View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace@MIT

# JET:
## An Application of Partial Evaluation in Dynamic Code Generation for Java

by

**Tony Chao**

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology.
May 26, 2000

Copyright 2000 Tony Chao. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author_____________________________________________
Department of Electrical Engineering and Computer Science
May 17, 2000

Certified by________________________________________
M. Frans Kaashoek
Thesis Advisor

Accepted by_________________________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

# JET:
# An Application of Partial Evaluation in Dynamic Code Generation for Java

by

Tony Chao

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology.
May 26, 2000

# Abstract

Java is a popular new language with enormous potential; however, its lack of speed is a major drawback. Staged compilation and runtime specialization through procedure cloning are techniques used to improve code generation and execution performance. The research described in this paper applies these techniques in the design and implementation of a runtime system to improve Java performance. Analyses indicate that staged compilation results in a major improvement in performance. In this current implementation, runtime specialization and constant propagation provides a smaller incremental benefit, but with more aggressive and new forms of specialization, the benefits of dynamic specialization will likely increase.

Thesis Supervisor: Robert Morgan
Title: Principal Member of Technical Staff, Compaq Computer Corporation

Thesis Advisor: M. Frans Kaashoek
Title: Associate Professor, MIT Laboratory of Computer Science

# Table of Contents


**1 INTRODUCTION** .......... **4**

**2 BACKGROUND** .......... **6**

**3 MOTIVATION** .......... **9**

**4 DESIGN** .......... **10**

4.1 TURBO .......... 12
4.2 AFTERBURNER .......... 12
*4.2.1 Parser* .......... *14*
*4.2.2 Generator* .......... *15*
4.2.2.1 Scheduler .......... 16
4.2.2.2 Register Allocation .......... 16
4.2.2.3 Emitting Code .......... 19
*4.2.3 Specializer* .......... *21*

**5 ANALYSIS** .......... **24**

**6 CONCLUSION AND FUTURE WORK** .......... **26**

**APPENDIX A: JET IR SPECIFICATION** .......... **29**

*JET* .......... *29*
*Method Information* .......... *30*
*Block Information* .......... *31*
*Value Information* .......... *33*
*Auxiliary Information* .......... *36*
*Bit Vector* .......... *44*
*Exception Information* .......... *45*
*UTF8-String constants* .......... *45*

**APPENDIX B: JET OPCODES** .......... **46**

**APPENDIX C: JET CALLING STANDARD** .......... **50**

**APPENDIX D: REGISTER ALLOCATION** .......... **52**

**APPENDIX E: CONSTANT PROPAGATION** .......... **56**

**APPENDIX F: SAMPLE GENERATION AND SPECIALIZATION** .......... **60**

**APPENDIX G: PERFORMANCE DATA** .......... **62**

**REFERENCES** .......... **64**

# Table of Figures

Figure 1: JET Stages and Integration ........ 11

Figure 2: JET Code Generation Process ........ 13

Figure 2: JET Specialized Generation Process ........ 15

# 1 Introduction

Staged compilation and runtime specializations are techniques used to improve code generation and execution performance. This research applies such techniques in a system designed to improve Java performance. This thesis summarizes the work done in design and implementation of this architecture. Preliminary performance data and analysis of design are presented at the end of this paper, and specifications of the implementation as well as algorithms used are found in the appendices.

Java[1] is a programming language developed in recent years by Sun Microsystems. Java was created as a language for the future: a secure, readily extensible, platform independent, portable language. Its objected-oriented nature, strong type verification, automatic memory management, and portability have helped Java spread in popularity. While originally designed for embedded systems, Java has found overwhelming acceptance among Web applications. Virtually everything, from e-commerce applications to traditional standalone applications to computationally intensive numerical analysis tools, is now being developed in Java. As the world becomes more and more immersed in the Internet, Java will continue to grow in importance.

[1] Java™ is a registered trademark of Sun Microsystems, Inc.

Java does, however, have its shortcomings. Current technology is limited, and the very characteristics that make Java an appealing language for the future are the ones that make Java slow and intolerable in today's applications. Java programs are typically compiled into bytecodes and then interpreted by a virtual machine. Newer Java Virtual Machines (JVM) incorporate a lightweight compiler that is able to dynamically compile the bytecodes into machine code to speed up execution. This is known as Just-In-Time compilation (JIT). Because compilation is performed at execution time, the compilation overhead must be kept to a minimum. As a result, JIT is usually not capable of intensive optimizations. Some developments have also been made to allow bytecodes to statically be compiled into machine code before actual execution takes place. In contrast to JIT, this is known as Ahead-of-Time compilation (AOT). AOT is very much identical to traditional compilers except that it is applied to Java. Since compilation time is not part of performance, AOT is able to perform more intensive optimizations. While AOT is able to offers faster execution, AOT is limited in its flexibility. Java allows dynamically loaded classes, and unless the AOT produces code to duplicate JVM functionalities, AOT programs will be limited in capabilities as well as portability. While JIT and AOT are both good approaches, neither is entirely satisfactory.

Traditional programs are typically compiled at the end of development. No additional analysis or compilation is performed at runtime because runtime information cannot be calculated during static compilation. AOT also follows this process like any other traditional compiler. On the other hand, JIT is compiled at runtime and has access to runtime information to allow customization and optimization, but because such analysis

can be intensive, JIT generally avoids almost all types of optimization and analysis at runtime. There are many advantages to runtime code analysis and compilation. If programs could be analyzed during runtime and modified, programs could be specialized according to the calling context and offer additional efficiency in execution. When programs are compiled statically, the compiled programs must be generic, and optimizations for specialized arguments and calling contexts cannot be incorporated without assuming a specific application or use. To specialize programs for various calling contexts, machine codes must be dynamically generated, and some compilation must be delayed. However, it is not practical to delay all of the compilation until runtime in the manner of JIT; compilation is a computationally intensive process and incurs significant performance degradation. Therefore, this research has chosen to stage the compilation. Programs are statically analyzed and partially evaluated, similar to compilation, but the actual machine code will not be generated until runtime. By making use of knowledge learned during static analysis, dynamic runtime code generation can efficiently generate machine code with little overhead. Java Execution-time Transformations (JET) is an alternative JVM architecture that allows staged compilation analysis and performs efficient code generation and specialization through procedure cloning at runtime.

# 2 Background

In recent years, many research projects have focused on improving Java performance; JIT's have been developed on most major architectures and JVM's. The most interesting JIT approach is Sun's HotSpot. HotSpot dynamically analyzes "hotspots" or critical areas in the program, and adaptively optimizes the executed code. JET is similar to HotSpot in

that optimizations are made automatically and dynamically, but instead of delaying all analysis and compilation until runtime, JET divides the work into two phases and minimizes the amount of computation involved at runtime. Although not a JIT, JET works closely with the JIT to extend the capabilities of the JVM.

AOT's have also had great success in improving Java performance. Some researchers approach AOT by translating Java to C (or C++ or Fortran) and then using a robust C compiler to generate efficient machine code. This includes Java source to C source translators such as JCC (Shaylor), and Java bytecode to C source converters such as Toba (Proebsting et al.) and Harissa (Muller et al.). Another approach is to generate machine code directly from Java bytecodes. Some examples of this approach include Marmot (Microsoft Research) and TowerJ (Tower Technologies). Since Java programs are able to dynamically load classes, it is not possible for an AOT to statically analyze all possible paths of execution and produce a single executable for every program. Some of these approaches solve the limitations of AOT by providing and linking libraries to provide JVM functionalities; others simply limit the types of programs that can be compiled or translated. Unlike AOT's, JET is an extension to the JVM and does not produce machine code that can be executed without a JVM.

Compiler research projects involving staged compilation, deferred code generation, partial evaluation, and runtime specializations are also closely related to this project. Runtime code generation in the form of self-modifying code was actually widely used in early systems when memory was expensive. Fast executables could be generated from a

compact representation. While portability, complexity, and changing technologies caused runtime code generation to fall from favor, runtime code generation is still a valuable and useful technique (Keppel, Eggers, and Henry).

Many staged compilation and runtime code generation architectures have been developed. To perform dynamic code generation and specialization, a design must be able to determine which variables and functions to specialize. A popular approach is to augment the source language with annotations to aid and direct runtime specializations and code generation. Examples of this declarative approach include Fabius for ML (Leone and Lee), and Tempo (Consel et al.) and DyC (Grant et al.) for C. Fabius is a dynamic code generator for a functional subset of ML and is one of the earliest examples of dynamic code generation. Fabius uses function currying to determine what variables and functions to specialize. DyC's code generation process is very similar to the code generation process of JET. Given an annotated C program as input, DyC produces an executable program consisting of a static code portion and a generating extensions (GE) portion. This executable program is the intermediate representation (IR) between the static analysis and the dynamic generation phase. In the JET system, the IR is not a piece of machine code. The JET IR is a binary data structure wrapped in a Java classfile attribute. The GE, DyC's dynamic specializer, is custom produced for each function and included in the IR. The JET specializer, on the other hand, is part of the system architecture and is not generated during static analysis. Both DyC and JET specializers are invoked during runtime to produce dynamically specialized machine code.

Even further on the user involvement spectrum is `C and tcc (Poletto et al.) `C is an extension of C that was designed specifically for dynamic compilation; tcc is the compiler for `C (Poletto et al.). `C programmers must explicitly design and declare functions and variables to be specialized. The biggest distinction between these existing systems and JET is that JET does not require any form of user specification. All annotation required by the JET runtime system are generated automatically during static analysis. Once a classfile has been statically analyzed and appended, the entire code generation and specialization process is transparent to the user.

# 3 Motivation

This thesis project is funded by Compaq Computer Corporation as part of the MIT VI-A internship program. Since the work is funded by a corporate organization, the project must not only satisfy an academic, research interest but also provide economic value to the company.

Compaq is interested in this project to help promote sales of its servers. Java is becoming an important means of developing server applications, and to be able to sell servers, especially web and e-commerce servers, the servers must be capable of running Java. By exploring alternative approaches to Java execution and possibly improving speed and performance of Java programs on its servers, this work can give Compaq an edge in promoting and marketing their servers.

From an academic perspective, runtime code generation and automatic specialization through procedure cloning is an interesting compiler research topic. Ideas and understanding gained through this project can help improve general compiler technology. When applied to Java, this is especially interesting because it seeks to fully optimize Java and allow Java to compete in performance with other classical languages such as C/C++ and Fortran.

# 4 Design

The goal of this research was to develop a Java environment that is capable of dynamically generating and optimizing code. From initial evaluations explained previously, the staged compilation approach was chosen. Staged compilation combines the best of both worlds. Heavy computation overheads are accounted for during a static analysis phase, and the runtime dependent optimizations are performed at runtime to allow the fullest range of possible optimizations. In compliance with the JET nomenclature, the static analysis phase is known as Turbo, and the runtime optimization phase is known as Afterburner. A graphical representation of the design can be found in Figure 1.

Once the decision of using staged compilation has been made, the next step is to design a means to integrate the two phases. The JVM specifications allow for implementations to extend functionality by attaching attributes to the Java classfile. By wrapping the JET information in a classfile attribute, the JET system can modularize the phases and allow classfiles that have been put through the static analysis phase (classfiles that have been

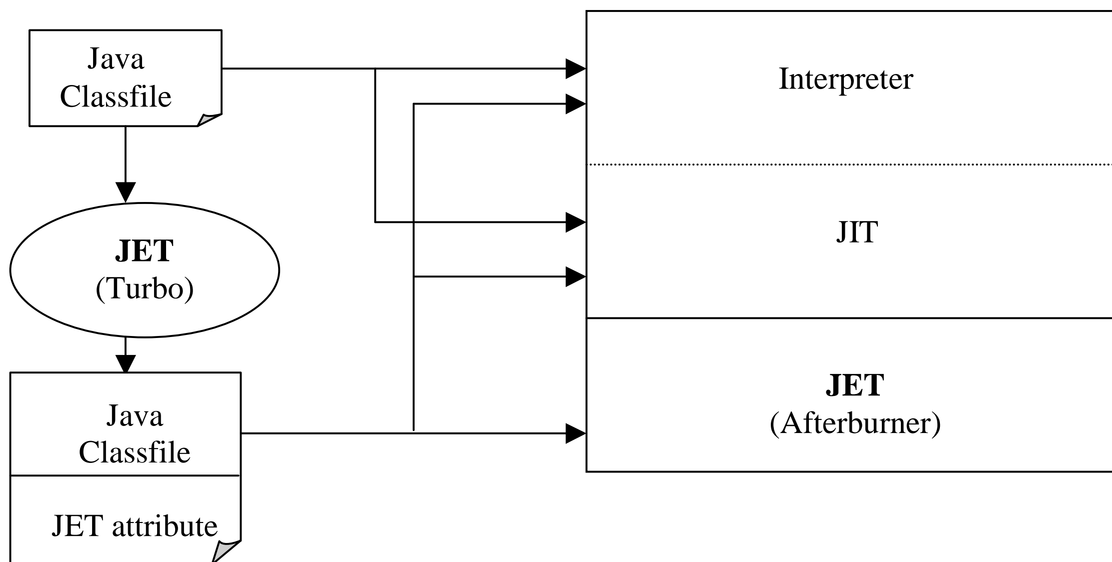


**Figure 1 JET Stages and Integration**

JET-ified) to be usable on JVM environments that do not support the JET architecture. System that do support the JET architecture and have an implementation of Afterburner will be able to make use of the JET information and provide better performance. This approach helps keep the Java classfiles platform independent while extending the functionality to include JET optimizations.

In designing the intermediate representation to encompass the information required for JET optimizations, a modified static single assignment (SSA) value graph and control flow graph (CFG) was chosen. The SSA and CFG are data representations commonly used in developing classic compilers. The CFG seeks to summarize the execution paths and sequences that are possible through a program, and the SSA represents the use and definition of values that make up the program and the instructions to be generated. Together the CFG and the SSA help create a compact flexible representation of the program that allows easy manipulation and optimizations to happen. A complete view of

the JET IR can be found in appendix A. The JET IR extends the classic SSA and CFG with the addition of specific optimization information used in JET.

## 4.1 Turbo

The static analysis portion of JET is codenamed Turbo. Jeremy Lueck, another student in the MIT VI-A program, worked on this portion of JET as his Master of Engineering thesis. Turbo was adapted from an AOT currently under development by Compaq's System Research Center called Swift. Turbo takes in Java classfiles, performs analysis, and outputs a new Java classfile. The new Java classfile contains a JET classfile attribute following the JET IR specifications and containing all the information required to help runtime JET optimizations to be performed efficiently and with minimal overhead.

## 4.2 Afterburner

The Afterburner represents the dynamic code generation, optimization, and specialization phase of the JET architecture. Afterburner extends the JVM to support the JET architecture. Afterburner performs efficient runtime code generation and analyzes execution of methods in order to perform specialization and optimization where possible. The Afterburner itself can be broken up into three stages: parser, generator, and specializer. When the JVM loads classfiles with a JET attribute, the JET parser will be invoked to load the JET IR. Following successful retrieval, a non-specialized code generation is created. Future execution of JET generated code will trigger specialization and optimization analysis. If specialized optimization is possible and worthwhile, a new

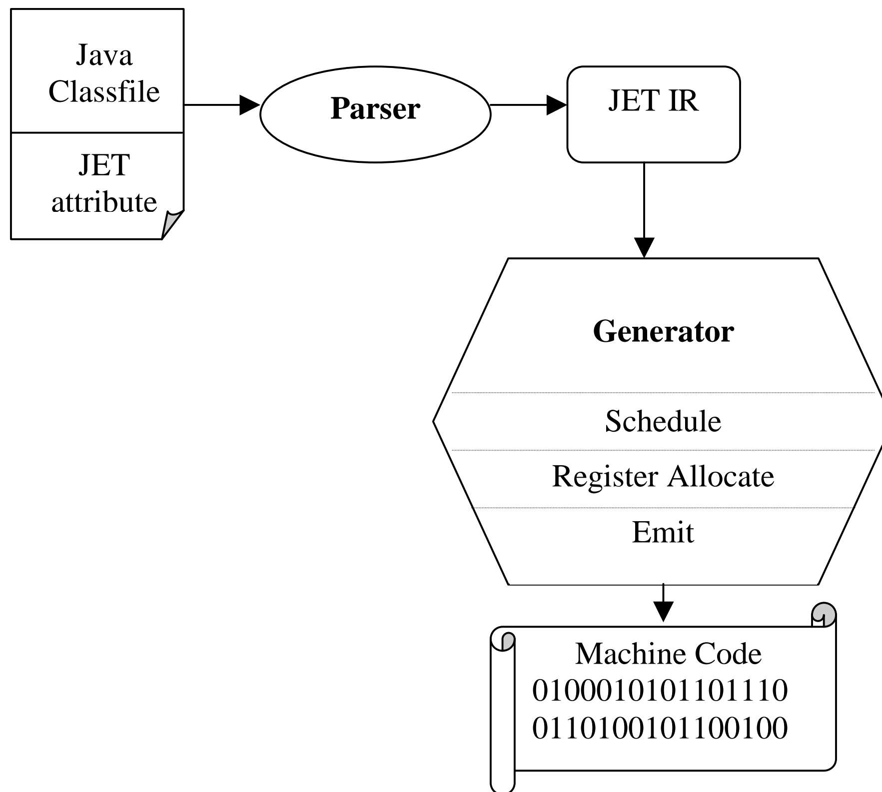


**Figure 2 JET Code Generation Process**

specialized version of the method will be generated. Figure 2 is a graphical representation of this architecture.

The Afterburner is designed to be an extension of the basic JVM. Thus, to accomplish the implementation and design of this system, a working commercial JVM and source code must be available. As this research is part of the MIT VI-A internship, Compaq was kind enough to provide this project with Srcjava. Srcjava is a Java 2 compliant JVM (Java 2 SDK Documentation) currently under development by Compaq's System Research Center in Palo Alto, California. Srcjava is written in C and Assembly, and is developed for Compaq's Tru64 Unix operating system on the Alpha architecture.

### 4.2.1 Parser

The parser reads in the JET attribute and creates JET objects and METHOD objects. The JET attribute is a binary file structure that represents the JET IR. The JET attribute was designed so that it would be easy to load, and the JET IR was designed so that it would be flexible.

The JET attribute is laid out in a manner that allows the parser to read in the attribute as an array of bytes and cast the region of memory into a JET structure. The pointers in the attribute are made so that they are relative offsets from the beginning of the attribute. After the parser casts the region of memory as a JET structure, it then walks through the JET IR converting relative pointers to absolute pointers.

While this process allows quick and easy loading, some assumptions are inherent and should be taken into consideration for future implementations. This implementation was developed on the Alpha architecture, which uses a little-endian byte-order. The Java classfiles are written using a big-endian byte order because the Sun architecture is big-endian. While the choice of big- versus little-endian is not important, consistency is important. Choosing a byte ordering that is consistent with the underlying processor architecture will allow the parser to load more efficiently; however, to widely deploy JET, all implementations must conform to a byte-ordering standard regardless of the underlying architecture.

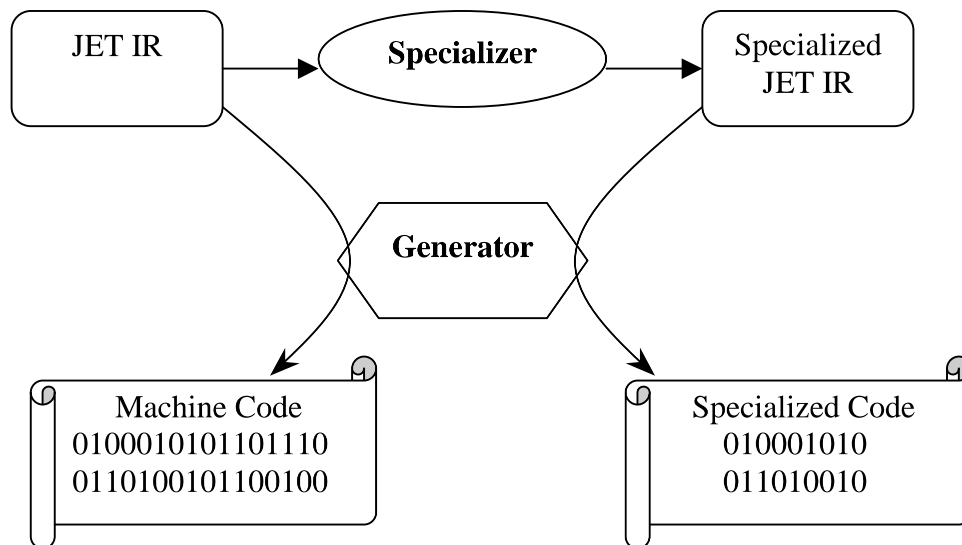


**Figure 3 JET Specialized Generation Process**

### 4.2.2 Generator

Once the JET IR has been created, it is kept for future reference. Copies are made and passed to the generator and the specializer. The generator and specializer operate by modifying the JET IR; making copies allows the generator and specializer to modify the IR as needed without requiring the parser to reload the JET attribute. This specialized generation process is summarized in figure 3. The generator has to perform three tasks: code scheduling, register allocation and code emission. The output from the generator is a standard piece of machine code that can be executed like any other program. The initial code generation for a method will produce machine code that is not specialized. This machine code is registered with the JVM in exactly the same manner as JIT generated machine codes.

#### 4.2.2.1 Scheduler

The current implementation implements a dummy no-op scheduler. Turbo, the preprocessing stage, has performed scheduling prior to this point; thus it is unnecessary to again perform scheduling during code generation to obtain well-scheduled code. The design does, however, suggest that a more sophisticated runtime scheduler may further improve the performance of some types of code. It is not clear at this point, how the trade off between extra scheduling overhead and further optimized machine code compare. Runtime scheduling is likely to be most beneficial in code that is specialized heavily. A specialized code generation will have fewer instructions compared to the non-specialized code generation, and certain constraints may be removed so that instructions may be rescheduled and further improved. The time constraints for this project did not allow design and implementation of a code scheduler.

#### 4.2.2.2 Register Allocation

Register allocation is divided into 2 phases: there are variables that are global to the entire method and used across code blocks as well as variables that are local to code blocks. The global variables include variables that must be allocated in designated locations to comply with calling standards such as arguments to procedure calls and return values. Local variables without restriction on locations may be allocated in between global variables or in another free registers. Local variables that do not conflict with each other can share the same register. The JET system takes advantage of this distinction between global and local variables by splitting up the allocation so that the majority of the computationally intensive tasks are performed during the Turbo phase.

In general, a two phased register allocation is somewhat inefficient in its use of registers; however, the JET system is designed so that Turbo performs a Limit phase where Turbo tries to maintain that the register pressure, the number of live registers needed at any point in the code, is at or below the number of register available. This property allows JET to efficiently break up register allocation into a global and a local phase. Turbo is responsible for allocating the global variables and procedure arguments according to the JET calling standard. Turbo is also responsible for grouping of local variables that do not conflict and can potentially share the same register. This is known as symbolic allocation, where each symbolic register is a color that is assigned to that group of variables.

During the runtime Afterburner phase, these symbolic registers are matched up with real physical registers. An attempt is first made to match symbolic registers up with physical registers that were allocated to global variables. This effort reduces the number of free, unrestricted physical registers needed and is possible if the involved variables are not used or not live over the same range of instructions. Symbolic registers that were not matched up with global registers are then allocated into the remaining available physical registers during the final allocation phase. The final allocation phase, whose name is one-pass allocation, takes a linear pass over the remaining unallocated variables and assigns physical registers to these variables.

There is the possibility that not enough physical registers exist to accommodate all the symbolic registers. In this situation, spill code to write variables into stack or memory

locations must be inserted into the generated code to partially relieve the register pressure. The implementation does not actually insert spill code during the register allocation phase but instead marks the values to indicate that their assigned location is on the stack and leaves the spill code insertion for the code emission phase.

The JET IR is designed so that each SSA value node includes a location field. The location field indicates where in the machine this value is stored. For example, the integer constant zero has a location of R31. Global variables are assigned physical register locations during the static phase. The symbolically allocated variables have symbolic register locations. Values nodes that do not emit instructions have a null location because they do not require registers to hold its value. When symbolic registers are matched up with physical registers during register allocation, all value nodes with that symbolic register get their locations updated from the symbolic register location to the allocated physical register location. Value nodes may also be allocated to stack slot locations when the number of available registers have run out.

This division of labor allows runtime register allocation to require only a few passes over the set of instructions. Register allocation is typically computationally intensive and a major bottleneck in dynamic compilation. JET is able to limit the amount of computation necessary at runtime and keep the register allocation overhead to a minimum. A detailed explanation as well pseudo-code related to the register allocation design can be found in Appendix D.

Some assumptions are inherent in the design of the register allocation process. First of all, many of the global variables are assigned according to a JET calling convention, which is the same as the Alpha calling convention. Future implementation of JET will need to also support the same calling convention within the JET architecture but may use its native calling convention for code outside the JET architecture. The next assumption is that there are 32 floating and 32 integer registers available. This is fairly typical among modern architectures. If more or fewer registers are available, then some form of emulation will be necessary. While these assumptions make the design somewhat less generically applicable, some basic assumptions are necessary, and these assumptions are based on fairly common trends in industry. These limitations and assumptions are maintained so that a JET classfile can be created once and executed anywhere an implementation of Afterburner is available.

#### 4.2.2.3 Emitting Code

The last component of the generator is the code emitter. The code emitter translates JET opcodes into actual machine instructions. The spill codes are inserted and the generated code stubs are registered with the JVM. Most SSA values can be translated directly into machine instructions; however, flow control instructions that require relative offsets such as branches must first be emitted with null offsets as the final code positions may not yet be known. Once all the instructions have been generated, relocations get applied so that the references and offsets are accurate and correct. The complete list of JET opcodes and its descriptions may be found in Appendix B.

Code is emitted by translating the opcode of the value node into an actual machine instruction. If the opcode is an instruction that requires additional arguments, the instruction must be emitted such that the correct registers are used as arguments. These registers are found by looking up the location of the argument value nodes. The output of the instruction must also be placed in the correct register by looking up the location of the current value node. If a value node's location is on the stack, then spill code must be generated and a temporary register must be employed. Since an instruction may require at most two arguments and the output register may reuse one of the input registers, the worst case, a situation in which both arguments as well as the output reside in stack locations, will require at most two temporary registers. The JET calling standard always reserves one temporary register that is not assigned to variables. The other temporary register is formed by using the return address (RA) register. RA register is used to keep track of the return address for a particular subroutine call. RA is saved on the stack whenever a subroutine code segment makes additional, nested subroutine calls. The scheme of using RA as a temporary register adds the constraint that RA will also need to be saved on the stack if there exists an instruction that requires two temporary registers. A detailed explanation of the JET calling standard can be found in Appendix C.

When classfiles are loaded by the JVM, a method table is created to map code stubs to the methods. This is necessary for the correct inheritance of objects. When methods are invoked, the actual executed code must first be looked up in the method table. If the caller method is a piece of generated code, then the process can be streamlined so that the caller method calls the callee code directly. This is possible if the method being invoked

is static or if it is the only one of its type and signature. For example, methods that are not static but have not yet been known to be overridden can be called directly. This optimization requires that each call site be registered so that if the generated code changes or if methods are overridden, these direct call sites can be updated to call the correct new code. This process allows the JVM to switch easily between interpreted code and JIT generated code. JET generated machine code behave similarly to JIT generated code, and must also be registered with the JVM before the JVM can make use of the JET generated code. Registration of JET generated code can be done using the same process as for JIT generated code.

### 4.2.3 Specializer

Upon the first execution of a JET generated call site, the specializer will be invoked before execution. The call site is set up to call a code stub that executes the specializer instead of the actual method. The specializer analyzes executed methods and the calling conditions to see if specialization is possible and worthwhile. If so, the specializer creates specialized versions of the JET IR and sends the JET IR to the generator. The specializer also registers the newly created specialized code with JET and updates the specialized call site.

The generator emits indirect subroutine calls. An intermediate specializer invocation code stub is called before the subroutine code is executed. The specializer stub determines whether specialization is possible, and if so, whether a specialized generation of this instance exists already or a new specialized generation must be created.

For each method, Turbo is responsible for determining what input arguments would be helpful as constants. Turbo also marks certain SSA values as constants if they're known as constants. During the specializing phase, the specializer determines which input arguments are constants for each executed subroutine call. The specializer then computes the intersection of the set of arguments that the method wants to have as constants and the ones that actually are constants. IF the resulting set is null then specialization is not possible and the indirect call through the specializer is replaced with a direct call to the callee. If the resulting set is not null, then specialization is possible. This intersection set will be known as specialization set.

Specializations with the same specialization set are considered the same category or type of specialization, as they will have similar specialization effects. If the specializer determines that specialization is possible, it searches through a linked list of existing specialization categories for that method. If one exists, then it continues by looking for a specialization instance with identical arguments. If such an instance exists, this particular specialization has been performed in the past and the specialized code can be reused. If no such instance exists, then a new specialization must be performed. The newly generated instance is registered under the appropriate category following generation. If no such specialization category exists, then a new category is created and a new specialization instance is generated. The categorization makes it easier to determine if a particular specialization instance exists already. This cataloging and registration process is similar to the code registration process of the JIT and the JVM.

While a number of various optimizations are possible during runtime specialization, this research focused on constant propagation and procedure cloning. When a call stubs invokes the specializer and the specializer determines that specializations are possible and needed, constant propagation is performed. Using an algorithm similar to constant propagation algorithms in classic compilers, the specializer generates specialized clones of the procedure being invoked. Because this specialization is performed at runtime, the system knows the values of the arguments. The specialization category determines which arguments are considered as constants in the function being specialized. For each argument that is considered a constant, the specializer propagates the constant by finding descendent values that use constants as inputs. If all arguments to an instruction are constants, the output of that instruction is also a constant and its value can be evaluated and stored as a constant. Constant values do not require code to be emitted unless the constant values are used by instructions that do not have constant values. Branches in the code can be eliminated if the condition variable can be determined to be a constant. A more detailed overview of the constant propagation algorithm used can be found in Appendix E.

This research limited the effects of specialization to strictly elimination of code. This limitation was put in place to minimize the complexity of implementation. While techniques such as loop unrolling are typically used in conjunction with constant propagation, such techniques were not utilized. In defense of this decision, loop unrolling is performed during Turbo for loops that benefit from unrolling. Obviously, additional

and more precise loop unrolling can be performed during runtime specialization and should be considered for the next iteration of this design.

Once the specialized code has been generated, the execution must resume seamlessly. The specializer has already generated the specialized code and registered it as a specialization instance under the appropriate specialization category. The original call site has also been patched so that future execution of that call site will no longer invoke the specializer call stub but instead invoke the appropriate specialized code. The specializer needs to execute the specialized code and return control to the JVM. Instead of having the specializer handle the execution and deal with all the possible exception scenarios, the easiest approach is to modify the call stack return address from the instruction after the call site to the call site itself. Now the specializer can simply return control to the JVM and be confident that the JVM will be fooled into re-execution of the now updated call site. This re-execution will now invoke the newly generated specialized code, and all will appear normal to the JVM and the user.

# 5 Analysis

Constant propagation is achieved by disseminating knowledge and assuming certain input arguments are constants. Computational instructions that do not involve I/O and memory access benefit from constant propagation because their results can be saved as constants and do not need to be calculated again in future executions. Appendix F illustrates some typical code generations and specializations starting from the Java source files.

To determine the benefit of JET system, a number of relative performance measurements were assessed. Factors considered in performance measurements include file size, code size, and execution speed. A detailed summary of the performance measurements can be found in Appendix G.

JET classfiles are normal Java classfiles with an added attribute. This JET attribute contains all the information required to efficiently generate code and perform specializations. This additional information comes at a price; the JET classfiles are on average thirty-two times as large as a regular Java classfile. For this reason, JET should not be implemented on a system that is constrained by disk storage space.

Benchmark performance tests were executed to evaluate JET performance. The Embedded Caffeine Mark 3.0 benchmark suite was chosen for this purpose. Each scenario is executed multiple times and the average performance values compared. The following five scenarios were executed: classic Java JVM using interpretation on normal classfiles, JIT enabled JVM on normal classfiles, classic Java JVM using interpretation on JET classfiles, JET enabled JVM with dynamic code generation but no specialization, and JET enabled JVM with dynamic code generation and specialization.

While JET classfiles are much larger than classic Java classfiles, the size does not affect the execution performance. In most benchmarks the performance is nearly identical. The Sieve benchmark actually benefited from a 30% improvement due to the larger JET classfile. It is not clear why the sieve benchmark benefits from a larger input file size.

JIT provide an average 300% performance improvement compared to classic, interpreted Java. The JET enabled JVM without specializations showed an additional 180% performance improvement over the JIT, and a 550% improvement over classic, interpreted Java. This analysis suggests that the larger file sizes of staged compilation have paid off handsomely in performance improvements. JET with specializations had only an insignificant advantage over the JET without specializations. In fact, on the Floating benchmark, JET with specialization was actually slightly worse than the JET without specialization. All JET performance numbers include dynamic compilation and specialization overhead.

# 6 Conclusion and Future Work

It appears that specializations were not extremely effective. A possible explanation is that the amount of analysis carried out was insufficient or not widely applicable. Perhaps a more ambitious specialization policy would allow more methods to be specialized and a broader effect and performance improvement can be observed. Deeper analysis of how and why programs benefit from specialization will help pave the way to new and more in-depth types of specialization. Currently, the only specialization performed is constant propagation. Without dynamic loop unrolling, it is difficult for constant propagation to produce dramatic performance improvements.

If a more relaxed specialization policy is implemented, more arguments can be treated as constants and more specialization would possible. This new design could attempt to

specialized on variables that are not guaranteed to be constants. If a variable is seen frequently with the same value, it could be treated as a constant and used to specialize. This approach allows more specialization to occur but incur the additional complexity that a verifier code stub must be produced to ensure that a variable that is assumed to be constant does not change value. While it is not clear that specialization will provide much more benefit, using a more relaxed specialization policy will allow more specialization to occur and a more definitive answer on the benefits of specialization can be obtained.

Further constant propagation can be achieved if procedure calls can be eliminated through constant propagation. This would require support from the static analysis phase. If a method can be determined to have predictable results given constant arguments, then that method can be executed during specialization and its result can be stored as a constant in the caller. Since the analysis to determine if a method is predictable will likely be highly involved, it is most suited for the Turbo stage. Even without Turbo support, there exist a few common runtime procedures that are currently known to be predictable and can be dynamically resolved into constants with only a few minor modifications to Afterburner.

Another idea for specialization is to eliminate unnecessary synchronization overhead. If a synchronized object is referenced by only one object, then there is no need to lock the synchronized object. Only one reference of the synchronized object exists and no possible race condition exists.

Java is a powerful but young language. There is lots of room for improvement. This research applied new approaches to execution of Java programs. The combination of static compilation and dynamic optimization combine to make a powerful staged compilation process that lends itself easily to optimization, specialization, constant propagation, and procedure cloning. These techniques were united to create JET. Dynamic code generation through staged compilation proved to be an excellent technique to improve Java performance; however, the types runtime specialization implemented were not extensive enough to produce definitive performance improvements. Insights gained from this experience can help refine the next iteration of JET and perhaps lead to further increases in performance.

# Appendix A: JET IR Specification

The following document contains the specification for the current version of the JET internal representation binary output (JET IR).

JET files are essentially Java class files with an added class attribute. This attribute is called "JET" and contains the JET structure that will be described below.

JET attributes can be outputted in either *little-endian* or *big-endian* order. Since the initial implementation was development on the Alpha architecture, and the Alpha architecture is little-endian, the little-endian byte-order was chosen. The choice is not important; however consistency across various implementations is important.

The bytes in a JET IR are laid out in an order that is very similar to how these structs would be laid out in program memory using a memory-manipulation language such as C. This should look exactly like the automatic memory alignment for structs that is performed by the C compiler. A relative "pointer" in a JET attribute is the file offset of the structure within the file.

The process for reading in a JET attribute:

1. Find the JET attribute from the class file parser in the JVM.
2. Grab the offset to the beginning of the JET attribute
3. Iterate through the different structures of the JET attribute casting file offsets into the structs and fixing pointers.

Since relative pointers are file offsets, they have to be converted into absolute pointers using the following calculation:

```
ptr = (T *)((long)starting_memory_location + (long)ptr)
```

T* is a pointer to some JET structure type and `starting_memory_location` is the memory address of the beginning to the JET structure (or the pointer address of the `malloc`-ed memory in step 2 above). The *long* data type is used for pointer calculations since the Alpha architecture uses 64-bit addressing.

## *JET*

Each JET attribute contains the following structure:

```
JET {
   u2           methodCount;
   methodInfo[] methods;
}
```

The items in the `JET` structure are as follows:

**methodCount**

This item holds the number of methods that are contained within this JET attribute. Should be the same as the number of methods contained in the classfile.

**methods**

The method item is a table of methods that contain the intermediate representations (IR's). The format of a method and its IR are defined by the `methodInfo` structure. The size of this table is defined by `methodCount` item.

## *Method Information*

Every method has an intermediate representation structure contained within `JET`. This IR consists of two separate structures with pointers between them. The first structure is the control flow graph (CFG) and the second is the value graph generated for static single assignment (SSA) form. The CFG is made up of basic blocks connected by successor and predecessor pointers. The SSA form is made up of value nodes that represent instructions and connected by pointers to other value nodes that make up the parameters of that instruction.

Each `methodInfo` item must have the following structure:

```
methodInfo {
   u2          blocksCount;
   u2          valueCount;
   u2          paramCount;
   value**     params;
   bitvec*     rtConstArgs;
   blockInfo[] blocks;
   valueInfo[] values;
}
```

The items in the `methodInfo` structure are as follows:

**blocksCount**

This item contains the number of basic blocks contained within the CFG for this method.

**valueCount**

This item contains the number of individual values that make up the SSA value graph for this method.

**paramCount**

This item contains the number of parameters to this method. This number includes the object references for instance methods (pointer to *this* object).

**params**

This item contains a list of values that are the parameters to this method. The length of this list is specified by the `paramCount` item.

**rtConstArgs**

This item contains a `bitvec` of size equal to the number of arguments to this method. Item `n` in `rtConstArgs` will be set if knowing that the `n`-th argument to this method is a runtime constant will help in specializing this method.

> **NOTE**: The size of this set will be equal to `paramCount`. This includes the instance object for instance methods. The size of this set will not be the same as the number of inputs to a method call value.

**blocks**

The `blocks` item contains a table of basic block information for a CFG. The format of these blocks is defined by the `blockInfo` structure. The length of this table is established by the `blockCount` item. The `blocks` array is ordered in the way that they should be written out during code generation.

**values**

The `values` item contains a table of values within the SSA value graph. The format of these values is defined by the `valueInfo` structure. The length of this table is established by the `valueCount` item.

## *Block Information*

Each basic block within the CFG contains information regarding the type of control within the block, the successor blocks, pointers to values used in this block, and liveness information to aid in the allocation of registers. Some block types need special added information in order to generate code for that block. Thus, there are three types of block output: `blockInfo`, `excBlockInfo`, and `multiBlockInfo`. The default basic block follows this structure:

```
blocksInfo {
   u1           type;
   u1           controlType;
   u2           valueCount;
   u2           predecessorCount;
   u2           successorCount;
   u2           index;
   u1           isLoopHead;
   valueInfo*   valueRoot;
   valueInfo**  values;
   blockInfo**  predecessors;
   blockInfo**  successors;
   bitvec*      liveOut;
   bitvec*      liveStart;
   bitvec*      liveEnd;
   bitvec*      liveThrough;
   bitvec*      liveTransparent;
   auxInfo*     aux;
}
```

The items in the `blockInfo` structure are as follows:

**type/control_type**

The type and control_type items contain two sets of type information. This information is created when the CFG is created inside of the Swift compiler.

The tags for the `type` and the `controlType` of the block are listed in the following table:

| ***Type*** | ***Value*** | ***ControlType*** | ***Value*** |
|---|---|---|---|
| NORMAL | 0 | SIMPLE | 0 |
| HANDLER | 1 | IF | 1 |
| EXTRA | 2 | SWITCH | 2 |
| ENTRY | 3 | THROW | 3 |
| NORM_EXIT | 4 | FAULT | 4 |
| EXC_EXIT | 5 | JSR | 5 |
| GRAPH_EXIT | 6 | RET | 6 |

**valueCount**

This item contains the number of values that make up the instructions for this basic block.

**index**

This is the numerical identifier of this `blockInfo` among the `blocks[]` of the `methodInfo` that holds this `blockInfo`.

**valueRoot**

This item contains a pointer to the value that decides the control flow out of this block.

**values**

This item contains a table of pointers to pointer to the values from the SSA value graph used in this basic block. The size of this table is determined by the `valueCount` item.

**successorsCount**

This item contains the number of successor blocks to this block within the CFG.

**successors**

This item contains a table of pointers to successors to this block. This table consists of a list of pointers to `blockInfo` structures that contain the basic block successors. The size of this table is determined by the `succesorCount` item.

> **NOTE:** In the case of IF blocks, the successors are ordered such that the fall-through block is the first successor, and the branch-to block is the second successor.

**liveOut / liveStart / liveEnd / liveThrough / liveTransparent**

These items contain bit vectors that represent the values that are live in the various categories for a block.

## *Value Information*

Each node in the value graph represents an operation and a value. Nodes are connected if a node if used as parameters in another node. Each method contains an array of values. The structures of these values have the following format:

```
valueInfo {
   u1          op;
   u1          type;
   u2          pc;
   u2          index;
   u2          paramCount;
   u2          useCount;
   u1          locationType;
   u1          locationFlags;
   u2          location;
   u1          isSpilled;
   u1          isRTConst;
   bitvec*     unavailRegs;
   valueInfo** uses;
```

```
    valueInfo** params;
    blockInfo*  container;
    auxInfo*    aux;
}
```

The items in the `valueInfo` structure are as follows:

**op**

The `op` item contains a number that represents a particular operation within an IR stage. These ops are defined in `Appendix B`.

**type**

The `type` item contains a number that represents a type for a value. These types are listed in the table below.

| type | value |
| --- | --- |
| `VOID` | 0 |
| `CC` | 1 |
| `FCC` | 2 |
| `TUPLE` | 3 |
| `BOOLEAN` | 4 |
| `BYTE` | 5 |
| `CHAR` | 6 |
| `SHORT` | 7 |
| `INT` | 8 |
| `LONG` | 9 |
| `FLOAT` | 10 |
| `DOUBLE` | 11 |
| `ARRAY` | 12 |
| `STORE` | 13 |
| `NULL` | 14 |
| `OBJECT` | 15 |

**pc**

The `pc` item contains the program counter for this particular instruction. This information is saved for use by the exception handling mechanism.

**index**

This is the numerical identifier of this `valueInfo` within the `values[]` item of the `methodInfo` that holds this `valueInfo`.

**locationType**

The `locationType` item contains place where the memory for this value should live. The following table describes the different `locationType`s.

| **locationType** | **value** |
|---|---|
| `VOID_LOCATION` | 255 |
| `INT_REG` | 0 |
| `FLOAT_REG` | 1 |
| `STACK_TMP` | 2 |
| `STACK_IN_ARG` | 3 |

**locationFlags**

Contains added information about the memory location of this value, such as whether it is assigned to a symbolic or physical register. The following table describes the different `locationFlags`. The value of a `locationFlags` item is a bit mask of the values in the table.

| **locationFlags** | **value** |
|---|---|
| IS_REGISTER | 1 |
| IS_CALLEE_SAVE | 2 |
| IS_RESERVED | 4 |
| IS_SYMBOLIC | 8 |
| IS_LOCAL | 0 |

**location**

The number associated with the physical register, or symbolic register, or the stack location that holds this value. This item along with `locationType` and `locationFlags` gives all the possible memory locations for a value within a method.

**isSpilled**

This item is set to 1 if this variable is spilled during global or local allocation and needs to be stored to a stack location after its definition and loaded from the stack before every use.

**isRTConst**

This item is set to 1 if this variable is a runtime constant. This information can be used to specialize a method.

**unavailRegs**

This item contains a set of registers should not be assigned to this value because it somehow conflicts with the lifetime of this value. This set could include the scratch registers, if the value is live across a method call, or other physical registers which have been assigned to values and conflict with this value.

**useCount**

The use`Count` item contains the number of values that use this value as an input argument.

**uses**

The `uses` item contains a list of pointers to `valueInfo` structures that contain the values that use this value as an input argument. The length of this list is defined by the `useCount` item.

**paramCount**

The `paramCount` item contains the number of argument values to the operations described within this value.

**params**

The `params` item contains a list of pointers to `valueInfo` structures that contain the arguments to the operation described within this value. The length of this list is defined by the `paramCount` item.

**container**

The `container` item contains the `blockInfo` that holds this value. Thus, the `values` item of this `container` will also contain a pointer back to this valueInfo.

**aux**

The `aux` item contains any auxiliary information for a value that cannot be stored within the `params`. This information includes constants, field and method references, and argument numbers. This item is of variable length and depends upon the type tag that is defined by the `auxInfo` structure.

## *Auxiliary Information*

Every `valueInfo` and `blockInfo` structure contains an auxiliary item for holding any extra information. This information includes constants, field and method references, and argument numbers. This item is of variable length and depends upon the `tag`.

**NOTE:** Auxiliary information structs are shared amongst all of the methods in the JET file, much like the constants in the constant pool. When relocating pointers, make sure to relocate them once instead of numerous times which may happen if you relocate them on each encounter through the values arrays.

The basic form of all the auxInfo structures is the following:

```
auxInfo {
   u2      tag;
   u1      isResolved;
   u1[]    info;
}
```

The items in the `auxInfo` structure are as follows:

**tag**

The `tag` item contains a number that represents the type of information stored within the auxInfo structure. This tag also gives information as to the length of the info array.

The following table describes the possible tags for an auxInfo structure. The length column tells the length of the info array that can be expected.

| *Tag* | *Value* | *Length* |
|---|---|---|
| `VAL_Void` | 0 | 4 |
| `VAL_Integer` | 1 | 8 |
| `VAL_Float` | 2 | 8 |
| `VAL_Long` | 3 | 16 |
| `VAL_Double` | 4 | 16 |
| `VAL_String` | 5 | 16 |
| `VAL_Fref` | 6 | 32 |
| `VAL_Mref` | 7 | 32 |
| `VAL_BlockVec` | 8 | 16 |
| `VAL_RuntimeProc` | 9 | 16 |
| `VAL_RefType` | 10 | 16 |
| `VAL_FrefOffset` | 11 | 32 |
| `VAL_JumpTable` | 12 | 32 |
| `BLOCK_ExcLabel` | 13 | 16 |
| `BLOCK_IntLabel` | 14 | 16 |

**isResolved**

Since `auxInfo`s are shared between different values, a space is left available in the `auxInfo` struct for determining whether this `auxInfo` has been resolved during the read-in process.

**info**

The `info` item contains the actual data for an `auxInfo` structure. This array is of variable length, and its size is determined by its `tag`. The different types of `info` are described in the next sections.

## *VAL_Void*

```
VAL_Void {
   u2 tag;
   u2 isResolved;
}
```

A `VAL_Void` info structure contains nothing other than the header of the `auxInfo` structure.

## *VAL_Integer, VAL_Float*

```
VAL_Integer, VAL_Float {
   u2 tag;
   u2 isResolved
   i4 val;
}
```

`VAL_Integer` and `VAL_Float` info structures have four bytes for the value of the number. `VAL_Float` values are encoded using the Float.floatToIntBits() method. Numbers are encoded in little-endian byte order.

## *VAL_Long, VAL_Double*

```
VAL_Long, VAL_Double {
   u2 tag;
   u2 isResolved;
   i8 val;
}
```

`VAL_Long` and `VAL_Double` info structures have eight bytes for the value of the number. `VAL_Double` values are encoded using the Double.doubleToLongBits() method. Numbers are encoded in little-endian byte order.

## *VAL_String*

```
VAL_String {
   u2    tag;
   u2    isResolved
   utf8* string;
}
```

`VAL_String` info structures contains a pointer to a null-terminated character array which is the utf-encoded version of a Java string

### *VAL_Fref, VAL_Mref*

```
VAL_Fref, VAL_Mref {
   u2    tag;
   u2    isResolved;
   utf8* class;
   utf8* name;
   utf8* signature;
}
```

The items in a `VAL_Fref` or `VAL_Mref` info structure are as follows:

**class**

The `class` item contains a pointer into the `cpool` table for the name of the owner class that contains this field/method.

**name**

The `name` item contains the name of this field/method.

**signature**

The `signature` item contains the Java signature of this field/method.

### *VAL_BlockVec*

```
VAL_BlockVec {
   u2          tag;
   u2          isResolved;
   u2          blockCount;
   BlockInfo** blocks;
}
```

The items in a `VAL_BlockVec` structure are as follows:

**blockCount**

The `blockCount` item contains the number of blocks within this BlockVec.

**blocks**

The `blocks` item is an array of block numbers that make up this BlockVec. These block numbers are also the offset into the `blocks` table used to create the CFG.

### *VAL_RuntimeProc*

```
VAL_RuntimeProc {
   u2          tag;
   u2          isResolved;
   u2          type;
   auxInfo*    aux;
}
```

The items in a `VAL_RuntimeProc` structure are as follows:

**type**

The `type` item contains the unique numerical identifier of a runtime procedure. The list of types can be found the table below with their aux types.

**aux**

The `aux` item contains any extra information needed by the RuntimeProc. This is a VAL_Integer in the case of creating a new array, otherwise it is a VAL_String that contains the name of a class.

The following table contains all of the possible RuntimeProcs as well as their associated values and the types of their aux fields:

| type | value | aux type |
|---|---|---|
| SYNC_ENTER | 1 | VAL_Void |
| SYNC_EXIT | 2 | VAL_Void |
| NEW | 3 | VAL_String |
| FCMPL | 4 | VAL_Void |
| FCMPG | 5 | VAL_Void |
| DCMPL | 6 | VAL_Void |
| DCMPG | 7 | VAL_Void |
| NEWARRAY | 8 | VAL_Integer |
| ANEWARRAY | 9 | VAL_String |
| MULTIANEWARRAY | 10 | VAL_Void |
| INSTANCEOF | 11 | VAL_String |
| CAST_CK | 12 | VAL_Void |
| IDIV | 13 | VAL_Void |
| IREM | 14 | VAL_Void |
| LDIV | 15 | VAL_Void |
| LREM | 16 | VAL_Void |
| FREM | 17 | VAL_Void |
| DREM | 18 | VAL_Void |
| THROW | 19 | VAL_Void |
| INIT_CK | 20 | VAL_String |
| D2I | 21 | VAL_Void |

| D2L | 22 | VAL_Void |
|---|---|---|
| AASTORE | 23 | VAL_Void |
| SYNC_EXIT_RET | 24 | VAL_Void |

### *VAL_RefType*

```
VAL_RefType {
   u2    tag;
   u2    isResolved
   utf8* name;
}
```

The items in a `VAL_RefType` info structure are as follows:

**name**

The `name` item contains the name of the class that this value is referencing.

### *VAL_FrefOffset*

```
VAL_FrefOffset {
   u2    tag;
   u2    isResolved;
   u2    offset;
   utf8* class;
   utf8* name;
   utf8* signature;
}
```

The items in a `VAL_FrefOffset` info structure are as follows:

**offset**

The `offset` item contains the byte offset of this field reference from the beginning of the object it references in memory.

**class**

The `class` item contains the name of the owner class which this field.

**name**

The `name` item contains the name of this field.

**signature**

The `signature` item contains the Java signature of this field.

### *VAL_JumpTable*

```
VAL_JumpTable {
   u2          tag;
   u2          isResolved;
   u2          type;
   u2          baseIndex;
   u2          tableOffset;
   i4          min;
   i4          max;
   u4          nonDefaultEntries;
   blockInfo*  block;
}
```

The items in a `VAL_JumpTable` structure are as follows:

**type**

This is the type of the Jump Table. The `type` item contains a value of 0 for a `LOOKUPSWITCH` type and a value of 1 for a `TABLESWITCH` type.

**baseIndex**

Index of the instruction that gets the base pointer

**tableOffset**

Offset of the emitted table from the baseIndex instruction

**min**

The minimum value for a label in the jump table

**max**

The maximum value for a label in the jump table

**nonDefaultEntries**

The number of non-default labels in the jump table.

**block**

This item is a pointer to the block that holds the switch statement responsible for this Jump Table.

### *BLOCK_ExcLabel*

```
BLOCK_ExcLabel {
   u2         tag;
   u2         isResolved;
   u2         excCount;
   excLabel* exceptions;
}
```

```
excLabel {
   utf8*      excName;
   blockInfo* handler;
}
```

The items in a BLOCK_ExcLabel info structure are as follows:

**excCount**

The item contains the number of exceptions that can be raised within this block.

**exceptions**

The item contains a table of `excLabels` that hold the information on which exceptions can be thrown in a block and the blocks that contain the handler information.

The items in an `excLabel` structure are the following:

**excName**

This item contains a pointer to the class name for an exception thrown by this block. If a faulting instruction returns to this block with an exception of `excName` type, it jumps to the `handler` block. If the handler catches all exception, `excName` will contain the empty string.

**handler**

This item contains a pointer to the `blockInfo` that holds the handler code for a caught exception of `excName` type.

## *BLOCK_IntLabel*

```
BLOCK_IntLabel {
   u2         tag;
   u2         isResolved;
   u2         labelCount;
   intLabel* labels;
}
intLabel {
   i4          label;
   blockInfo* destination;
}
```

The items in a `BLOCK_IntLabel` structure are as follows:

**labelCount**

This item contains the number of labels that can be used to transfer control in this block.

**labels**

The item contains a table of `intLabel` labels that are the branch determining values. Each successor has an `intLabel`, but the number of labels associated with a successor may be greater than one. The size of this table is determined by the `labelCount` item.

The items in an `intLabel` structure are as follows:

**label**

This item contains the integer that is the label for a switch statement. The successor to branch to for this label is described in the `destination` item.

**destination**

This item contains the block in the `successors` array that should be the destination block of a switch statement if the result of the switch is the value in `label`.

## *Bit Vector*

Each block in the CFG also contains helpful information about the liveness of the variables as execution proceeds through the block. This liveness information is calculated in a final phase of the Swift compiler and passed along into the binary output. The representation chosen to represent these sets is bit vectors. Each value in the set is represented by the integer from the values' `index` field.

```
bitvec {
    u8  length;
    u8* set;
}
```

The items in a `bitvec` structure are the following:

**length**

This item contains the maximum magnitude of integers represented in this set. It is also the number of values in the method containing this set.

**set**

This item contains a list of quadwords that form the bit vector representation of this set.

## *Exception Information*

Each block that can throw an exception must contain a list of `excLabels`. These labels contain the type of the exception, by class name, and a pointer to the block that holds handler for that particular exception. This information is gathered from the exception table in the classfile and passed down with the IR.

```
excLabel {
    utf8*      excName;
    blockInfo* handler;
}
```

The items in an `excLabel` structure are the following:

**excName**

This item contains a pointer to the class name for an exception thrown by this block. If a faulting instruction returns to this block with an exception of `excName` type, it jumps to the `handler` block. If the handler catches all exception, `excName` will contain the empty string.

**handler**

This item contains a pointer to the `blockInfo` that holds the handler code for a caught exception of `excName` type.

## *UTF8-String constants*

An UTF8-String contains a list of one-byte utf-encoded Unicode characters followed by a null-terminating character. One can consider the `utf8` type to be the same as an `u1` or a `char`.

# Appendix B: JET Opcodes

IN_ARG 1
: Input argument to subroutine

SELECT 2
: Abstract value representing the result of a memory operation.

COPY 3

NOP 4

PIN 5
: Used as a fake input to operations that need to be pinned within a certain region of the CFG.

SPILL 6
: Store to stack location (used for spilling)

RESTORE 7
: Load from stack location (used for spilling)

ADD 8
SUB 9
MUL 10
DIV 11

AND 16
OR 17
XOR 18

NEG 19
: Arithmetic negation

NOT 26
: Logical not

PHI 31
: Abstract value representing a node whose value differs depending on the instruction path executed to reach this node.

IF 32
: Abstract value representing flow control.

REAL_RETURN 33

SWITCH 34
: tableswitch or lookupswitch

NULL_CK 35
: Null check.

LENGTH 37
: Array length.

GET_FIELD_ADDR 45
: Get object field address

INVOKE_VIRTUAL 46
INVOKE_SPECIAL 47
INVOKE_INTERFACE 48
INVOKE_STATIC 49
INVOKE_DIRECT 50

INSTANCEOF 51
: Object type verification.

RT_CALL 59
: Generic call to a runtime routine

EXCOBJ 61
: Object for exception handler

PUT_MT 62
: Store method table of object.

PUT_SIZE 63
Store size of object.

ALPHA_LITERAL 64
Immediate constant (0…255) Value stored in auxiliary field.

ALPHA_IZERO 65
Integer register r31
ALPHA_FZERO 66
Floating-point register f31

ALPHA_LDA 67
Load 16-bit constant.
ALPHA_LDAH 68
Load 16-bit constant high. (Use in combination with LDA to load 32-bit constant.)
ALPHA_SRA 69
Shift right arithmetic
ALPHA_SLL 70
Shift left logical
ALPHA_SRL 71
Shift right logical

ALPHA_S4SUB 72
Scaled by 4 subtract
ALPHA_S4ADD 73
Scaled by 4 add
ALPHA_S8SUB 74
Scaled by 8 subtract
ALPHA_S8ADD 75
Scaled by 4 add

ALPHA_BITCOMP 76
Bit-wise negation

ALPHA_BEQ 77
Branch if register equal to zero
ALPHA_BGE 78
ALPHA_BGT 79
ALPHA_BLE 80
ALPHA_BLT 81
ALPHA_BNE 82
ALPHA_BLBS 83
Branch if register low bit is set

ALPHA_BLBC 84
Branch if register low bit is clear

ALPHA_CMPEQ 85
Compare signed equal
ALPHA_CMPLE 86
ALPHA_CMPLT 87
Compare signed less than
ALPHA_CMPULT 88
Compare unsigned less than
ALPHA_CMPULE 89

ALPHA_CMOVEQ 90
ALPHA_CMOVGE 91
ALPHA_CMOVGT 92
ALPHA_CMOVLE 93
ALPHA_CMOVLT 94
ALPHA_CMOVNE 95
Conditional move if register not equal to zero

ALPHA_ZAPNOT 96
Zero out Ra bytes not specified in Rb

ALPHA_SEXTB 97
Sign extend byte
ALPHA_SEXTW 98
Sign extend word
ALPHA_EXTQH 99
Extract quadword high
ALPHA_EXTBL 100
Extract byte low
ALPHA_EXTWL 101
Extract word low
ALPHA_INSBL 102
Insert byte low
ALPHA_INSWL 103
Insert byte low
ALPHA_MSKBL 104
Mask byte low
ALPHA_MSKWL 105
Mask word low
ALPHA_SEXTL 106
Sign extend longword.

ALPHA_STB 107
Store byte
ALPHA_STW 108
Store word
ALPHA_STL 109
Store longword
ALPHA_STQ 110
Store quadword
ALPHA_STS 111
Store longword floating
ALPHA_STT 112
Store quadword floating
ALPHA_STQ_U 113
Store unaligned quadword

ALPHA_LDBU 114
Load unaligned byte
ALPHA_LDWU 115
Load unaligned word
ALPHA_LDL 116
Load longword
ALPHA_LDQ 117
Load quadword
ALPHA_LDS 118
Load longword floating
ALPHA_LDT 119
Load quadword floating
ALPHA_LDQ_U 120
Load unaligned quadword

ALPHA_ITOFT 121
Convert integer to floating
ALPHA_CVTQS 122
Convert quadword integer to longword floating
ALPHA_CVTQT 123
Convert quadword integer to quadword floating
ALPHA_CVTTSSU 124
Convert quadword float to longword floating
ALPHA_CVTSTS 125
Convert longword float to quadword floating

ALPHA_CVT_STORE 126
Store to stack for conversion
ALPHA_CVT_LOAD 127
Load from stack for conversion
ALPHA_CVT_STS 128
Store to stack for conversion
ALPHA_CVT_LDL 129
Load from stack for conversion

ALPHA_LOAD_MT 130
Load method table pointer
ALPHA_LOAD_VTABLE 131
Load proc pointer from vtable
ALPHA_LOAD_ITABLE 132
Load proc pointer from itable

ALPHA_LOAD_IID_LOW 133
Load low bits of interface method id
ALPHA_LOAD_IID_HIGH 134
Load high bits of interface method id

ALPHA_GLOBAL_HIGH 135
Operation for getting the high 16 bit displacement in preparation for accessing global data
ALPHA_GLOBAL_LOW4 136
ALPHA_GLOBAL_LOW8 137
Operation for loading a global constant. Takes as input the corresponding GLOBAL_HIGH value.

ALPHA_PUT_STATIC4 138
ALPHA_PUT_STATIC8 139
ALPHA_GET_STATIC4 140
ALPHA_GET_STATIC8 141
Operations for loading/storing global variables. Takes as input the corresponding GLOBAL_HIGH value, and have an FREF as its auxiliary field.

ALPHA_FLOAD1 142
ALPHA_FLOAD2 143
ALPHA_FLOAD4 144
ALPHA_FLOAD8 145
Simple field load

ALPHA_FSTORE1 146
ALPHA_FSTORE2 147
ALPHA_FSTORE4 148
ALPHA_FSTORE8 149
Simple field store

ALPHA_FLOAD_BASE 150
Load from rounddown(offset, 4)
ALPHA_FSTORE_BASE 151
Store to rounddown(offset, 4)
ALPHA_FIELD_SLL 152
Shift left for sign-extension
ALPHA_FIELD_EXTBL 153
ALPHA_FIELD_EXTWL 154
Extract field bits into right place
ALPHA_FIELD_INSBL 155
ALPHA_FIELD_INSWL 156
Insert field bits into right place
ALPHA_FIELD_MSKBL 157
ALPHA_FIELD_MSKWL 158
Mask out all except field bits

ALPHA_LOAD_STKARRAY 159
Load pointer to temp array on stack

ALPHA_BOUNDS_TRUE 162
Raise bounds if input is true

ALPHA_TRAPB 163
Trap instruction for catching exceptions

ALPHA_JMP 164
Indirect jump

ALPHA_BRNEXT 165
Branch to next instruction: note that this instruction is not really a control instruction. It is just used to capture the address of the next instruction.

ALPHA_LD_JT 166
Load entry from jump table

# Appendix C: JET Calling Standard[2]

## Register Usage Conventions

### Integer Registers

| Register | Description |
|---|---|
| $0 | Function value register. In a standard call that returns a non-floating-point function result in a register, the result must be returned in this register. In a standard call, this register can be modified by the called procedure without being saved and restored. |
| $1 - $8 | Conventional scratch registers. In a standard call, these registers can be modified by the called procedure without being saved and restored. |
| $9 - $14 | Conventional saved registers. If a standard-conforming procedure modifies one of these registers, it must save and restore it. |
| $15 | Stack frame base (FP) register. For procedures with a run-time variable amount of stack, this register is used to point at the base of the stack frame (fixed part of the stack). For all other procedures, this register has no special significance. If a standard-conforming procedure modifies this register, it must save and restore it. |
| $16 - $21 | Argument registers. In a standard call, up to six non-floating-point items of the argument list are passed in these registers. In a standard call, these registers can be modified by the called procedure without being saved and restored. Additional arguments must be passed through the stack. |
| $22 - $25 | Conventional scratch registers. In a standard call, these registers can be modified by the called procedure without being saved and restored. |
| $26 | Return address (RA) register. In a standard call, the return address must be passed and returned in this register. |
| $27 | Procedure value (PV) register. In a standard call, the procedure value of the procedure being called is passed in this register. In a standard call, this register can be modified by the called procedure without being saved and restored. |
| $28 | Volatile scratch register. The contents of this register are always unpredictable after any external transfer of control to or from a procedure. This unpredictable nature applies to both standard and nonstandard calls. This register can be used by the operating system for external call fixing, auto loading, and exit sequences. |
| $29 | Global pointer (GP) register. For a standard-conforming procedure, this register must contain the calling procedure's global offset table (GOT) segment pointer value at the time of a call and must contain the calling procedure's GOT segment pointer value or the called procedure's GOT segment pointer value upon return. This register must be treated as scratch by the calling procedure. |

[2] JET calling standard is the same as the Alpha calling standard

| | |
|---|---|
| $30 | Stack pointer (SP) register. This register contains a pointer to the top of the current operating stack. Aspects of its usage and alignment are defined by the hardware architecture. |
| $31 | ReadAsZero/Sink register. This register is defined to be binary zero as a source operand or sink (no effect) as a result operand. |

**Floating-Point Registers**

| Register | Description |
|---|---|
| $f0 | Floating-point function value register. In a standard call that returns a floating-point result in a register, this register is used to return the real part of the result. In a standard call, this register can be modified by the called procedure without being saved and restored. |
| $f1 | Floating-point function value register. In a standard call that returns a complex floating-point result in registers, this register is used to return the imaginary part of the result. In a standard call, this register can be modified by the called procedure without being saved and restored. |
| $f2 - $f9 | Conventional saved registers. If a standard-conforming procedure modifies one of these registers, it must save and restore it. |
| $f10 - $f15 | Conventional scratch registers. In a standard call, these registers can be modified by the called procedure without being saved and restored. |
| $f16 - $f21 | Argument registers. In a standard call, up to six floating-point arguments can be passed by value in these registers. In a standard call, these registers can be modified by the called procedure without being saved and restored. Additional arguments must be passed through the stack. |
| $f22 - $f30 | Conventional scratch registers. In a standard call, these registers can be modified by the called procedure without being saved and restored. |
| $f31 | ReadAsZero/Sink register. This register is defined to be binary zero as a source operand or sink (no effect) as a result operand. |

# Appendix D: Register Allocation

The register allocation algorithms used in this research are adapted from classic compiler register allocation algorithms and may be found in Morgan's *Building An Optimizing Compiler*. The two major algorithms that accomplish local register allocation are the FAT algorithm by Hendron and the One-Pass Allocation algorithm. Below is the pseudo-code that describes how these algorithms work:

```
struct LALLOC {
      startTime = 0
      endTime = 0
      numRegister = 0
}

procedure local_allocate(METHOD *m) {
      info = new LALLOC

      for each block b in m {
            build_local_conflict_graph(info, b)

            while set of values in livestart_set is not null {
                  t = a value from livestart_set
                  remove t from livestart_set

                  try_allocate_with_global(info, b, t)
            }

            one_pass_allocate(info, b)
      }
}
```

A generic local allocation algorithm would typically begin by classifying each value in the method and determine the various liveness sets. This calculation was done in Turbo and is not necessary in Afterburner. This algorithm allocates registers for values one block at a time. The first step is to build a local conflict graph that contains liveness ranges for each value node in the block. The live start set is the set of values that are live or in use at the beginning of the block. The values in the live set are global variables. Local allocation attempts to allocate local variables in the same register as these global variables if the liveness ranges of the local variables do not conflict with the liveness ranges of these global variables. Finally, a linear one-pass allocation algorithm is called on the block to allocate the remaining local variables.

Typically local allocation would also have to reduce the register pressure before the algorithm begins, but since Turbo maintains that the maximum register pressure is not larger than the number of register available, a register spilling and pressure reducing step is unnecessary. While this property states that the maximum register pressure is no greater than the number of registers available, this does not imply register spilling is

entirely avoidable. One-pass allocation will still encounter situations where it is too difficult or impossible to allocate all the variables and spill code will have to be emitted.

```
procedure build_local_conflict_graph (LALLOC *info, BLOCK *b) {
      timeCount = 0
      live_set = liveout_set
      endTime[] = int[number of values in method]
      startTime[] = int[number of values in method]
      pressure = 0

      for each value v in the method that contains this block {
            endTime[v] = timeCount
      }

      for each value v in block b in reverse order {
            timeCount++
            startTime[v] = timeCount
            remove v from live_set

            timeCount++
            for each input p to value v {
                  if p is not in live_set {
                        endTime[p] = timeCount
                        insert p into live_set
                  }
            }
            cnt = number of elements in live_set
            if (cnt > pressure) {
                  pressure = cnt
            }
      }
      timeCount++
      for each value v in live_set {
            startTime[v] = timeCount;
      }

      info->startTime = startTime
      info->endTime = endTime
      info->numRegister = pressure
}
```

Build_local_conflict_graph is responsible for figuring out the range over which each variable is live within a particular block. This is accomplished by determining the first and last times a variable is used. This range is the liveness range of the variable. This calculation begins by initializing all end times for each variable to zero. The algorithm then steps through each value in the block in reverse execution order. Time is measured as relative displacement from the end of the block. Each instruction contributes 2 units of time, one unit for write and one for read. The live set starts out as the set of values that are live at the end of the block. As the algorithm marches through the values, values are inserted into the set of live values if they are read and are not already in the live set. If values are written to then the values are removed from the live set. Start time is set to be the first time that the value is written, and end time is the last time that the value is read.

Some values are live at the beginning of the block and have start time set equal to the beginning of the block.

```
procedure try_allocate_with_global (LALLOC *info, BLOCK *b, VALUE *g) {
      /* this is the FAT algorithm */
      beginTime = info->endTime[g]
      finishTime = 0
      for each value v in block b in reverse order {
            if v has the same type and color as g {
                  finishTime = info->startTime[v]
            }
      }

      for each value v in block b in reverse order {
            if v has the same type as g and v has not been colored {
                  /* if liveness of g and v do not overlap */
                  if (info->endTime[v] >= finishTime and
                      info->startTime[v] < beginTime) {
                        v->location = g->location
                        finishTime = info->startTime[v]
                  }
            }
      }
}
```

Local variables can be allocated with global variables if their liveness ranges do not overlap. This algorithm is called once for every global variable. During its execution, it calculates the liveness range for the register used by the global variable by concatenating liveness ranges of all variables that share the same register. Next, it assigns local variables to the same register as the global variable if the liveness ranges of the local variable do not overlap the liveness range of the register.

```
procedure one_pass_allocate (LALLOC *info, BLOCK *b) {
      /* Initialize the free register sets */
      free_regs = set of available registers

      /* Initialize the global register sets */
      for each value v in livestart and livethru sets for block b {
            insert v->location into global_regs
      }

      /* delete liveEnd from freereg set and copy into live set */
      live_set = new set
      for each value v in liveend set {
            remove v->location from free_regs
            insert v into live_set
      }

      free_regs = free_regs - global_regs

      f_set = new set
      for each value v in block b in reverse execution order {
            remove v from live_set
```

```
            if (v requires a location and
                v->location is not on stack and
                v->location is not in global_regs) {
                  insert v->location into free_regs
            }

            for each parameter p of value v {
                  if (p requires a location and
                      p is not in live_set) {
                        insert p into live_set

                        if (p->location is symbolic and
                            p->location is not on stack) {
                              f_set = free_regs - v->unavailRegs
                              if f_set is empty {
                                    set p->spill SPILL_OUT flag
                                    p->location.type = STACK_TMP
                                    p->location = new stack_tmp slot
                              } else {
                                    s = a register from f_set
                                    remove s from free_regs
                                    p->location = s
                              }
                        }
                  }
            } /* end for each parameter */
      } /* end for each value */
}
```

One-pass allocation is a linear algorithm that allocates all the remaining local variables. The algorithm determines the set of free registers by subtracting the registers occupied by global variables from the set of available registers. The algorithm then iterates over each value in the current block in reverse execution order. Each time a read occurs, the value is inserted into the live set; each time a write occurs the value is removed from the live set. When a value is removed from the live set, its registers are put back into the free register set. When a value is added to the live set, a register is chosen from the intersection of the set of free registers and the set of registers that are available to that variable. This is because some variables can only be assigned to certain registers, while other variables are less restrictive. Once one-pass allocation completes, all variables should have been allocated.

# Appendix E: Constant Propagation

The constant propagation algorithm employed is an adaptation of the classic compiler constant propagation algorithm by Wegman and Zadeck. These algorithms may be found in Morgan's *Building An Optimizing Compiler*. The following is pseudo code and detailed descriptions of the algorithm:

```
procedure constant_propagation (METHOD *m, JSP_TYPE *jsp_type) {
      rtconsts = set of arguments that are constants for jsp_type

      worklist = new FIFO queue
      blocklist = new FIFO queue

      /* initializations */
      add entry block of m to blocklist
      for each value v in m {
            set v->state.status = TOP
            if v is a phi node {
                  v->state.count = 1;
            } else {
                  v->state.count = v->paramCount + 1;
            }
      }
      for each argument value v in rtconsts {
            v->state.value = actual value of corresponding argument
      }

      while (worklist is not empty or blocklist is not empty) {
            while blocklist is not empty {
                  b = remove next block from blocklist
                  if b is marked reachable {
                        skip
                  }
                  mark b as reachable
                  for each value v in b {
                        if (--v->state.count <= 0) {
                              /* all inputs are initialized */
                              add v to worklist
                        }
                  }
                  if b is not a conditional block {
                        add each successor block of b to block list
                  }
            }
            while worklist is not empty {
                  v = remove next value from worklist
                  oldstatus = v->state.status
                  if (oldstatus == BOT) {
                        skip
                  }

                  eval_state(v)
```

```
                if v->state.status != oldstatus {
                      if v is a conditional branch {
                            if v->state.status = CONST {
                                  determine which branch’s successor
                                     block will be executed and add
                                     that block to blocklist
                            } else { /* status = BOT */
                                  add all successor blocks to
                                     blocklist
                            }
                      } /* end if v is conditional branch */
                      for each value u that uses v {
                            if (u->state.count <= 0) {
                                  add u to worklist
                            }
                      }
                } /* end if new status != old status */
          }
    }

    /* update m to use new constants and remove excess code */
    update(m)
}
```

The main concept in this constant propagation algorithm is that each value node can be in any of 3 status states, TOP, CONST, or BOT. Status states can only transition from TOP towards BOT. Specifically, once a status reaches BOT it can never transition to CONST or TOP. TOP indicates a node is un-initialized or not yet executed. CONST indicates a node is assumed to have constant value. BOT signifies a node has variable value and cannot be transformed into a constant. The constant propagation algorithm uses a block FIFO queue and a value FIFO queue. The algorithm begins with only entry block in the block queue. The algorithm continues until both the block and the value queues are empty. If a block being examined is not a conditional block, its successor blocks are placed on the block queue. Blocks that are conditional must wait until its corresponding branch instruction has been examined before the algorithm can determine which successor block to add. Values whose inputs have all been examined at least once are potential candidates for evaluation and are added to the value queue. When a value is examined, its state is evaluated. If the value’s new state differs from its old state, then users of this value need to be updated and are added to the value queue. If the value that was evaluated is a conditional branch then its appropriate target blocks need to be added to the block queue. Once all reachable blocks and values are examined and no further updates are necessary, the algorithm completes.

```
procedure eval_state(VALUE *v) {
      switch (v) {
      case v is input argument:
            if v is an argument marked as constant by the jsp_type {
                  v->state.status = CONST
                  v->state.value = argument value
            else {
                  v->state.statis = BOT
            }
```

```
        case v is a constant:
                    v->state.status = CONST
        case v is a phi node:
              if (all inputs to v whose state.status != TOP have the same
                  state.value) {
                    v->state.status = CONST
                    v->state.value = state.value of v's inputs
              else {
                    v->state.statis = BOT
              }
        }
        case v is a node that can be propagated:
              if any param of v has state.status = BOT {
                          v->state.statis = BOT
              } else if any param of v has state.status = TOP {
                          v->state.statis = TOP
              } else {
                    v->state.status = CONST
                    v->state.value = value of operation given constant
                                        params
              }
        default:
              v->state.statis = BOT
}
```

This algorithm determines the state for a give value node. A node's state is determined by examining the inputs to that value. There are roughly three categories of nodes, constant, phi, and normal. If a node is a constant node or an input argument that is classified as a constant by the specialization category, its state is set to CONST and its value set accordingly. A phi node is a value node with multiple inputs. The output value of a phi node equals the value of one its input. The selection of which input differs depending on the flow of execution taken to reach the phi node. This is used typically when a variable is set to different values in different branches. A phi node can be marked as a constant if all of the input nodes that are initialized have the same value. This approach is known as optimistic because it assumes that phi nodes will usually be constants and corrects the assumption once the algorithm determines that the phi node is actually not a constant. This optimistic approach allow more values to potentially become constants. Normal (non-phi and non-constant) nodes are marked as BOT if any of that node's inputs are BOT. Nodes are marked as TOP if any of that node's inputs are TOP. If all inputs to a node are CONST, then that node can be evaluated and marked as a constant. Certain nodes have operations that cannot be transformed into a constant and are marked as BOT.

```
procedure update(METHOD *m) {
        const_set = set of values in m whose state.status = CONST
        needed_set = set of values in const_set that are used by values
                          whose state.status = BOT
        remove_set = (const_set - needed_set) + set of values in m whose
                                                    state.status = TOP

        delete from m all values in remove_set
        modify all values in needed_set so that the node is an
               appropriate constant
```

```
      remove from m all blocks that are not marked as reachable
}
```

The update phase of the constant propagation algorithm makes use of all the information gained during constant propagation. Nodes that are not initialized are not executed and can be removed from the JET IR. Blocks that are not reached are also not executed and can also be removed from the JET IR. Constant nodes need to remain only if the constant node is used as input to a non-constant node. All other constant nodes can be removed. Constant nodes that are used by non-constant nodes need to be modified into a JET IR constant. The output of the process is a specialized and reduced JET IR that will be passed to the generator. The generator will use the specialized JET IR to generated specialized code.

# Appendix F: Sample Generation and Specialization

Java Source

```
public static int Test(int a, int b) {
    int r = a+b;
    int x = 5;
    int y = 9;
    int z = 234;

    if (a > b) {
      r += a;
      r *= y;
      r %= x;
    }
    else {
      r %= x;
      r += a;
      r /= x;
      r += z;
    }
    return r;
  }
```

Non-Specialized Generation

```
0x20021b40 Test [exp 1.60, cum 3.08]
      0x20021b40 lda    sp, -16(sp)
      0x20021b44 stq    ra, 0(sp)
      0x20021b48 bis    a0, a0, t1
      0x20021b4c addl   a0, a1, a0             ; r = a+b
      0x20021b50 lda    t2, 5(zero)            ; x = 5
      0x20021b54 cmple  t1, a1, v0
      0x20021b58 bne    v0, 0x20021b60
      0x20021b5c br     zero, 0x20021b7c
      0x20021b60 bis    t2, t2, a1             ; else {
      0x20021b64 bsr    ra, 0x203f5cd4         ;     r %= x;
      0x20021b68 addl   v0, t1, a0             ;     r += a
      0x20021b6c bis    t2, t2, a1
      0x20021b70 bsr    ra, 0x203f5d4c         ;     r /= x
      0x20021b74 addl   v0, 0xea, v0           ;     r += z
      0x20021b78 br     zero, 0x20021b8c       ; } if (a > b) {
      0x20021b7c addl   a0, t1, v0             ;     r += a
      0x20021b80 s8addl v0, v0, a0             ;     r *= y
      0x20021b84 bis    t2, t2, a1
      0x20021b88 bsr    ra, 0x203f5cd4         ;     r %= x
      0x20021b8c ldq    ra, 0(sp)
      0x20021b90 lda    sp, 16(sp)
      0x20021b94 ret    zero, (ra), 1
```

<u>Specialization</u>

```
Test(a=76, b=77) {
    int r = a+b = 76+77 = 153;
    int x = 5;
    int z = 234;

    if ((a > b) = false) {
    }
    else {
      r %= x;
      r += 76;
      r /= x;
      r += 234;
    }
    return r;
  }
0x20021b50 Test [exp 0.87, cum 3.04]
      0x20021b50 lda    sp, -16(sp)
      0x20021b54 stq    ra, 0(sp)
      0x20021b58 lda    a0, 153(zero)          ; r = 153
      0x20021b5c lda    a1, 5(zero)            ; x = 5
      0x20021b60 bsr    ra, 0x203f5cd4         ; r %= x
      0x20021b64 addl   v0, 0x4c, a0           ; r += 76
      0x20021b68 lda    a1, 5(zero)
      0x20021b6c bsr    ra, 0x203f5d4c         ; r /= x
      0x20021b70 addl   v0, 0xea, v0           ; r += 234
      0x20021b74 ldq    ra, 0(sp)
      0x20021b78 lda    sp, 16(sp)
      0x20021b7c ret    zero, (ra), 1


Test(a=77, b=76) {
    int r = a+b = 77+76 = 153;
    int x = 5;

    if ((a > b) = true) {
      r = 230;  /* r += 77; */
      r = 2070; /* r *= 9;  */
      r %= x;
    }
    return r;
  }
0x20022e80 Test [exp 0.58, cum 3.07]
      0x20022e80 lda    sp, -16(sp)
      0x20022e84 stq    ra, 0(sp)
      0x20022e88 lda    a0, 2070(zero)         ; r = 2070
      0x20022e8c lda    a1, 5(zero)            ; x = 5
      0x20022e90 bsr    ra, 0x203f5cd4         ; r %= x
      0x20022e94 ldq    ra, 0(sp)
      0x20022e98 lda    sp, 16(sp)
      0x20022e9c ret    zero, (ra), 1
```

# Appendix G: Performance Data

Classfile Size Comparison

| File | Java (KB) | JET (KB) | JET / Java |
|---|---|---|---|
| cmark3.0/AboutDialog.class | 1231 | 23155 | 18.81 |
| cmark3.0/BenchmarkAtom.class | 331 | 695 | 2.10 |
| cmark3.0/BenchmarkMonitor.class | 174 | 250 | 1.44 |
| cmark3.0/BenchmarkUnit.class | 1322 | 52677 | 39.85 |
| cmark3.0/CaffeineMarkApp.class | 506 | 7425 | 14.67 |
| cmark3.0/CaffeineMarkApplet.class | 914 | 22373 | 24.48 |
| cmark3.0/CaffeineMarkBenchmark.class | 3382 | 208553 | 61.67 |
| cmark3.0/CaffeineMarkEmbeddedApp.class | 1049 | 43554 | 41.52 |
| cmark3.0/CaffeineMarkEmbeddedBenchmark.class | 2858 | 156695 | 54.83 |
| cmark3.0/CaffeineMarkFrame.class | 6406 | 413572 | 64.56 |
| cmark3.0/DialogAtom.class | 1641 | 61985 | 37.77 |
| cmark3.0/FloatAtom.class | 1184 | 60434 | 51.04 |
| cmark3.0/GraphicsAtom.class | 2179 | 118903 | 54.57 |
| cmark3.0/ImageAtom.class | 2811 | 101285 | 36.03 |
| cmark3.0/LogicAtom.class | 1261 | 19639 | 15.57 |
| cmark3.0/LoopAtom.class | 940 | 26685 | 28.39 |
| cmark3.0/MethodAtom.class | 934 | 26775 | 28.67 |
| cmark3.0/SieveAtom.class | 817 | 24620 | 30.13 |
| cmark3.0/StopWatch.class | 638 | 19942 | 31.26 |
| cmark3.0/StringAtom.class | 1121 | 27819 | 24.82 |
| cmark3.0/TestDialog.class | 1159 | 31483 | 27.16 |
| cmark3.0/TestWindow.class | 467 | 7234 | 15.49 |
| **Average** | | | **32.04** |

Caffeine Mark Performance Ratios

| Overall Ratios | Java | JIT | Java on JET | JET w/o SP | JET w/ SP |
|---|---|---|---|---|---|
| **Java** | 1 | 3.085985 | 1.046884 | 5.514781 | 5.517856 |
| **JIT** | 0.324046 | 1 | 0.339238 | 1.787041 | 1.788037 |
| **Java on JET** | 0.955216 | 2.947782 | 1 | 5.267806 | 5.270743 |
| **JETw/o SP** | 0.181331 | 0.559584 | 0.189832 | 1 | 1.000558 |
| **JET w/ SP** | 0.18123 | 0.559272 | 0.189727 | 0.999443 | 1 |

Caffeine Mark Performance Data

These benchmarks were run on a 4 SMP EV-56 Alpha machine.

| | Sieve | Loop | Logic | String | Float | Method | Overall |
|---|---|---|---|---|---|---|---|
| Java | 1854 | 4662 | 11368 | 3182 | 3107 | 1105 | 3199 |
| Java | 1854 | 4662 | 11357 | 3202 | 2943 | 1110 | 3176 |
| Java | 1854 | 4660 | 11369 | 3198 | 3102 | 1109 | 3203 |
| Java | 1854 | 4661 | 11361 | 3196 | 3063 | 1110 | 3196 |
| Java | 1854 | 4662 | 11334 | 3222 | 2813 | 1121 | 3159 |
| **Average** | **1854** | **4661.4** | **11357.8** | **3200** | **3005.6** | **1111** | **3186.6** |

| | Sieve | Loop | Logic | String | Float | Method | Overall |
|---|---|---|---|---|---|---|---|
| JIT | 5146 | 10170 | 33745 | 10730 | 5306 | 8894 | 9815 |
| JIT | 5146 | 10166 | 33733 | 10542 | 5307 | 8915 | 9789 |
| JIT | 5147 | 10164 | 33904 | 10815 | 5307 | 8917 | 9839 |
| JIT | 5143 | 10160 | 33938 | 10972 | 5304 | 8908 | 9860 |
| JIT | 5148 | 10166 | 33980 | 10957 | 5307 | 8920 | 9866 |
| **Average** | **5146** | **10165.2** | **33860** | **10803.2** | **5306.2** | **8910.8** | **9833.8** |

| | Sieve | Loop | Logic | String | Float | Method | Overall |
|---|---|---|---|---|---|---|---|
| Java on JET classfiles | 2434 | 4663 | 11372 | 3236 | 2950 | 1081 | 3317 |
| Java on JET classfiles | 2434 | 4663 | 11361 | 3256 | 3097 | 1084 | 3348 |
| Java on JET classfiles | 2434 | 4461 | 11396 | 3260 | 2964 | 1084 | 3326 |
| Java on JET classfiles | 2434 | 4661 | 11364 | 3265 | 3099 | 1076 | 3346 |
| Java on JET classfiles | 2434 | 4662 | 11349 | 3237 | 3102 | 1081 | 3343 |
| **Average** | **2434** | **4622** | **11368.4** | **3250.8** | **3042.4** | **1081.2** | **3336** |

| | Sieve | Loop | Logic | String | Float | Method | Overall |
|---|---|---|---|---|---|---|---|
| JET without Specialization | 6211 | 26835 | 140178 | 10525 | 10836 | 11146 | 17597 |
| JET without Specialization | 6229 | 26821 | 140145 | 10524 | 10751 | 11142 | 17579 |
| JET without Specialization | 6233 | 26830 | 140189 | 10516 | 10796 | 11146 | 17594 |
| JET without Specialization | 6220 | 26814 | 140100 | 10209 | 10824 | 11138 | 17503 |
| JET without Specialization | 6229 | 26810 | 140109 | 10589 | 10741 | 11146 | 17594 |
| **Average** | **6224.4** | **26822** | **140144.2** | **10472.6** | **10789.6** | **11143.6** | **17573.4** |

| | Sieve | Loop | Logic | String | Float | Method | Overall |
|---|---|---|---|---|---|---|---|
| JET with Specialization | 6238 | 26826 | 140192 | 10511 | 10738 | 11142 | 17578 |
| JET with Specialization | 6238 | 26817 | 140205 | 10577 | 10707 | 11150 | 17589 |
| JET with Specialization | 6228 | 26826 | 140170 | 10497 | 10821 | 11146 | 17592 |
| JET with Specialization | 6238 | 26833 | 140153 | 10490 | 10746 | 11146 | 17575 |
| JET with Specialization | 6235 | 26820 | 140184 | 10522 | 10746 | 11146 | 17582 |
| **Average** | **6235.4** | **26824.4** | **140180.8** | **10519.4** | **10751.6** | **11146** | **17583.2** |

# References

1. Arnold, Ken and James Gosling. *The Java Programming Language*. 2nd ed. Reading, Massachusetts: Addison-Wesley, 1998.

2. Cooper, Keith D., Mary W. Hall, and Ken Kennedy. “Procedure Cloning.” *Proceedings of 1992 IEEE International Conference on Computer Languages* (1992): 96-105.

3. Consel, C., L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. “Tempo: Specializing Systems Applications and Beyond.” *ACM Computing Surveys, Symposium on Partial Evaluation* (1998).

4. Fitzgerald, Robert, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. *Marmot: An Optimizing Compiler for Java*. Technical Report MSR-TR-99-33, Microsoft Research, June 1999.

5. Gosling, James, Bill Joy, and Guy Steele. *The Java Language Specification*. Reading, Massachusetts: Addison-Wesley, 1996.

6. Grant, Brian, Markus Mock, Matthai Philipose, Craig Chambers, and Susan Eggers. “DyC: An Expressive Annotation-Directed Run-Time Specializations in C.” To appear in *Theoretical Computer Science*.

7. Hendron, L.J., G. R. Gao, E. Altman, and C. Mukerji. *Register Allocation Using Cyclic Interval Graphs: A New Approach to an Old Problem.* (Technical Report.) McGill University, 1993.

8. Keppel, David, Susan J. Eggers, and Robert R. Henry. “A Case for Runtime Code Generation.” Technical Report 91-11-04. University of Washington, 1991.

9. Lindholm, Tim and Frank Yelling. *The Java Virtual Machine Specification*. Reading, Massachusetts: Addison-Wesley, 1997.

10. Morgan, Robert. *Building an Optimizing Compiler*. Boston: Digital Press, 1998.

11. Muller, Gilles, Bárbara Moura, Fabrice Bellard, and Charles Consel. "Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code." *Proceedings of the Third Conference on Object-Oriented Technologies and Systems* (1997).

12. Poletto, Massimiliano, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. "`C and tcc: A Language and Compiler for Dynamic Code Generation." *ACM Trans. Prog. Lang. Sys*. 21(2): 324-369 (1999).

13. Proebsting, Todd, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. "Toba: Java For Applications: A Way Ahead of Time (WAT) Compiler." *Proceedings of the Third Conference on Object-Oriented Technologies and Systems* (1997).

14. Shaylor, Nik. *JCC – A Java to C Converter*. 8 May 1997 <http://www.geocities.com/CapeCanaveral/Hangar/4040/jcc.html>.

15. Stoltz, Eric, Michael Wolfe, and Michael A. Gerlek. "Demand-Driven Constant Propagation." Technical Report 93-023. Oregon Graduate Institute of Science and Technology, 1994.

16. Sun Microsystems. *Java 2 SDK Documentation*. <http://java.sun.com/products/jdk/1.2/docs>

17. Sun Microsystems. *Java HotSpot Performance Engine*. <http://www.javasoft.com/products/hotspot/index.html>.

18. Tower Technologies. *A High Performance Deployment Solution for Java Server Applications (Native Java Compiler and Runtime Environment)*. <http://www.towerj.com>.

19. Wegman, M. N., and F. K. Zadeck. "Constant Propagation with Conditional Branches." Conference *Proceedings of Principals of Programming Languages XII*, 1985. 291-299.