so that synchronization only has such effects for accesses that are connected by chains of synchronizations on the same objects [1].

```
class Example {
  int A[] = new int[10];
  int B[] = new int[10];

  public void method() {
    for (int i = 0; i < 10; i++)
      A[i] = 0;
      B[i] = 1;
  }
}
```

Figure 8: Code with Data Dependences between Memory Operations

allocation operation is removed. If the object was allocated by a called method, then another version of that method is generated which initializes the object allocated on the stack, rather than creating its own object. If Swift determines that synchronization on an object can be removed, then it scans all uses of the object and removes any synchronization operations. This process may also result in new, unsynchronized versions of called methods being created.

3.2.5 Alias Analysis

Swift includes a phase which does a form of *alias analysis* and attempts to relax some of the data dependences in the SSA graph so as to allow further optimizations. This phase tries to eliminate data dependences between two memory operations when it can prove that the accessed locations are not *aliased*, i.e. cannot be the same. As an example, consider the code in Figure 8. The loop contains two array store operations, so its representation in Swift IR requires a phi node at the top of the loop merging the initial global store entering the loop and the global store at the end of each loop iteration. The **get_field** operations that load the values of A and B take the global stores produced inside the loop as inputs, and cannot move above the loop. The generated code will therefore reload the values of A and B in each iteration of the loop.

Clearly, though, from Java's type rules, the values of A and B cannot be modified by stores to integer arrays, so the loads of A and B should be moved out of the loop. In general, a memory operation that accesses a particular field or an array element of a particular type can only be affected by memory operations that access the same field or same type of array element. We would like a way to represent these reduced dependences among memory operations caused by Java semantics, while still enforcing other required dependences.

One possible way of representing dependences among memory operations with more precision is to have many different types of "global stores". For example, we could have a separate global store for each distinct field and each type of array element. A write to field f of an object would modify the "f" global store, but not any other global store. Similarly, a read of field g of an object would only take the "g" global store as input, and therefore would not be affected by a preceding write to field f.

Store operation v of LocType L	⇒	update input mapping with (L, v)
Method call v with known store effects on L1, L2, etc...	⇒	update mapping with (L1, v), (L2, v), etc...
Method call v with unknown effects	⇒	change mapping to (L,v) for all L
Lock or unlock operation v	⇒	change mapping to (L, v) for all L
Phi node v	⇒	for any L that maps to u on all inputs, use (L, u) else use (L, v)

Figure 9: Transfer Function for Store Resolution

There are several problems with this approach. First, there would be many more edges and nodes in the SSA graph. Extra phi nodes would now possibly be required for many different kinds of global stores, and a method body would now require **arg** nodes for many kinds of global stores. If the effects of a method call are unknown, then it would have to take every possible kind of global store as input, and produce new versions as output. The same requirement would also apply for synchronization nodes. Also, method inlining would become much more complicated, and might require that new global store **arg** nodes for the caller method be created to represent the effects of the called method.

We have chosen to stay with the basic Swift IR, but to use a process which we call *store resolution* to relax memory dependences by changing the global store inputs of some memory operations. For example, in Figure 8, the desired effect can be achieved by changing the store inputs of the loads of A and B to be the original store input of the method, rather than one of the global stores in the loop. The loads of A and B can then be moved out of the loop.

Store resolution first does a forward dataflow computation to build up information about memory operations, as follows. Let us call a value a "StoreOp" if it produces a new global store. StoreOps are either writes to memory, method calls, or phi nodes. The idea is to look at the subgraph of the SSA graph which consists of StoreOps. We want to compute information at these nodes about sets of locations that are easy to classify. As we mentioned above, in Java we can easily distinguish stores to particular fields of various types, and also to array elements of particular types. Let us refer to these different sets of locations as LocTypes. Then, we wish to compute information at each node of the subgraph that indicates, for each LocType, the most recent StoreOp that might have modified locations of that type.

In our dataflow analysis, the state at each node is conceptually a mapping from each LocType to the most recent preceding StoreOp which might have modified locations of that LocType. For implementation efficiency, our state has one entry indicating the default StoreOp that applies to most LocTypes, and then individual ordered pairs (LocType, StoreOp) for any LocTypes that don't map to the default StoreOp. The forward transfer function for the dataflow problem is shown in Figure 9. Because our method is a full dataflow analysis, it does correct summaries for loops. In particular, it will correctly determine when a particular LocType cannot be mod-

ified in a loop.

Once we have obtained the state information for each node, store resolution then uses this information to update dependences based on this information. In particular, we examine each load operation in the method. Each load operation has a particular global store G as input. We use the state information at G to determine the actual global store H that the load operation may depend on, based on the LocType of the load. If G is different from H, than we change the load operation to use H as its input rather than G.[7]

By changing the store input, store resolution makes the relaxed dependence information available to all later passes. For example, if there are two identical load operations and store resolution changes the store input of one to be the same as the other, then a later CSE pass will naturally combine the loads into a single operation. Similarly, a load operation will automatically be scheduled earlier if possible, and may be moved out of a loop.

Our analysis is interprocedural, since it makes use of summaries of the effects of methods. As with other method properties, the write effects of methods are computed on demand by the method analysis module. The summary is a flow-insensitive list of fields and array element types (LocTypes) that the method (and all methods that it might call) might modify. Unlike many other properties, the immediate effects of a method can be determined by examining the bytecode without building a CFG or SSA graph.

By categorizing locations into distinct field and array element types, we have used a type-based alias analysis scheme (where the field name of a location is also considered part of its type). However, store resolution can easily be used with a more sophisticated alias analysis technique which manages different kinds of LocTypes and produces more precision.

3.2.6 Method Resolution and Inlining

As described in the above sections, Swift resolves virtual method calls using information from various kinds of analyses. If CHA is being used, then method information may indicate that there is only one possible implementation of that could be referenced by a virtual call, given the static type of the receiver. If CHA is not being used, Swift can still determine that there is only one implementation if certain classes or methods are declared as `final`. Alternatively, type propagation (possibly with help from field analysis) may indicate that the receiver must have a particular exact type and therefore can only invoke a specific method. Information on exact object types is also used to resolve interface calls.

We already described the basic implementation of method inlining in Section 2.5. If inlining is turned on, Swift inlines a method if the method call can be resolved and the size of the method is below a certain threshold. Together with the other interprocedural optimizations, method inlining is applied recursively a small number of times. Recursive inlining is especially important for eliminating the costs of calling superclass constructor methods, most of which may be empty.

If Swift is being used in an environment where dynamic loading can occur, Swift can limit its use of CHA to simplify correctness issues. In general, if a method M is compiled with information that a particular class has no subclasses or a particular method has only a single implementation below a certain class, then M's code must be modified if the dynamic loading of code means that such information is no longer true. However, it can sometimes be difficult to modify existing code. The problem is that the method M that we want to modify may actually be running or active (on the stack) at the time. In fact, M may be executing an infinite loop or a long-running computation (e.g. reading input from a large file). Therefore, we cannot simply "wait" until M is no longer running to modify it.

If M has a virtual method call C that is resolved to a direct call using CHA, but the call is not inlined, then it is easy to modify M to turn off the CHA optimization, even if M is running. The JVM can generate a stub that does the virtual call instead, and then atomically change the call C to jump to the stub. A more difficult problem is to reverse the effects of CHA when the virtual method call C has been resolved and inlined. However, as observed in [18], the resolved call C only becomes incorrect if its receiver can possibly contain a new object from the dynamically loaded class. If the receiver of C is an argument of the containing method M, then the receiver existed before the current entry to M and cannot change. Therefore, the receiver cannot reference an object of the dynamically loaded class, so the resolution of C will be correct at least within the current invocation of M. So, M can be invalidated simply by creating a new version of M with the virtual call reinstated, and ensuring that any future calls to M use the new version. Alternatively, instead of generating new compiled code, the JVM can just ensure that future calls to M revert to interpreting the bytecode directly or using less-optimized JIT code.

When Swift is invoked in specified mode (which we call simple-CHA), it will limit its use of CHA as follow. Swift will not use CHA to resolve methods for use by any interprocedural analysis, such as alias analysis or escape analysis. Swift will only use CHA to resolve virtual method calls in the method being compiled and convert them to direct calls. In addition, Swift will not generally do method inlining on a method call that was resolved by CHA. However, it will allow method inlining if the receiver of the call is an input argument of the containing method. Given these limitations, the JVM can readily update code optimized using CHA when a class is dynamically loaded. Swift simply needs to provide the JVM with a list, for each compiled method M, of the classes which, if subclassed by a dynamically loaded class, may require M to be modified or invalidated. Along with each class, Swift should indicate which new method imple-

[7] Swift maintains a record of the original store input G, so that the appropriate anti-dependences can be calculated in the scheduling phases, as described in Section 2.4.

mentations would require a modification of M, and, for each of these, whether a particular direct call can be modified or if M's code must be invalidated or recompiled.

3.2.7 Method Splitting

Object-oriented programs frequently have small methods that access an encapsulated field or array element in an object. Method inlining naturally reduces the overhead of these small methods, by eliminating the method call overhead and allowing further opportunities for optimizations. However, sometime these methods contain one or more quick tests for unusual cases before executing a small amount of code in the common case. In the unusual case, these methods may do much more computation or throw an exception. Important examples include methods in the standard Java library such as `java.util.Vector.elementAt` and `java.lang.String.charAt`. A problem with these kinds of methods is that they will not be inlined, because of the code that handles the unusual cases. In addition, when they are compiled as a separate routine, there will likely be extra register saves and restores, because of the extra code.

Swift addresses this problem by checking if it would be beneficial to split up a method. In general, a method should be split if it has only a small amount of code that is commonly executed, as defined by profile information or static estimates. Currently, Swift defines the "splittable" property much more conservatively, in order to simplify the implementation of method splitting. A method is splittable only if it has a single path to the return statement that calls no other methods, as well multiple other paths that eventually throw exceptions. In such cases, the path to the normal exit is the common case.

The information about whether a method is splittable is another property that is computed and cached by the method analysis module. When Swift compiles a splittable method, it will create auxiliary methods that execute the code along the abnormal paths that throw exceptions. Swift then modifies the IR of the original method by replacing the abnormal paths with calls to the auxiliary methods. In addition, if a method calls a splittable method, then Swift only inlines the "split" version of the method with calls to the auxiliary methods.

3.3 Intraprocedural Optimizations

Once all the interprocedural analysis and associated optimizations have been done, Swift does all the standard intraprocedural optimizations on the existing IR, as well as a number of optimizations that are more Java-specific. In this section, we describe some of the interesting aspects of these optimizations and their implementation in the Swift IR.

3.3.1 Global Common Subexpression Elimination

By default, Swift does global common-subexpression elimination (CSE) and global code motion, as described by Click [11]. That is, by default, Swift will apply CSE to any two values that are equivalent, no matter where they occur in the control-flow graph. In particular, Swift will replace one value by an equivalent value, even if neither value dominates the other. In this case, the resulting value does not necessarily dominate all its users. Global CSE must therefore be followed by a pass of global code motion, which places values in blocks so that all values dominate all their users.

As described in Section 2.6, two values are equivalent if and only if they have identical operations (including the auxiliary operation) and have equivalent inputs. During global CSE, Swift computes equivalent values via a partitioning algorithm [3, 11]. It splits all values into initial partitions based solely on their operation fields and puts the partitions on a work-list. Each time it takes a partition P off the work-list, it builds the set of values S_i which take one of the values in the partition as the i^{th} input, as i ranges over the possible input numbers. If there is a partition Q which has some set of its values Q', but not all its values, in S_i, then Q is split up into Q' and $Q - Q'$, and the smaller of the two resulting partitions is added to the worklist. Eventually, this partitioning algorithm terminates, and all values in a partition are equivalent.

We make one modification to the partitioning algorithm to deal with values that cause exceptions. Exception-causing values cannot be moved from their original block, so we cannot use code motion to ensure that these values dominate their uses after CSE is applied. So, if there are two identical exception-causing values, the first value can only replace the second if the first value's block dominates the second value's block. In addition, the second value is redundant only if the first value definitely did not throw an exception. So, Swift also requires that the second value's block is dominated by the non-exception successor of the first value's block.

To satisfy these conditions, we modify the CSE algorithm as follows. Our rule is to keep the dominating value of a partition of exception-causing values as the first element of the partition. Every time that the above partitioning algorithm reaches quiescence, we find the value V in each partition with exception-causing values that is at the highest level of the dominator tree. We make V be the first value of the partition and test that V dominates the rest, according to the above condition. If not, then we break up the partition into those dominated by V and those that are not. The smaller of the two partitions is added to the worklist, and we again run the partitioning to quiescence, and so on. Eventually, all partitions with exception-causing values will have their first value dominating the rest.

Once Swift has computed the final partitions of the values, it iterates through all the partitions. If a partition has more than one element, Swift chooses a representative value from the partition, which must be the value that dominates the oth-

ers in the case of exception-causing values. It then changes all users of a value in the partition to use the representative value. All of the other values in the partition are no longer used and are removed.

Global CSE is highly effective at removing many run-time checks. For example, many null checks of the same input argument or local variable will be eliminated automatically, as long as one of the null checks dominates the others. Whenever a run-time check is eliminated via CSE, Swift automatically removes the exception edge of the containing block and merges the block with the successor block. Thus, the elimination of redundant run-time checks greatly compresses the CFG as well.

Note that CSE also applies to loads of fields or array elements. When preceded by the store resolution phase that makes alias information available by relaxing memory dependences, global CSE automatically accomplishes the removal of redundant loads, which frequently requires a separate pass in other intermediate representations. [14].

Swift currently does global CSE (and global code motion) once during the machine-independent processing, after all interprocedural optimizations. This CSE pass eliminates redundant expressions that are in the original method or that are exposed by interprocedural optimizations, and can greatly reduce the size of the SSA graph. Swift does another round of global CSE (and global code motion) after the conversion to machine-dependent IR, in order to eliminate common expressions revealed by the conversion to the lower-level form.

3.3.2 Global Code Motion

Swift currently uses Click's global code motion algorithm [11]. The exact choice of a code motion algorithm is not very important, since Swift has a final scheduling pass which can move values over traces consisting of multiple basic blocks. The main purpose of the code motion pass is to move loop-independent values outside of loops and to ensure that all values are in blocks that dominate their users after global CSE.

Click's code motion algorithm proceeds via two passes over the SSA graph. The first pass finds the earliest possible placement for all values by requiring only that a value be placed such that it is dominated by all its inputs. The second pass finds the latest possible placement for all values by requiring only that a value be placed in a position that dominates all its users (and also respects any necessary anti-dependences, as described in Section 2.4). For any particular value, the early placement will dominate the late value. After determining the latest placement for the value in the second pass, the algorithm scans the sequence of blocks from the early to the late placement of a value. It then places the value in the latest possible block (i.e. lowest in the dominator tree) which is at the lowest loop nesting. The result is that a value is moved out of loops as much as possible, and is otherwise put in the most control-dependent block.

```
public boolean equals(Object o) {
  if (this == o)
    return true;
  ValueVector vv = (ValueVector) o;
  if (ptr != vv.ptr)
    return false;
  for (int i=0; i< ptr; i++)
    if (!v[i].equals(vv.v[i]))
      return false;
  return true;
}
```

Figure 10: Example of Elimination of Duplicate Inputs to Phi Nodes

Immediately before code motion, Swift does a quick pass to try to eliminate duplicate inputs to phi nodes. The motivation is for code such as the method from a SpecJVM98 benchmark shown in Figure 10. The SSA graph for the method has a phi node in the final block that merges all the possible return values. This phi node has two inputs which are the value 'true' (actually just 1) and two inputs which have the value 'false' (actually just 0). The generated code has two sequences that load zero and jump to the epilog, and two sequences that load 1 and jump to the epilog. (Even worse, the constant 1 is sometimes put in a different register and hoisted to the dominating block of the branches, resulting in greater register pressure.)

To get rid of these duplicate code sequences, Swift scans blocks for phi nodes with duplicated inputs. If such a phi node is found, and the only other phi node in the block is for global stores, then the compiler eliminates the duplicate inputs by inserting an extra predecessor block that joins all the blocks corresponding to the duplicate inputs. (The new block may need to have a phi node to merge the store inputs for these blocks, but this doesn't not actually generate any code.) The result of the optimization is that the register copies for the duplicated phi inputs will be merged into a single code sequence, and the duplicated value can now possibly move down into the new block.

So, in the example in Figure 10, there will be just one code sequence that loads one and returns, and one code sequence that loads zero and returns. This optimization occurs most frequently with return values, but is applied when appropriate for any phi nodes with duplicated inputs.

3.3.3 Branch Removal

Swift has a phase that uses two methods to attempt to remove branches. The first method tries to remove branches that are used just to compute a value that is either 0 or 1. Such branches often occur in the computation of boolean values, since the Java virtual machine represents **false** with a value of 0 and **true** with a value of 1. However, the Java virtual machine does not have any bytecodes that produce a boolean value directly from a comparison, so boolean values must be

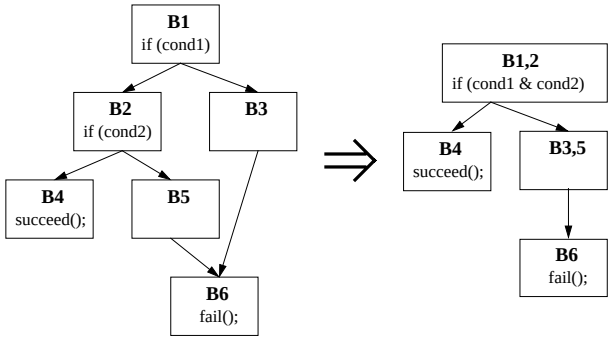

Figure 11: Branch Removal Transformation

computed via a comparison branch whose two destinations assign a 0 or a 1 into a local variable. The goal of the optimization is to convert the branch and associated phi node into a single comparison operator that produces a boolean directly. The actual optimization is fairly simple. Swift simply searches through all the phi nodes in the method, looking for phi nodes whose two inputs are constants of 0 and 1. If such a phi node is associated with a simple IF-THEN-ELSE control flow, then Swift can replace uses of the phi node with the controlling boolean of the **if** node (or its negation). If the IF-THEN-ELSE becomes useless when the phi node is optimized away, then the **if** node and the associated control-flow structure are also removed.

Swift also tries to remove branches by converting logical AND/OR expressions to bitwise AND/OR expressions. For example, the code for a conditional such as `if (cond1 && cond2)` normally requires two branches, since the semantics of the logical-AND operator in Java requires that the second condition is not executed if the first one fails. However, if the second test is fairly simple and has no side effects, then code that executes both tests, combines the result (via a bitwise-AND), and then does a conditional branch will perform better than code that does two conditional branches. The reduced number of branches is especially important for out-of-order processors, since extra branches can reduce the rate of instruction fetching and also may result in speculation down a wrong path.

This optimization is also fairly simple to implement in the Swift IR. This transformation is illustrated in Figure 11. Swift simply looks through all blocks in the CFG for a block that is controlled by an **if** node and one of whose successors is also a block controlled by an **if**. If a candidate block is found, then Swift checks other criteria. In general, the blocks involved (**B1**, **B2**, **B3**, and **B4** in the figure) must contain a small number of operations and must not contain any memory write operations, method calls, or other exception-causing operations. If the criteria are met, then the conditions of the **if**s in **B1** and **B2** are combined (with an **and** or **or** as appropriate), **B2** is eliminated, and the contents of **B5** are merged into **B3**.

```
public static int sum(int a[]) {
  int r = 0;
  for (int i = 0; i < a.length; i++)
    r += a[i];
  return r;
}
```

Figure 12: Example of Bounds check Elimination

3.3.4 Run-time Check Elimination

Swift includes a simple pass that attempts to eliminate some run-time checks or type tests based on properties of values that can be proved from the control flow of a method. For example, if a method contains an IF statement checking if a value is non-null, then null checks can be eliminated in one branch of the IF, because the value is known to be non-null.

Swift currently does a fast, simple analysis, rather than trying to prove properties using general theorem proving or general manipulation of inequalities. The basic technique is as follows. Swift scans the current SSA graph for run-time checks that have not been eliminated. Suppose it encounters a null check of a value `v`. Swift then examines all the users of `v`, i.e. values that take v as an input. Swift searches for a user that is a comparison of v with `null`, such that the comparison is used by an **if** value. Then v is known to be non-null on one of the outgoing edges of the IF block. If the successor block S along that edge dominates the block containing the null check, then the null check can be eliminated. However, values that depend on that null check must not float above S. So, Swift places a special **pin** value in block S and changes all users of the null check to use the **pin** value instead. **pin** values do not generate any machine code, but, as their name suggests, they are pinned to their original block. Therefore, values that depend on the null check will stay within the correct branch of the IF statement. A similar process applies for cast checks. In the case of a cast check of value `v`, Swift scans the users of `v` for instanceof checks on `v` that control an IF. Redundant IFs are similarly eliminated by searching for a dominating IF with the same controlling condition.

Swift does somewhat more work to try to prove that a bounds checks will always succeed. Suppose Swift is trying to eliminate a bounds check whose index is value `v`. Again, Swift scans the users of `v` to derive conditions on `v`. If it finds a comparison on `v` that controls an IF, then it may know that the comparison or its negation is true. Alternatively, if `v` is found to be an induction variable of a loop that is monotonically increasing/decreasing, then `v` is guaranteed to be greater/less than its starting value. For example, consider the code in Figure 12. The for-loop termination condition guarantees that `i` is less than `a.length` anywhere inside the loop body. Also, since `i` is a monotonically increasing induction variable that starts at zero, it is guaranteed to be greater than or equal to zero. Together, these two properties guarantee that the bounds check associated with accessing `a[i]` will always succeed, so Swift will remove it.

As we said above, Swift does not attempt to solve a general system of inequalities involving `i` in order to prove properties about `i`. It simply looks at all the users of the value representing `i`, and determines if any direct comparisons of `i` can be shown to be true. It also determines if `i` is an induction variable, and, if so, whether another condition about `i` can be shown. Swift will apply the condition gathering recursively down to a fixed depth, so it can show that `x` is greater than zero if it shows that `x` is greater than `y`, and it shows that `y` is greater than zero. Swift also incorporates the knowledge of a few algebraic rules, such as that adding two values greater than zero gives a result that is greater than zero.

Our simple approach is effective because our IR provides immediate access to all the users of a value `v`, and therefore makes it easy to find program expressions that help prove properties about `v`. In addition, Swift does a round of global CSE before doing check removal, so Swift has already combined values equivalent to `v` and collected all their uses together.

Storing to an array in Java also requires a run-time check, called an *array store check*, to ensure that the object being stored into the array is compatible with the base type of the array. Swift attempts to eliminate array store checks via several methods. However, these tests for removal are actually done during the conversion of array operations to machine-dependent form. Swift will eliminate an array store check if the base type of the array is exactly known and can hold the known type of the stored value. It will also eliminate the store check if the stored value is known to be null or has been loaded from another element of the same array. Finally, Swift also checks if the class of the base type of the array has no subclasses (either based on the use of `final` or via CHA.). If so, the array store check cannot fail and is eliminated.

3.3.5 Loop Peeling

Loop peeling is an optimization that "peels" off one or more of the initial iterations of a loop into straight-line code preceding the loop. Loop peeling has been used for optimizing loops in scientific programs that have a special condition associated with the initial iteration(s) because of cylindrical boundary conditions. However, we have adapted loop peeling for use in a Java compiler to aid in optimizing loops which contain run-time checks. The problem that we wish to solve is that there are often loop-invariant run-time checks in a loop. For example, there may be a required null check for an array that is first accessed inside the loop. We would like to move such a check outside the loop, but the check is pinned, since it may cause an exception that should not be thrown if no iterations of the loop are executed. However, once the check has succeeded in the first iteration of the loop, it is redundant, since it will succeed in all later iterations. Swift therefore uses loop peeling to separate out part of the first iteration. Global CSE will then automatically eliminate the run-time check still in the loop as redundant. Note that the Swift compiler only does loop peeling to move a check out of a loop if the check has not otherwise been eliminated by field analysis, CSE, or the check removal described in Section 3.3.4. Interestingly, in simple cases, the end result often looks like standard code motion of the run-time check out of the loop.

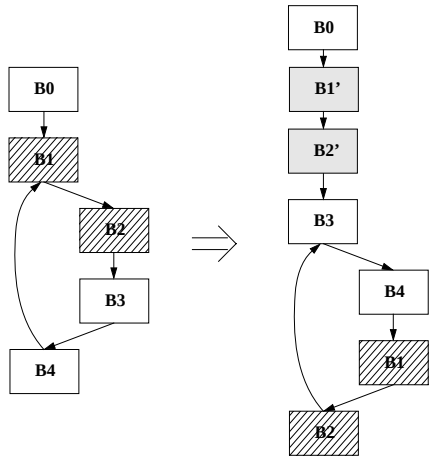

Figure 13: Loop Peeling Transformation

Our loop peeling pass begins with a search phase that looks for loops which should be peeled. The search includes all loops, starting with the most highly nested loops first. A loop is a candidate if it contains a loop-invariant faulting operation. Also, the faulting operation must dominate the source blocks of all of the loop backedges. Otherwise, it may not be executed on every iteration or it may actually be in code that is exiting the loop. The set of blocks that are dominated by the loop header and dominate the block containing the faulting operation will be the "peel", the part of the first iteration that will be peeled off before the main loop. Because we do not want to create too much duplicated code, another criteria is that the peel is not too large. We therefore choose the largest useful peel that does not exceed a maximum number of blocks. To simplify the peeling process, there are also a few other requirements. For example, a peel is not allowed to contain a nested loop.

The loop peeling process is complex, but straightforward. The main idea is to duplicate the blocks of the peel and change the control flow to go first through the duplicated blocks and then enter the loop below the peel. The transformation is illustrated in Figure 13. Call the originals blocks in the loop that are being peeled the "peel-originals" (striped in the figure) and the duplicated blocks the "peel-copies" (gray in the figure). First, the peel-original blocks are copied to form the peel-copy. The values in the peel-original are also duplicated to the peel-copy. During the copying, uses of the phi nodes at the top of the loop peel are converted to uses of the associated phi input entering from outside the loop. Then the edges entering the loop header from outside the

loop are changed to point to the peel-copy. Removing the edges entering the loop header may cause some phi nodes in the original loop header to disappear. The successor of the last block of peel-copy is set to be the block in the loop just below the peel-original, which will now become the new loop header. Finally, for all values which were copied, phi nodes are created in the new loop header and all other blocks (such as loop exits) that join edges from the peel-copies and peel-originals. There are very rare cases when the addition of these phi nodes may require the creation of further phi nodes. In these cases, we abort the peeling process. Alternatively, we have also done an implementation in which we rerun the phi-node placement algorithm [30] on just these new nodes to determine all additional phi nodes required.

3.3.6 Pattern-based Peephole Optimizer

Swift contains several passes that do pattern-based transformations of the SSA graph. For example, one pass does machine-independent peephole optimizations, and another pass does machine-dependent conversion, as well as some machine-dependent peephole optimizations. These passes are all automatically generated by a pattern-matcher generator. This generator takes a set of graph rewriting rules as input and produces a compiler pass which transforms the SSA graph according to those rules.

Each rule consists of a pattern that can match a piece of the SSA graph, and some Java code which is executed when the pattern matches. The pattern language allows matching of node operations, auxiliary operations, and types, and allows nested matching on the inputs to a node. For example, here is a simple rule that matches an integer addition of an arbitrary value and zero:

```
add@int(_, constant[0]) { ... }
```

The pattern language allows matching of any operation, a particular operation, a list of operations, or any operation not in a list. Similar matching capabilities exist for types. The language also allows binding of a variable to a particular value in a pattern, for use in the action part of the rule. The top-level value that is matched by the pattern is available in the action via the keyword `self`. The action can mutate the `self` value or any of its inputs (recursively).

The graph rewriting order is as follows. An SSA value is processed before all of its inputs (except when there are cycles in the value graph due to phi-nodes.) For each value, rules are applied in the order specified by the rules file until one matches and mutates a value. The rewriting then proceeds to the next value. Repeated passes are made through the entire SSA graph until no changes occur on a pass. The ordering of rules is therefore somewhat important in ensuring that optimizations are applied in the most effective order, but we have not found it difficult to choose such an ordering.

Many of our machine-independent peephole optimizations apply algebraic or boolean identifies. We have previously mentioned the optimization in which the length of a value which is a new array is replaced by the allocated size of the array. Another interesting optimization is replacing a **get_field** operation by the value stored by a previous **put_field** operation, given that the object and global store inputs of the two values are identical.

3.3.7 Other Optimizations

Swift does a number of other standard machine-independent optimizations. First, it does repeated passes of dead code elimination, typically after a pass that has greatly changed the structure of the SSA graph. As mentioned in Section 2.2, the live values in a Swift SSA graph are the control values and (recursively) all inputs of live values. Swift does dead code elimination by simply marking all the live values, and then removing all the unmarked values.

Another standard pass that Swift does is conditional constant propagation [32]. This pass does constant propagation and elimination of impossible control flow in the same pass. In this way, more optimization opportunities are found than would be found by each pass individually. Finally, Swift also includes a general strength reduction pass. We use the method described in [13], which is an elegant version of strength reduction for SSA graphs.

3.4 Machine-dependent Processing

In this section, we describe some of the machine-dependent phases in the Swift compiler. The pass that converts the IR to machine-dependent operations is generated by the same pattern-matcher generator that is described in Section 3.3.6, and we will not describe it further. The other machine-dependent passes described below include sign-extension elimination, instruction scheduling, register allocation, and code generation.

3.4.1 Sign-extension Elimination

The semantics of Java require that sign extension operations be performed quite often. However, a little bit of analysis can often prove that these sign-extension operations are not needed. For example, in the `setX` method in Figure 14, an `int` is cast to a `short` and then stored into a `short` field. Because the store only uses the low-order 16 bits of the casted value, we don't actually need to perform the cast. The sign-extension cleanup pass searches for cases like this one and eliminates any redundant operations it finds.

Two different techniques are used to perform sign-extension elimination. The first technique computes how many low-order bits each input of a value needs. This information is computed by a simple backwards walk through the SSA graph. Then, any operation whose output is equal to one of its inputs on those low-order bits can be replaced with that input. For instance, the `sextw` in Figure 14 only needs to compute the low-order 16 bits of its output, and a

```
class Example {
  short x;
  void setX(int v) {
    x = (short) v;
  }
  int highHalf(long v) {
    long t = v >> 32;
    return (int) t;
  }
}

** code for Example.setX
0x00000 sextw   a1, a1      ; useless!
0x00004 stw     a1, 8(a0)
0x00008 ret     (ra), 1

** code for Example.highHalf
0x00000 sra     a1, 32, v0
0x00004 sextl   v0, v0      ; useless!
0x00008 ret     (ra), 1
```

Figure 14: Examples of useless sign extensions.

`sextw` is the identity function on its low-order 16 bits, so the `sextw` can be safely replaced with its input.

The second technique computes the state of the high-order bits of each value. The state of a value includes the number of known bits and whether those bits are zero or sign-extension bits. Any operation that we can show to be unnecessary because its input is in a certain state can then be eliminated. For example, in Figure 14, the `t` variable in `highHalf` is known to have its high 32 bits in a sign-extended state. Because any `sextl` is a NOP when its input is already 32-bit sign extended, the subsequent `sextl` can be eliminated.

3.4.2 Trace Scheduling and Block Layout

Swift has a full instruction scheduler that can operate on traces of one or more basic blocks. The scheduling process includes three main steps: (1) decomposing the CFG into traces; (2) scheduling the instructions within each trace; and (3) determining a good sequence for the layout of the traces.

The choice of traces is driven by profile information or static estimates of block and edge execution frequencies. A simple greedy algorithm proceeds by choosing the most frequently executed block B that is not yet placed in a trace as the first block in a new trace. The algorithm then tries to extend the trace upwards and downwards in the CFG. If B has only a single predecessor P in the CFG and P is not yet in a trace, then P is added to the trace and so on, recursively, with P's predecessor. The trace is then extended downward by following the most frequently executed successor edge of B. If that edge goes to block S, and S is not in a trace yet, and S has only a single predecessor, then S is added to the trace, as so on, recursively, with S's successors. The algorithm proceeds until all blocks have been placed in a trace. The result is a set of traces that are extended basic blocks and that cover the CFG.

The instruction scheduler operates on a trace at a time. It first builds up a set of all the dependences that determine where a value (instruction) can be placed. Most of these dependences already exist in the Swift IR, but the instruction scheduler explicitly adds control and anti-dependences. Because all the control dependences are explicit, memory store operations are not pinned in any particular block during scheduling. However, control-flow and exception-causing operations are still required to be at the end of the basic block which they control. In addition, special “out-of-trace” dependences are added to make sure that a value is scheduled early enough in the trace to dominate any of its users that are not in the trace.

The scheduler also includes a model of the Alpha 21164 and 21264 pipelines, embodied in a finite automata. This automata allows the scheduler to determine when the result of an operation is likely to be ready, based on the expected latency of the operation and the state of the pipeline [4]. Given the dependence and latency information, the overall scheduling algorithm is fairly simple. At each point, the scheduler chooses to schedule a value whose dependences have all been satisfied, and whose inputs are all ready or will be ready at the earliest time. Whenever a control-flow or exception-causing value is chosen, then the current basic block is ended. When a value is scheduled, the state of the finite automata is updated to reflect its execution.

Swift again uses profile information for determining a good layout for the traces. The main goal is to ensure that traces joined by frequently executed edges are placed next to each other, so that an extra branch instruction is avoided and instruction cache locality is improved. Swift uses a simple version of Pettis and Hansen's code layout algorithm [29], which is a greedy algorithm that gradually merges blocks/traces into sequences, and always merges the two sequences that have the heaviest-weight edge between an element of one and an element of the other. Eventually, the algorithm produces a single sequence, which is the layout of the blocks/traces. Swift modifies the dynamic or statically estimated profile information slightly, by reducing the weight of other outgoing edges of a block which has an edge that exits a loop. This change makes it likely that a loop exit block will be placed at the end of a loop, thereby ensuring that there is only one branch per loop iteration in the case of a simple loop. Swift also gives lower weight to edges that leave a trace in the middle, since these edges have already been determined to be less important than the remaining edges in the trace.

3.4.3 Register Allocation

The register allocator in Swift is a modified Briggs-style coloring allocator [7]. Our allocator is novel in that it does not use coalescing, but instead uses a special data structure, the

bias graph, to direct coloring and limit the number of copies introduced.

Register allocation proceeds by assigning each value a *color* which represents a particular register assignment. Restrictions on which values can be in which registers at particular times are encoded in a graph coloring problem which is then solved by coloring heuristics. Allocation proceeds according to the following algorithm:

1. Insert copies
2. Precolor
3. Construct bias graph
4. Construct interference graph
5. Compute coloring order
6. Color values
7. If some values failed to be colored
   (a) Spill uncolored values to the stack
   (b) Go to step 4
8. Clean up

We now briefly describe each of the above phases. The first phase adds copies in the SSA graph to allow coloring to occur. Copies are required because the fundamental assumption of a coloring allocator is that each value is allocated to exactly one register for its entire lifetime. If a value may need to move from one register to another, a copy must be inserted so that the value can be in more than one register. Copies are inserted wherever a copy might be required, as, for instance, when moving a method argument or return value to or from a fixed register. Copies are also required for all the inputs of a phi node, since the input values of the phi node may not be assigned to the same register as the phi node itself. In addition, we use the LIMIT algorithm [27] to split the live ranges of values around loops in which they are not referenced. The live range of a value is split by inserting copies of the value before and after the loop, which facilitate their spill of the value in deference to values which are used in those loops.

The next phase is value precoloring. The compiler determines which values must be assigned to certain registers and fixes their color assignment. Values which have fixed register assignments include method arguments and return values.

Next, we construct the *bias graph*, which is an undirected graph that has values as nodes and edges between nodes which we want to color the same color. The idea behind the bias graph is to undo as many of the copy insertions from step 1 as possible by trying to color the input and output of a copy the same color. The bias graph will be used in step 6 to select among the possible colors for a value.

Then, we construct the *interference graph*. The interference graph has values as nodes and edges between nodes that cannot be assigned the same color because their live ranges overlap. The interference graph completely encodes the possible legal assignments of colors to values. The register allocation problem thus reduces to the graph coloring problem: finding an assignment of colors to nodes such that no two nodes which are adjacent in the interference graph have the same color. The graph coloring problem is NP-hard in general, so we use heuristics to solve it.

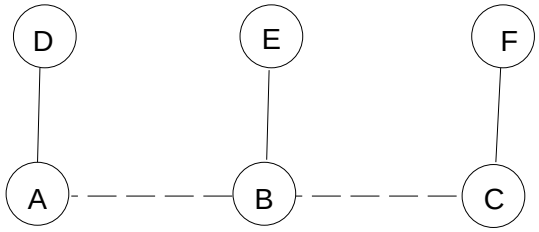

Figure 15: Bias graph coloring

The next step is to compute a coloring order for all of the nodes. The idea is to select a coloring order that colors "hard" nodes first and "easy" nodes last, where the difficulty of coloring a node is approximately proportional to its degree in the interference graph. The algorithm proceeds by repeatedly removing a node with minimum degree from the interference graph and deleting any edges which involved that node. This step is repeated until all nodes are removed from the graph. Note that the node with the smallest *remaining* degree among all nodes remaining in the interference graph is removed. The order of coloring is then the reverse order in which the nodes were removed. This order ensures that nodes with low degree (and thus easy to color) are colored after nodes with higher degree.

The last major step is to color each of the nodes in the order computed. To color an individual node, we first compute the set of possible legal colorings of that node. The legal colorings of a node include all registers that could hold the associated value (i.e., all floating-point registers for floating-point values), minus the colors of any colored neighbors of the node in the original interference graph. Any color in this set would be a legal color to assign to that node. If the set is empty, then the node cannot be colored and step 7 will trigger when coloring is completed.

The bias graph is used to make an intelligent choice of a color from the set of legal colorings allowed by the interference graph. Figure 15 illustrates how biased coloring works. Dotted lines are edges in the bias graph, and solid lines are edges in the interference graph. From the node we wish to color, we do a breadth-first search in the bias graph. If we find a node that is already colored, we color the original node the same color if no node along the path from the start to the colored node cannot be colored that color. For instance, if we are trying to color node A, then we first try to color it the same color as node B, if B is colored. If B is not colored, we try to color it the same color as node C, but only if that color is also an allowed color for node B, i.e. is not the color of node E. In other words, there is no point in coloring A and C the same color if that color can't also be used for B.

If none of the nodes found have a color that can be used for A, then we do another BFS on uncolored nodes in the bias graph, intersecting the set of colors allowed for A with the set of colors allowed for each uncolored node. The color we choose for A is any color from the last nonempty set of colors computed this way. This rule allows for the maximum number of nodes connected to A in the bias graph to later pick A's color to match with.

If coloring does not succeed, then we must spill values to the stack. The value corresponding to each node that was not colored is spilled by inserting a **spill** value just after its definition and a **restore** value before each use. That way, the original value and the newly added **restore** values need to be in a register over a shorter range and thus will hopefully be easier to color on the next pass.

When coloring succeeds, a final cleanup pass is done to remove copies that have the same source and destination, as well as remove unnecessary **restore** operations. The cleanup pass does a dataflow computation to determine what value each register holds after each instruction and uses this information to replace each input value of each instruction with the oldest copy of that input value which is still in a register.

3.4.4 Code Generation

Swift's code generation pass is a translation of the SSA operations into machine code. Most operations remaining in the SSA graph after machine-dependent translation (Section 3.3.6) correspond to zero or one Alpha instructions, so emitting code for those operations is straightforward. The code generator is also responsible for the following tasks:

- Computing the stack frame size.
- Emitting prolog code.
- Emitting code for each block, in the order determined by the scheduling pass (Section 3.4.2).
- Emitting a branch when one of the successors (possible the only successor) of a block is not the immediately following block in the block layout.
- Emitting epilog code.
- Emitting auxiliary information, including a list of relocation entries, a list of associated constants, an exception table, and a bytecode map.

Swift does some extra processing to determine which blocks are empty (i.e. contain only values that generate no code), so that it can easily determine when branches are necessary and which non-empty block is the final destination of a branch.

4 Performance Results

In this section, we give some performance results for the Swift Java compiler. We first describe the experimental platform, the applications used in our study, and some overall performance results. We then analyze in detail the usefulness of various optimizations.

4.1 Experimental Platform

Our performance results are for the Swift compiler running under Tru64 Unix (formerly known as Digital Unix) on an Alpha workstation. The workstation has one 667 MHz Alpha 21264 processor, which has 64 Kbyte on-chip instruction and data caches and a 4 Mbyte board-level cache. The generated code is installed into a high-performance JVM for Java 1.2 that has a mostly-copying garbage collector, extremely fast synchronization, and quite good JIT code generation [12]. The fast JVM also does a limited form of CHA, since it will resolve a virtual call to a direct call based on the set of classes already loaded, but go back to using a virtual call if a new class is loaded which causes a virtual call to be necessary. The heap size used in all the runs is 100 Mbytes.

4.2 General Results

We measure our results for a number of applications, including those in the SpecJVM98 suite. Table 1 lists the applications and problem domains, as well as the number of lines of code. Column 4 contains the base running times of each application when running on the high-performance JVM (without using Swift). Columns 5, 6, and 7 contain the times when running using Swift-generated code. The results in Column 7 are when all optimizations are used. The result in Column 5 are for the same optimizations, except class hierarchy analysis is not used. Clearly, the use of CHA greatly increases the overall performance. The overall speedup of the Swift-generated code without CHA over the fast JVM is somewhat low, because the fast JVM is already using CHA to resolve method calls, as indicated above. The results in Column 6 include the use of simple-CHA (s-CHA), as described in Section 3.2.6. Interestingly, the performance is only somewhat less than the performance with full CHA. Hence, simple-CHA appears to be a useful alternative when Swift-generated code is to be used in the presence of dynamic loading.

For the applications in Table 1, Swift compiles at the rate of about 1800-2200 lines per second on the local machine, when all optimizations are on except those enabled by escape analysis (synchronization removal and stack allocation). Escape analysis can take significant CPU time, since methods are recursively analyzed for their effects on their arguments. In particular, escape analysis may require examining numerous methods in the standard Java library. In general, the compilation slows down by about 20-40% when escape analysis is used, and it may sometimes be even slower.

problem domain		lines of code	JVM time	Swift run-time		
				w/o CHA	with s-CHA	with CHA
compress	text compression	910	12.68s	9.61s	9.72s	9.66s
jess	expert system	9734	4.97s	4.35s	4.17s	4.12s
cst	data structures	1800	8.02s	5.97s	5.65s	5.38s
db	database retrieval	1026	17.73s	15.62s	12.73s	12.44s
si	interpreter	1707	8.09s	6.48s	5.93s	6.33s
javac	Java compiler	~18000	8.50s	7.57s	7.14s	7.00s
mpeg	audio decompr.	~3600	10.63s	5.74s	5.60s	5.68s
richards	task queues	3637	8.09s	8.52s	5.30s	4.69s
mtrt	ray tracing	3952	4.69s	5.11s	2.09s	1.59s
jack	parser generator	~7500	5.92s	5.27s	4.90s	4.96s
tsgp	genetic program.	894	35.89s	25.70s	24.10s	24.05s
jlex	scanner generator	7569	4.96s	4.10s	3.84s	2.95s
speedup over JVM				**1.21x**	**1.43x**	**1.52x**

Table 1: Java Applications and Overall Results

	inl	cha	fld	objinl	split	stk	sync	sr	cse	gcm	peel	ckelim	selim	br
compress	1.16	1.20	1.16					1.09	1.06				1.04	
jess	1.07	1.09							1.04		1.03	1.04		
cst	1.08	1.04					1.05		1.07					
db	1.05	1.26	1.04	1.03	1.03	1.04					1.03			
si	1.27	1.14	1.05	1.04	1.06	1.16			1.12				1.04	1.09
javac	1.09	1.09												
mpeg	1.07		1.13					1.05	1.35					
richards	1.40	1.76												1.11
mtrt	1.57	2.68	1.27	1.16		1.13		1.09	1.06					
jack		1.05												
tsgp	1.03	1.05						1.12	1.05		1.05			
jlex	1.22	1.19		1.15	1.18		1.15							

Table 2: Effects of Various Optimizations

4.3 Detailed Results

Table 2 specifies the effects of performance improvements of many of the Swift optimizations on our applications. The table includes rows for each application and columns for analyses/optimizations. The labels on the columns are: **inl** (method inlining), **cha** (using CHA), **fld** (field analysis), **objinl** (object inlining), **split** (method splitting), **stk** (stack allocation), **sync** (synchronization removal), **sr** (store resolution), **cse** (global CSE), **gcm** (global code motion), **peel** (loop peeling), **ckelim** (run-time check elimination via program properties), **selim** (sign-extension elimination), and **br** (branch removal). The number in each box specifies the increase in execution time for the specified application when the associated optimization or analysis is *not* used.[8] (A box is left blank if there is no significant change in performance.) The baseline performance is the run-time of the application with all optimizations enabled. Note that the numbers going across a row are not additive, since disabling one optimization or analysis often reduces the effectiveness of another optimization. In particular, the field analysis results include the effect of object inlining, since object inlining is completely disabled if field analysis is disabled. Note also that performance can change when disabling an optimization simply because the layout of the application in the instruction cache changes. The usefulness of various optimizations would also change if we measured results on a processor with different properties (e.g. a statically-scheduled processor). So, the numbers in Table 2 should be taken simply as rough indicators of the usefulness of various optimizations.

We can make a few observations from the table. First, method inlining, CHA, and global CSE are effective at improving the performance of almost all applications. Method inlining is especially important for mtrt and richards, because they have many calls to small methods. CHA is highly important for richards and mtrt as well, since most of the calls to small methods are virtual calls, and no classes are declared `final`. The use of CHA has a large effect on sev-

[8] When global code motion is disabled, the CSE pass is modified to only combine values when one value dominates the other.

	total allocated		% stack allocated		inlined
	objects	bytes	objects	bytes	objects
compress	529	109M	5.1	.0005	75
jess	7907874	246M	.0004	.0002	407
cst	3699556	91.6M	3.9	2.5	31
db	3058608	71.7M	95.0	64.8	15636
si	9122929	139M	12.3	21.9	3
javac	4557786	101M	12.6	8.7	29344
mpeg	1136	92.1K	0.2	.0006	250
richards	20815	458K	10.0	5.4	5600
mtrt	6560044	116M	83.4	79.7	381671
jack	5905911	107M	38.0	56.1	48734
tsgp	5256	3.65M	0.9	.0003	1
jlex	1359094	43.7M	14.2	10.6	33

Table 3: Dynamic Allocation Counts

eral applications, not only because it helps method resolution and inlining, but also because it allows other analyses (such as escape analysis) to derive more useful information in the presence of virtual calls.

Field analysis has a large effect in compress and mpeg, because it helps eliminate many null checks and bounds checks on some constant-sized buffers and computation arrays. Similarly, in mtrt, field analysis helps eliminate checks on small arrays used in data structures. For example, in mtrt, a data element known as an oct-node always contains arrays to hold six references to adjacent oct-nodes, eight references to child oct-nodes, and six references to face objects. Object inlining also improves the performance of mtrt by inlining some of these arrays. In db, Swift successfully inlines a vector object contained within a database entry (including its header), thereby eliminating an extra pointer dereference on each access.

Method splitting is useful in db because db uses `java.util.Vector.elementAt`. Splitting is also highly effective in si because the `ensureOpen` method in `java.io.PushbackInputStream` does only a simple check on a file handle, and it is called by the frequently-used `read` method.

Stack allocation is often successful in finding many objects that can be stack allocated, but performance does not always improve correspondingly, because of the large heap and the effectiveness of the JVM's garbage collector. Table 3 shows dynamic counts of the objects and bytes allocated in each application, and the percentage of the objects and bytes that were allocated on the stack. The performance of db improves because an important comparison routine repeatedly generates and uses an enumeration object, which can be stack allocated. Similarly, an important routine in mtrt uses a temporary object in its computation which can be allocated on the stack. Gains from stack allocation would be greater for runs using a smaller maximum heap.

As with stack allocation, synchronization removal typically finds numerous objects whose operations can be unsynchronized, especially manipulations of `StringBuffer` objects (which are used in building new strings). However, many synchronization sites do not contribute significantly to an application's run time. The improvements in cst and jlex result from removing synchronization for frequent operations on vector and hash table data structures that are implemented in the Java standard library.

Storage resolution is effective in compress, mtrt, and tsgp, because each has important loops whose memory operations and run-time checks are effectively optimized only after memory dependences are relaxed. Global code motion doesn't seem to greatly affect performance, perhaps because most important movement of values out of loops is already accomplished by CSE. Sign-extension elimination is effective in compress and si, because these applications do many memory accesses to byte arrays. Branch removal is especially effective in richards, because it eliminates the branches in the computation of a boolean by a frequently called method.

In Figure 16 and 17, we show the effects of various optimizations on the numbers of null checks and bounds checks executed in each application. For each application, the first bar, which is labeled 'N' and scaled to 100%, represents the number of checks executed when all optimizations except CSE, check elimination, field analysis, and loop peeling are used. Successive bars represent the number of checks executed as the following optimizations are successively added: CSE ('C'), check elimination ('E'), field analysis ('F'), and loop peeling ('P'). Clearly CSE is highly effective at eliminating null checks, but does not eliminate many bounds checks (except for mpeg and tsgp). Similarly, loop peeling eliminates a large number of null checks in several applications, but does not typically remove bounds checks. These results make sense, since bounds checks are not typically loop-invariant. Field analysis is useful for eliminating many null checks and bounds checks on the contents of fields. Check elimination removes significant null checks in a few applications and a noticeable number of bounds checks in a few applications.

In Figure 18, we show the effect on execution time when all run-time checks are removed. Clearly, removing run-time checks without proving that they cannot cause an exception is in violation of Java semantics, but we provide these results to give an estimate of the cost of the remaining run-time checks that Swift cannot eliminate. The left bar represents the time for each application when all optimized applications are turned on, and is scaled to 100%. The right bar represents the time when the remaining run-time checks are replaced by **pin** operations. We cannot completely remove the run-time checks, since then operations depending on the checks might move up above a conditional or other operation that guards the operation and ensures that it is legal. We do move the **pin** operations upward in the CFG as far as possible without going past a branch or memory store operation. Figure 18 shows that the overhead of the remaining run-time checks in Swift code is about 10-15%.

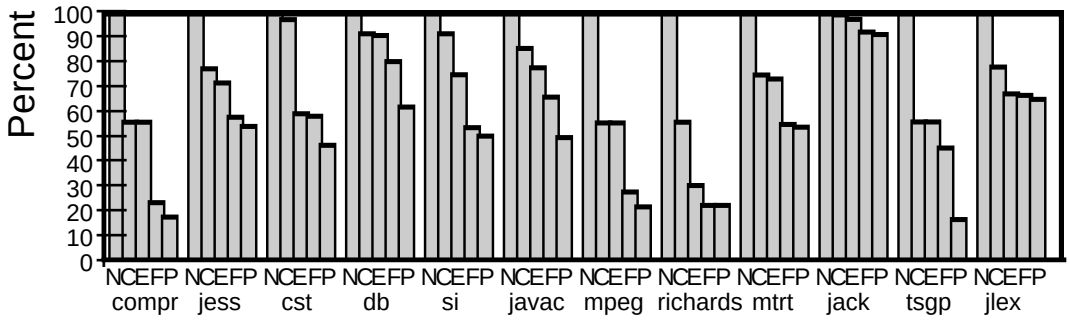

Figure 16: Dynamic Counts of Null Checks

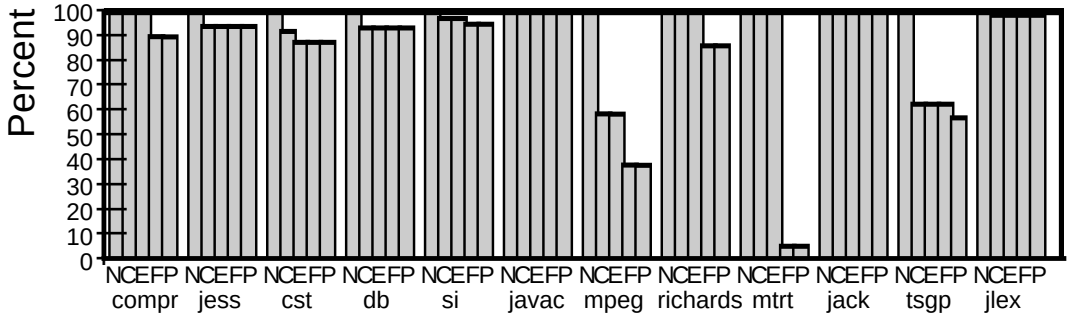

Figure 17: Dynamic Counts of Bounds Checks

In Figure 19, we show the effects of several optimizations on the number of virtual and interface method calls. The first bar, again labeled 'N' and scaled to 100%, for each application represents the total number of unresolved virtual and interface method calls when only virtual calls to private or final methods are resolved. The following bars represent the unresolved calls as the use of the following information is successively added: type propagation ('T'), class hierarchy analysis ('C'), field analysis ('F'), and information about the types of the return values of methods ('M'). CHA clearly helps resolve a large number of calls in most of the applications. Type propagation, field analysis, and method return information each have mostly small effects on a number of applications. The method return information is useful in db for resolving a number of interface calls to enumeration objects.

5 Related Work

There is clearly much work related to the design of the Swift IR and the optimizations done by Swift. In this section, we first give details on some other optimizing Java compilers and then describe some of the work related to Swift's most interesting techniques.

Many just-in-time (JIT) Java compilers have existed for several years, but we will not discuss them here. However, there are also several other complete optimizing compilers for Java. Marmot [22] is a research compiler from Microsoft, and uses SSA for its intermediate form. It does a form of CHA, but doesn't have much interprocedural analysis, does limited code motion, and does not do any instruction scheduling. Unlike Swift, it has a separate low-level IR that is not SSA-based. The Jalapeno [8] system includes a JVM written almost exclusively in Java and an optimizing compiler. The compiler has three different levels in its IR and is not SSA-based. Jalapeno currently does very limited optimizations across basic blocks. BulletTrain [28] is a commercial compiler which uses SSA for its IR. It apparently does some check elimination, loop unrolling, type propagation, and method inlining. HotSpot [31] is a commercial Java compiler from Sun that dynamically compiles code that is frequently executed and can use run-time profiling information. It does method inlining based on class hierarchy analysis. TurboJ [25] is a commercial Java compiler that translate bytecodes to C, for compilation by the native C compiler. It does some method resolution, inlining, CSE, and code motion during the translation.

With respect to the ordering of memory operations, Marmot [22] appears to keep memory operations in order, except for doing a special pass to promote loads out of loops. Jalapeno [8] builds an instruction-level dependence graph that includes all data and control dependences after lowering the IR. This information is not available to the earlier high-level passes. Diwan [19] used type-based alias analysis to disambiguate locations, as we do. However, he does not incorporate the results of the analysis into an SSA representation. Cytron [16] represents alias information in an SSA graph by explicitly inserting calls that may modify values if an associated pointer operation may modify the value. Such an approach can greatly increase the size of the SSA graph. Instead, we enforce strict memory ordering via the global

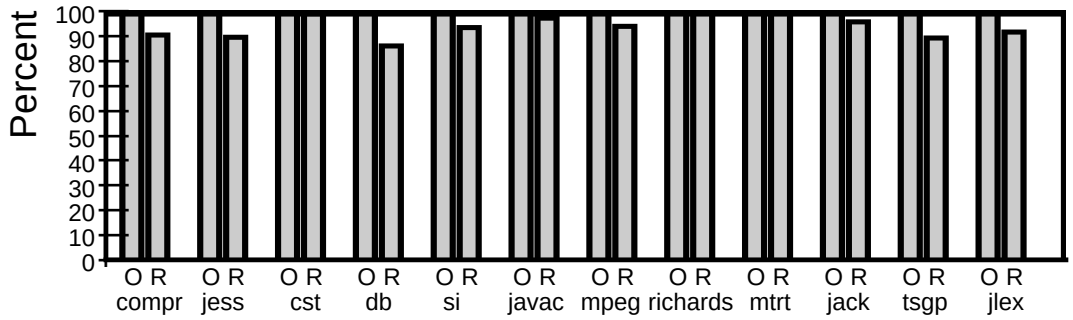

Figure 18: Change in Execution Time When Removing Remaining Bounds Checks

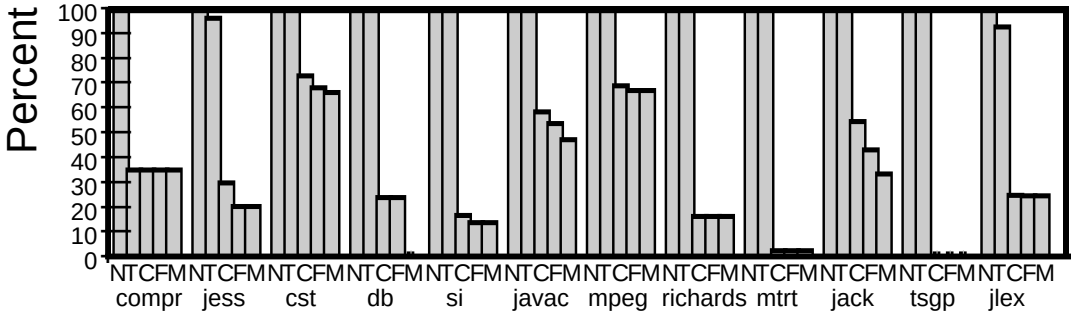

Figure 19: Dynamic Counts of Virtual and Interface Calls

store inputs and relax dependences when we can prove there are no aliases.

Diwan *et al.* [20] proposes the use of *aggregate analysis* to detect when a polymorphic data structure is used in a monomorphic way by looking at the types of all assignments to a particular field. For example, the analysis is used to show that a linked list of general objects actually contains only objects of a certain class or its subclasses. This analysis is related to our field analysis that determines exact types, but aggregate analysis does not make use of any modularity properties in the language, or investigate any other properties of fields.

Dolby and Chien [21] describe an object inlining optimization for C++ programs. Their analysis uses a fully context-sensitive interprocedural framework and thus allows object inlining in specific cases on a field that cannot be inlined in general. In particular, they do not require that the field be initialized with a new object in the constructor. However, their analysis times are measured in minutes, whereas our analysis is always only a small number of seconds. Also, we allow objects to be inlined (with a header), even if a reference to the objects escape the local context. A large amount of related work with respect to object inlining (or *unboxing*) also exists for functional languages, as described in [21].

There are have been a variety of recent approaches to doing useful escape analysis. [2, 5, 6, 9, 33]. We have done the simplest form of escape analysis [23], augmented with information from field analysis. As far as we know, none of the preceding systems use field access properties to aid in their escape analysis. Some Java compilers no doubt perform extra optimizations for `final` fields, but none seem to have exploited field properties to the same extent as Swift. Most of the other recent approaches have used methods that are more complicated and potentially quite expensive computationally.

6 Conclusion

We have implemented a complete optimizing compiler for Java with an intermediate form based on static single assignment (SSA). We have found that the design of the Swift IR has simplified many aspects of the compiler, and almost all optimizations have been easy to express in SSA form. Because the Swift IR includes machine-dependent operations, all passes can operate directly on the same SSA-based representation, thereby allowing them to take advantage of common functionality for manipulating the CFG and SSA graph. Our compiler has a speed comparable to normal C compilers, even though it does extensive interprocedural analysis.

Overall, we have found that numerous optimizations are necessary to remove the many sources of overhead in Java. As might be expected, the most effective optimizations in Swift are method inlining, class hierarchy analysis (which helps resolve virtual methods), and global CSE. However, both field analysis and store resolution, which are techniques unique to Swift, have significant effects on a number of applications. Many other optimizations have a big effect on one application and/or limited effects on several applications.

There is still much room for improving performance of Java programs. More extensive interprocedural analysis and more powerful provers of program properties could help

eliminate more of the run-time checks. Much overhead still remains because of the use of virtual method calls and object inheritance. Swift could eliminate more of this overhead by techniques such as context-sensitive interprocedural analysis or eliminating inheritance for an important class by copying the inherited methods into the class. Finally, much of the overhead in Java appears to result from the object-oriented style that results in many smaller objects (or structures) than in a corresponding C program. This style results in greater memory latencies during execution as pointers between objects are dereferenced. Hence, there are many opportunities to increase performance by optimizations such as prefetching, co-locating objects, or more aggressive object inlining.

Acknowledgments

We would like to thank Raymie Stata and Mark Vandevoorde, who participated in the early design and implementation of Swift, and Girish Kumar, who did the initial implementation of stack allocation. We would also like to thank Caroline Tice for comments on an earlier draft of this paper.

References

[1] Java Memory Model Mailing List. At URL `http://www.cs.umd.edu/~pugh/java/memoryModel/`.

[2] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Sixth International Static Analysis Symposium*, Sept. 1999.

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *15th ACM Symposium on Principles of Programming Languages*, pages 1–11, Jan. 1988.

[4] V. Bala and N. Rubin. Efficient Instruction Scheduling Using Finite Automata. In *28th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 1995.

[5] B. Blanchet. Escape Analysis for Object Oriented Languages. Application to Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.

[6] J. Bogda and U. Holzle. Removing Unnecessary Synchronization in Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.

[7] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Apr. 1992.

[8] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Dynamic Optimizing Compiler for Java. In *1999 ACM Java Grande Conference*, June 1999.

[9] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.

[10] C. Click. From Quads to Graphs: An Intermediate Representation's Journey. Technical Report CRPC-TR93366-S, Center for Resesearch on Parallel Computation, Rice University, Oct. 1993.

[11] C. Click. Global Code Motion; Global Value Numbering. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[12] Compaq Computer Corporation. Compaq Fast Virtual Machine V1.2.2-1 for Alpha. At URL `http://www.compaq.com/java`.

[13] K. Cooper, T. Simpson, and C. Vick. Operator Strength Reduction. Technical Report CRPC-TR95-635-S, Rice University, Oct. 1995.

[14] K. D. Cooper and J. Lu. Register Promotion in C Programs. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 308–319, June 1997.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Jan. 1989.

[16] R. Cytron and R. Gershbein. Efficient Accommodation of May-Alias Information in SSA Form. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 36–45, June 1993.

[17] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 77–101, Aarhus, Dänemark, 1995.

[18] D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *ECOOP '99*, pages 258–278, June 1999.

[19] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-Based Alias Analysis. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[20] A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In *1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1996.

[21] J. Dolby and A. Chien. An Evaluation of Automatic Object Inline Allocation Techniques. In *1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1998.

[22] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An Optimizing Compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, 1999.

[23] D. Gay and B. Steensgaard. Stack Allocating Objects in Java. At URL `http://www.research.microsoft.com/apl/stackalloc-abstract.ps`.

[24] S. Ghemawat, K. H. Randall, and D. J. Scales. Field Analysis: Getting Useful and Low-cost Interprocedural Information. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, June 2000. To appear.

[25] Groupe Silicompe. TurboJ Java compiler. At URL `http://www.ri.silicomp.fr/adv-dvt/java/turbo`, 1999.

[26] T. B. Knoblock and J. Rehof. Type Elaboration and Subtype Completion for Java Bytecode. In *27th ACM Symposium on Principles of Programming Languages*, Jan. 2000.

[27] R. Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, Woburn, Massachusetts, 1998.

[28] NaturalBridge, LLC. BulletTrain Java compiler. At URL `http://www.naturalbridge.com`, 1999.

[29] K. Pettis and R. Hansen. Profile Guided Code Positioning. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[30] V. C. Sreedhar and G. R. Gao. A Linear Time Algorithm for Placing Phi-nodes. In *22nd Annual ACM Symposium on Principles of Programming Languages*, Jan. 1995.

[31] Sun Microsystems. Java HotSpot VM. At URL `http://www.javasoft.com/products/hotspot`, 1999.

[32] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Trans. Prog. Lang. Syst.*, 13(2):181–210, Apr. 1991.

[33] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.