Department of Computer Science

Technical Report

Undergraduate Dissertation: Optimising the Java virtual machine instruction set

Alan Shaw

Technical Report 2006-11 **June 2006**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Optimising the Java virtual machine instruction set

Alan Shaw
BSc(Hons) Computer Science

May 4, 2006

Optimising the Java virtual machine instruction set
Submitted by Alan Shaw

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed.....

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed.....

Abstract

The implementation of statically and dynamically optimised virtual machines for resource constrained devices does not appear easy without trade offs between functionality, performance and memory use. Java virtual machines that interpret Java bytecode appear to be the answer, but are inherently slow. Avenues have been explored that can help with this, all of which appear to be equally viable. Our attention is turned towards the actual driving force behind the machine (the instruction set) and shows that through the use of simple, well established methodology's, a more compact instruction set can be derived for the Java virtual machine through the analysis of static properties of application data. This is desirable for a number of reasons allowing us to optimise implementations of virtual machines without explicitly modifying the way in which a Java virtual machine works. This dissertation describes the implementation of a system for extracting extensive statistics from a number of Java applications, and makes use of the statistics to describe a number of possible amendments to the existing Java virtual machine instruction set.

Acknowledgements

Prof. John Fitch and Dr. Julian Padget for their helpful advice and reassurance. My family, and my girlfriend Lizzy Mallett.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Java virtual machine (JVM) is the driving force behind all Java programs. It is the abstract representation of a computing machine that has its own instruction set and manipulates areas of memory at run time. The virtual machine as specified by Sun Microsystems (Lindholm & Yellin, 1999), currently has a number of implementations spanning a vast number of platforms allowing any Java application to be virtually platform independent.

The first implementation of the Java virtual machine was conceived by Sun and appeared on a handheld device that resembled a contemporary Personal Digital Assistant (PDA). The Java virtual machine however, is not biased towards any particular technology and was predominantly found in web browsers in previous years. Today, the bytecode that the Java virtual machine uses is not limited to just the Java language. Indeed there are pre-compilers which allow other languages to be used as development tools when specialist features like symbol manipulation (Lisp) or logic programming (Prolog) are needed. More recently, the Java virtual machine has been found in devices such as mobile phones, databases, smart cards, PDAs and other embedded systems. The scope for the use of Java applications is huge.

Considering the massive growth of the Java platform, it follows that the underlying technology should be as efficient and robust as possible. Hand held devices such as mobile phones require the executing size of a program to be kept to a minimum. A larger executing size inherently means more memory usage. This not only adds weight and cost to the product but also means that more battery power is needed to power the RAM in order to keep data from decaying.

The problem is solvable if a method of reducing either the program or executing size can be found. Encapsulating more complex operations in a single instruction would be one such

method, however the main issue is when, where and how this could happen. A number of methods of optimisation could be extended or modified to apply to the Java virtual machine. This project will attempt to decipher the most relevant and efficient method, or methods, with a view to producing an implementation based on these foundation ideas.

Chapter 2

The problem of optimisation

2.1 Virtual machines

Before we begin our discussion of the problem at hand, let us define our subject. The word virtual implies existence in essence or effect though not in actual physical form. Therefore, a virtual machine is a machine that exists in the virtual world. It is not a physical machine, but a general term used to describe a software abstraction of an idealised hardware architecture. Virtual machines are typically interpreters used to transform a computer program from one language to another, this can be literally for the purpose of obtaining the result or it could be for other reasons such as optimisation.

> "The Java virtual machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time." (Lindholm & Yellin, 1999)

The recent realisation of cross platform compatibility pioneered by Sun's Java virtual machine has probably become the most famously recognised use for virtual machines. Sun's virtual machine specification (Lindholm & Yellin, 1999) provided a standard machine for Java programs to run on. Once a virtual machine had been implemented for each computing machine available, it was possible for programmers to compile almost any program with the knowledge that it would run on any machine architecture. This spawned Sun's claim "write once, run anywhere".

If two Java virtual machines are implemented on two dissimilar hardware architectures and both are specialised with the same optimisation methods, there is no guarantee that they

will both execute the same Java bytecode at the same speed. Neither is it likely that they will exhibit the same improvement in quality. Furthermore, there is no guarantee that optimisations made at any stage in compilation/interpretation to execution have resulted in code that cannot be optimised further.

2.2 Types and techniques

A number of optimisation techniques have been implemented since the dawn of the computer and they have indeed been at the heart of many a computer program. The question that arises is what do we mean by optimisation? Literally optimisation is the act of rendering optimal, in other words "the best" or as close as we can get it. In the past, optimisation has been viewed as an improvement in quality and has been explored in two very different ways.

1. *Speed* — The speed a program can run at is an obvious optimisation technique. The problem that arises is the measurement of this absolute speed. Is the speed improvement due to the hardware environment, or due to efficient code? For the purposes of this discussion, the speed of a program will be denoted by *dynamic size*. (Bennett, 1988) suggests that dynamic size is the amount of space occupied by a compiled program, weighted by the frequency in which each instruction is executed. At first it appears slightly misleading that speed of a program is bounded by size. It does however, make sense that reducing the frequency an instruction is executed decreases expensive memory to processor operations that are needed during a computation. The reduction in overhead allows for faster execution of programs. With reference to the Java virtual machine model, we see that there are ultimately two machines for which dynamic size needs to be considered. We would ideally like to feed the physical *and* virtual machine dynamically optimised code.

2. *Size* — The size of compiled code will be denoted by *static size*. Static size of code in modern day machines is not often an issue. However, the problem has re-emerged as the scope for Java platforms on many connected, embedded and mobile devices has grown phenomenally. Mobile devices tend to utilise cheap and physically small dynamic memory which imposes a constant drain on the battery. It follows that a small static program size results in smaller memory requirements and thus less battery use. This enables the battery to power the device for longer and allows manufacturers to reduce size and weight of their product.

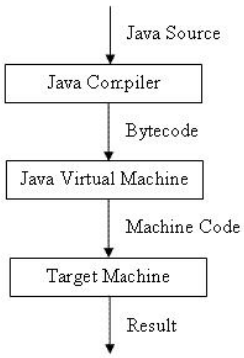

Figure 2.1: Data flow from program conception to execution

It is clear that in the case of a Java virtual machine, there are two opportunities for code optimisation. If we consider the processes from the point of code writing to machine execution we see that (loosely speaking) there are four steps, as illustrated in figure 2.1. It should be obvious that we have machine code at two stages; before and after the virtual machine — the questions that arise are what optimisations need to be considered at each stage, and how can optimisations at or before the second machine code generation preserve platform independence for the Java virtual machine?

In view of our two optimisation stages, it follows that optimisation can be performed on each one seperately.

1. *Bytecode optimisations* — bytecode is the result of compilation from a high level language that is fed into the Java virtual machine. These optimisations are concerned with the virtual machine itself, the instruction set and how it operates. It is important to remember that a virtual machine is a mapping from an abstract architecture to a physical architecture. There is scope for manipulation provided the abstract architecture can always be represented by the physical architecture.

2. *Native code optimisations* — these optimisations are concerned with the code that the virtual machine generates for a real, physical machine. Given that some virtual machine implementations compile code just-in-time, there is obvious scope for the optimisations that can be performed in order to generate efficient compiled code i.e. strength reduction, constant folding, etc. A purely interpreting virtual machine makes these kinds of optimisations difficult if not impossible.

2.3 JIT compilation

Analysis of the problem suggests that there are two main JVM types that have been explored. Probably the most popular type of JVM is the classic interpreter. The second is the dynamic just-in-time (JIT) compiler which as its name suggests, compiles bytecode immediately before execution of a program. In general, the Java virtual machine is run on resource-constrained devices as an interpreter. Less restricted devices such as a desktop computer can handle the large data footprint that compilation can sometimes generate. As such, Java virtual machines based on these devices can and do incorporate the JIT compilation technique.

The first JIT compilers typically compiled each method on first invocation and used a static set of optimisations each and every time. This set of optimisations would be simple and inexpensive, limiting their effective performance. These compilers had relatively little overhead but would result in a larger code base due to their simple "every method" strategy. Further development ensured JVMs contained much more complicated optimisation techniques. However this resulted in higher overheads (causing startup delays) and undesired base code growth. More recent JVMs have thus contained a multi level execution model in order to balance these unacceptable side effects. The two level model architectures (as seen in Sun's JDK 1.1) incorporated an interpretation engine in addition to a JIT compilation engine. They also enabled optimisation profiles so that overheads at application startup were reduced. Sun's Hotspot (Sun Microsystems Inc., 2002) is currently the most elegant JIT compilation JVM to date. Initially it runs as a standard interpreter. Concurrently, a run-time execution profile is created identifying code hot spots. Compilation of infrequently executed code is neglected, enabling the machine to compile and focus optimisation methods without necessarily increasing overhead.

The Hotspot virtual machine makes use of "adaptive optimisation" (as above) and method inlining. Both of which are the main concepts expressed in (Suganuma et al., 2005). Method inlining replaces method call sites with their target procedural bodies at every instance a method is invoked. The question arises when and which of these methods should be inlined? It is answered again by profiling, which asserts which areas should be optimised, de-optimised or re-optimised. This is then dynamically performed by the compiler — which helps to keep the footprint of the virtual machine from growing uncontrollably. The main use of method inlining is to reduce the dynamic frequency of method invocation. It also creates larger code blocks for the optimiser to work on, which increases the effectiveness of traditional optimisation techniques. Another optimsation suggested in (Suganuma et al.,

2005) which is not used in the Hotspot virtual machine is a type of non-standard interpretation called partial evaluation (or program specialisation), and is discussed in section 2.4.2. The method described uses profiling to create scenarios in which partial evaluation would be of significant value. These scenarios are then compiled in the hope that they will be utilised at some stage later in the execution process.

2.4 Non-standard interpretation

Standard interpretation is almost always performed with the view to computing the answer of a problem. Non-standard computation or "abstract interpretation" (Cousot & Cousot, 1977) involves interpretation for the purpose of obtaining properties of a program. This class of optimisations allows us to deduce properties about a program without performing the motions of actually computing the answer. A simple example to demonstrate this concept is the rule of signs. Here, we can deduce information about the sign of the result of a simple arithmetic operation without knowing the magnitude of the variables involved. The example; *(-)x * (+)y* implies a negative solution. Whereas *(-)x + (+)y* implies neither a negative or positive solution. We have two possible answers, or "no information". Clearly we can define a finite domain of abstraction to enable us to evaluate signs for all multiplications. Ideally we would create a finite abstract domain for each and all arithmetic operations. Although as can be seen from the later example, this is not always possible.

2.4.1 Pass separation

Staging transformations were identified by Jørring & Scherlis (1986) as a method of moving computation from a later stage to an earlier stage. This, in essence, is the basis of compilation. One such process is pass separation, which when given a program p, transforms it into two separate programs p_1 and p_2 via two phases known as stratification and separation; as shown in equation 2.1.

$$p(x, y) = p_2(p_1(x), y) \tag{2.1}$$

Ideally, x is a purely static part of the computation and y is dynamic. It is easy to see that computation manipulated such that it is executed less often (frequency reduction) or is shifted from run-time[1] to compile time (pre-computation) will allow us to make very

[1] run-time is used rather loosely in this sense — interpret time on the JVM equates to run-time from the point of view of the programmer

powerful optimisations. Both of these methods are identified by (Jørring & Scherlis, 1986) as the main two classes of optimisations that enable us to detect and shift computation between stages. These classes encompass a range of methods such as partial evaluation, dynamic programming, reduction in strength and data transformations.

A series of papers by Hannan (1994), demonstrated that the ideas put forward by Jørring & Scherlis (1986) could be applied to the automatic extraction of a compiler and executor from operational semantics. The target language generated by this compiler and executor represents a semantics-directed machine architecture — something not dissimilar to the idealised hardware architecture of a virtual machine. p_1 becomes a kind of weak compiler which translates the source language into an intermediate level language and p_2 provides the definition for this language.

Further work by Padget supported both Hannan and Jørring's work. An ANF transformer for a CEK machine was extracted and expressed as a dynamic process by staging. We notice that in this process, the source and target language are the same, and that stratification and separation is used to move the ANF transformation to an off-line stage. Not only has computation been saved at run-time, but Padget's research demonstrates how lazy a-normalisation improves the CEK machine (e.g. by saving on activation records for tail-recursive function calls), resulting in dynamic *and* static size savings.

2.4.2 Partial evaluation

The differences between pass separation and partial evaluation are subtle but nevertheless distinguishable. Partial evaluation is essentially a specialisation of a given program with respect to an input value x. For example, a function $f(x,\ y) = x^y$ can be specialised with respect to a value of $x = 5$. The specialised function becomes $f_5(y) = 5^y$. Program specialisation as a concept is not a new subject, and was first formulated by Kleene and the s-n-m theorem, which determined the computability of specialised programs. Formally, partial evaluation is expressed in equation 2.2.

$$I(x, y) = I_x(y) \tag{2.2}$$

Partial evaluation is the specialisation of an interpreter I in order to obtain a weakly compiled program I_x. The application of I_x to y becomes the execution stage for the program. The essence of partial evaluation is a triple serving to pre-compute, unfold and reduce programs when given a specialisation n.

> "The technique is to *precomute* all expressions involving n, to *unfold* the recursive calls to function f and to *reduce x*1* to x" (Jones et al., 1993)

The implementation of partial evaluation by Suganuma et al. (2005) showed limited value during the benchmarking tests that were performed (a 2%-3% performance improvement). The code growth was also unfavourable, yielding a 7%-30% increase. The paper, however, does argue that the implementation is flawed for a number of reasons — one of which is that code specialisation was only considered on a method by method basis and will only specialise a whole method. This results in a suggested technique called *method outlining* which specialises sections of code within a method. One encouraging outcome was that the hit ratio[2] of specialised code generated was quite high overall, suggesting that there is scope for the use of the technique given the correct implementation.

2.5 Standard interpretation

Our two domains have become resource-constrained devices and resource-rich devices for interpretation and JIT compilation respectively. JIT compilers provide the best performance although interpreters do offer favourable advantages, especially for resource-constrained devices. JIT compilers enlarge the JVM memory footprint. This is due to the extra code that makes up the JIT engine, the intermediate data structures used during compilation and the compiled code produced by the compiler. Research by Zhang & Krintz (2005) presents an analysis in size of bytecode and compiled native code on the ARM and IA32 machines with both the Jalapeño VM (Alpern et al., 2000) and Kaffe VM (Kaffe, 1998). The study shows that the IA32 native code is 6-8 times the size of the bytecode for both Jalapeño and Kaffe. ARM native code is even larger, reflecting the RISC based architecture: 16-25 times that of the bytecode equivalent. Standard interpretation based virtual machines do not possess this overhead and as such are much more applicable to the resource-constrained domain. There are other advantages to interpretation over compilation, such as ease of implementation and maintainability. Interpreters can be constructed in such a way that portability to other architectures is uncomplicated — facilitated by code re-use, an aspect which is respectively more difficult for the back end of a JIT compiler.

Naturally there has been research into closing the gap between memory use. The research by Zhang & Krintz (2005) (as above) was a feasibility study for adaptive code unloading as an alternative to not compiling code. The proposed code unloading framework decides

[2] i.e. the rate at which generated code was actually utilised

what code to unload and when unloading occurs. It enabled an on average reduction of code size of 36%-62%. Interestingly, unloaded code that was later invoked by the program (and as such needed to be re-compiled) still allowed a 23% execution-time improvement under highly resource-constrained conditions. Looking at this objectively, the framework trades off the memory management overhead with re-compilation overhead. This isn't an optimisation as such, but demonstrates that specialisation of the virtual machine[3] for target platforms is needed. There is also evidence that suggests compilation for resource-constrained devices will always be a juggling act between static size and dynamic size of both the virtual machine and the executing program. Considering this, and other advantages for interpreting virtual machines, it seems reasonable that Sun's JVMs for connected and embedded systems (CVM), smart cards (JavaCard) and mobile devices (KVM) are all based on interpreters.

The k virtual machine (KVM[4]) (Sun Microsystems Inc., 2005) is a virtual machine developed specifically for mobile devices. The KVM is the result of a "Spotless" (Bush et al., 1999) implementation of the Java virtual machine which essentially analyses the static size problem and attacks from a different angle. A lot of effort has been put into developing small statically sized code for programs that use the virtual machine. Conversely, typical Java VMs require a number of megabytes to run, mainly due to the standard classes of the JDK[5]. The "Spotless" implementation aimed to reduce the static size of the virtual machine without sacrificing functionality.

The main ideas employed in the design of the machine were not dissimilar to ideas put forward in the creation of RISC machines. Seven simple techniques were used when designing the system, of which most were focused on simple design. The KVM was constructed from the ground up — only what was absolutely necessary was added. The outcome was that the "Spotless" project did more than enough to achieve its goals. The text even states that "much of the standard JDK is simply not too big to fit on a memory limited platform" (Bush et al., 1999). We notice that the KVM does little more than to establish a smaller implementation of essentially the same computing machine. It is noteworthy that it takes in the order of 10 to 50 kilobytes of memory, but the price is paid in performance — where the standard Sun JDK 1.1[6] runs up to 70% faster. Indeed this is a good basis for a virtual machine but again we have hit the issue of trade offs in dynamic size for static size.

[3] i.e. the way in which it operates

[4] The "k" in KVM stands for kilo which equates to it's memory budget, which is measured in kilos.

[5] Bush et al. (1999) observed that in JDK 1.1 printing out one string of text required 20 - 30 classes to be loaded

[6] Without JIT compilation

2.5.1 Code threading

Predominantly switch dispatch has been the most popular interpretation method due to its simplicity. Switch dispatch consists of a giant switch statement where each case is a subset (consisting of a single opcode) of the entire instruction set. Work conducted by Ertl & Gregg (2001) showed instruction dispatch consumes a significant amount of running time (up to 62%). This is mainly due to the look up process that aids dispatch (fetch, decode, start). The look up process requires a table so operations can locate the address of the corresponding native subroutine. The idea of threaded code (Bell, 1973) is to encode operations as addresses of corresponding subroutines. Therefore no lookup table is needed and thus the dispatch overhead is eliminated.

The result of the work by Ertl & Gregg (2001) was an implementation of the threaded code concept for a number of interpreters, resulting in a factor of two speed up in some cases. Further work by Beatty et al. (2003) showed that the method was also possible for a Java virtual machine. The particular machine was Sun's CVM for connected and embedded systems. We notice however that code threading is associated in the most part with speed; a trait which is important but not always necessary. In our resource-constrained domain, we can see that the translation of bytecode to threaded bytecode at run-time would increase static size of the program. Given that the technique has been implemented within the CVM, Beatty et al. (2003) show no benchmarking for the footprint of the compiler (which originally is meant to run on devices with up to 2MB of memory). Hence, it remains to be seen whether the trade off in size for speed is worthwhile.

2.5.2 Run-time macro opcodes

An optimisation made possible by direct code threading is run-time macro opcodes. Here, bytecodes are encoded as pointers; thus more than 256 operations can be constructed. Run-time macro opcodes are concatenations of fragments of operations in much the same spirit as Sweet & James G. Sandman (1982). The two pass system[7] implemented by Piumarta & Riccardi (1998) means that macro creation happens during translation of bytecode to threaded bytecode. Unlike the BrouHaHa machine (Miranda, 1987), which detected and applied a static set of new macro opcodes, Piumarta & Riccardi (1998) implement an extension of this technique to enable dynamic analysis and macro creation — procuring a more specialised program. The technique results in a substantial reduction in instruction

[7]On the first pass, bytecode is translated to threaded bytecode. A second pass identifies and replaces (inline) bytecodes with macro opcodes.

dispatch. Space is saved during the second pass by a macro cache which stores dynamically generated macro codes, keyed on a hash value computed from the stream of unoptimised opcodes. A cache miss adds the new macro to the cache for subsequent use. In this way, the same macro code is not computed twice. The technique performed very well in benchmarking tests but similarly to the studies conducted on code threading, it does not give information about the overall run-time footprint of the machine.

2.5.3 Software pipelining

Our final standard interpretation optimisation concerns software pipelining[8]. The code compression system (Hoogerbrugge et al., 1999) shows an example of a pipelined interpreter. The observed issue with interpreters that do not consider processor pipelining is that their performance can suffer due to the amount of cycles needed to complete an instruction. Moving dispatch code for the next instruction into the current instruction enables more efficient processor usage by reducing the effect of branch delay. Clearly software pipelining is a valuable optimisation and facilitates implementation of a fast and efficient virtual machine.

2.6 Statistical optimisation

Clearly non-standard interpretation is applicable on-line and off-line, the techniques expressed by Suganuma et al. (2005) and Padget illustrate this respectively. Standard interpretation is interested mostly with on-line optimisations that either improve the speed or memory usage of the Java virtual machine. These optimisations are complex and need to be applied with precision and skill to procure noticeable improvements. We notice that both standard and non-standard interpretation optimisations ultimately modify existing technology by some possibly complex transformation. This section proposes that technical changes to the implementation or complex transformations to the code are not a necessity to producing a more optimal virtual machine. We hope to convey methods that allow optimisation of the Java virtual machine by observing properties of its instruction set.

2.6.1 RISC and CISC for virtual machines

Traditionally, two approaches have been used to bridge the semantic gap between high level languages and instruction sets. They reflect the language and machine respectively. A

[8]For the purposes of this discussion we will assume that the reader is familiar with pipelining as a concept.

RISC[9] based instruction set consists of a very small number of simple instructions that do very little. The principle behind this is that the simple instructions allow the machine to do work very fast. The small amount of instructions allow compiler writers to pick out an exact sequence of instructions easily and quickly. RISC instruction sets unfortunately create an issue with dynamic and static size. Programs for our virtual machine should not exhibit either of these traits. Indeed we also notice that a RISC instruction set for use with a virtual machine is not necessary. This is justified by the observation that a RISC instruction set nullifies many of the benefits that RISC instruction sets bring to real machines, namely speed and simplicity. Nullification occurs because of the fact that a virtual machine cannot process simpler instructions faster as it is bound by the speed of the machine it is running on. Consider a one to one mapping between RISC instructions for a virtual machine and RISC instructions for a real machine and that a function is expressed as a set of instructions in both machines. We can see that the one to one arrangement gains nothing but the time and work it takes to select a real machine instruction from the corresponding virtual machine instruction. If one virtual machine instruction could express more than one real machine instruction we can save time and effort interpreting a Java program. We also see that the benefits of using a virtual machine are compromised when using a RISC style instruction set — one of the major benefits for compiler writers is that all Java virtual machines use the same compiled code. This means that applications can be distributed in a compact binary form allowing them to be run easily and quickly. The key aspect here is the distribution in binary. If the compiled Java classes are represented using a RISC instruction set, surely it follows that they are not as compact as they could be? The simple assertion that compactness can be increased by using an instruction set which contains instructions that have more semantic content is reason enough to utilise a CISC[10] style instruction set with the Java virtual machine.

CISC machine instructions perform complex operations and are generalised (sometimes unsuccessfully) to represent the essential operations of high level languages. CISC instructions need to be selected carefully as it is possible redundant information may be created. This can be due to poor compilation but can also be because of an inexact semantic match.

2.6.1.1 The Warren Abstract Machine (WAM)

> WAM design principle 3: "Particular situations that occur very often, even though correctly handled by general-case instructions, are to be accommodated

[9] RISC — Reduced Instruction Set Computers.
[10] CISC — Complex Instruction Set Computers.

by special ones if space and/or time may be saved thanks to their specificity" (Aït-Kaci, 1991)

The Warren Abstract Machine is an example of a successful implementation of a CISC style instruction set for a virtual machine. WAM is the virtual machine developed for the execution of Prolog statements and has a complete instruction set consisting of just 39 instructions. Aït-Kaci (1991) is a complete account of how the current WAM came about through several intermediate abstract machine designs. We see here that the instruction set is evolved from analysis of the execution of Prolog producing a extremely compact and semantically rich instruction set.

Considering the compactness of the WAM instruction set, the use of CISC instructions with the Java virtual machine shows promise. A quick analysis of the Java virtual machine instruction set shows that it has a very RISC centred design and thus is a good candidate for conversion to a more CISC centred design. We do notice however that the Java virtual machine instruction set is already much bigger than the already specialised WAM instruction set. A methodical evolution of these Java instructions could take an excessive amount of time to perform. Section 2.6.2 demonstrates a slightly different method of analysing aspects of an virtual machine (or indeed any machine) for optimisation, which is possibly more applicable to the Java virtual machine. Instead of methodically analysing use of the language, a kind of brute force method is used to obtain statistical data on usage, giving an indication about where specialisation should occur and is most effective.

2.6.2 Static and dynamic profiling

Static and dynamic profiling is concerned with constructing an optimal CISC style instruction set based on a collection of statistics describing one or more program(s). Static profiling is concerned with collecting statistics on instructions from the compiled output of a program. Statistics from static profiling can facilitate code compaction (static size) because of the knowledge gained about the content of the application. Dynamic profiling is performed by capturing statistics relating to instructions as they are executed at run-time. As one might expect, dynamic profiling can help optimise execution speed (dynamic size) through both knowledge of program content and its execution paths. Either method produces an asymmetric instruction set, which is not necessarily easy to compile, but enables a compact representation.

The paper by Sweet & James G. Sandman (1982) is an example of static profiling of an existing instruction set. They describe a simple and effective experimental method along

with the statistics required to begin the process of suggesting newly specialised instructions. Their experimental plan was as follows:

1. Firstly, existing object code was normalised. This is a process of pulling apart instructions that had previously been specialised by some other process, in order to obtain a canonical instruction set. In total 2.5 million bytes were produced.
2. Secondly, statistics were collected and further refined to formalise their idea of which types of statistics would answer the most questions about language usage. These types became:
    - static opcode frequency
    - operand values
    - opcode successor
    - opcode predecessors
    - opcode pairs.
3. Thirdly, their pattern matcher was then used to uncover some interesting observations which allowed the creation of new and variable length opcodes.

After peephole optimisation, a total of 12% saving in space was procured by the new instructions. This may not seem like a huge saving, but is more than acceptable given the maturity of the language and the experienced computer architects that had previously worked on it. This saving is proof that machine based static profiling can produce optimisations for instruction sets despite years of previous manual optimisations.

The work by Bennett (1988) drew heavily on the formal processes described by Sweet & James G. Sandman (1982). In fact, Bennett's work went a step further to automate the process of instruction suggestion, and in the case of DL (a Design Language), even allowed users to specify how an instruction set was built from a set of statistics. Using the processes used for Mesa, Bennett's work derived a design methodology which formed the bases of his two main implementations — ISGEN and DL. This design methodology included the following design rules:

1. Create a new instruction with a smaller argument, where an argument is observed to fit into a smaller data type, a new instruction should be created with the aforementioned data type as the argument — the information here is derived from the second statistic type from the Mesa instruction set analysis, where operand values are considered.

2. Create a new instruction of a single argument with one value implied — this criteria is also derived from operand values statistic type, used in the Mesa instruction set analysis.

3. Create a new instruction by concatenating two existing instructions — the third, fourth and fifth statistic type from the Mesa instruction set analysis is used to derive this information.

In these design rules, we do not mention the first Mesa statistic type (static opcode frequency). It should be noted that Static opcode frequency tells us which instructions are most used and thus most worthy of specialisation. This is applicable to all rules as it would be wasteful to specialise an instruction or instructions that were rarely used.

The interesting aspect of the design methodology that Bennett suggests repeated application of the design rules to the instruction set. ISGEN is reported to achieve a predicted reduction in size to 74.87% of the original when the rules are applied twice. Bennett goes further to suggest that in this way a reduction to 29% is feasible. This, to the author seems like somewhat of an over estimate. However it is more than impressive to see an almost 30% predicted reduction in code size after just one application.

2.7 Conclusion

Given the variety of the methods discussed in the previous sections, it seems hard to draw a conclusion as to which method is most beneficial. Our conclusion was eventually drawn by once again considering our domain of choice (resource constrained devices) and matching our methods to our desired optimisation attributes. The following two examples, whilst not an exhaustive summary of everything discussed, do highlight some of the issues which seem to be inherent of a number of methods we have considered:

Suganuma et al. (2005) proposed a method primarily concerned with producing a high performance Java virtual machine by balancing the overhead caused by profiling methods, compilation and recompilation. We notice that Suganuma et al. (2005) make little or no reference to the memory footprint of the virtual machine in their implementation. Something we are primarily concerned about in our domain, where a large memory footprint is undesirable. Similarly, code threading can have desirable effects on performance, but is traded off by translation process needed to turn bytecode into threaded bytecode.

In fact it is easy to see that almost all the optimisations discussed are simply balancing acts between memory footprint and dynamic size. The trend is that dynamic size is being

reduced and memory consumption increased. In many of our considerations we have had to take a step back and consider memory consumption when it is not often mentioned. This memory consumption is acceptable on a lot of computer systems these days but is not acceptable for the limited memory resources found in mobile devices like telephones, PDAs and embedded systems.

The optimisation method(s) that appear to be most applicable to the Java virtual machine is that of static or dynamic profiling. Not only do these methods consider both static size and dynamic size, but they also appear to be the simplest, and yet most beneficial. In addition, using the Java virtual machine in combination with these methods seems fitting to say the least. The already large instruction set lends itself to a mechanised statistical solution, and would benefit greatly from the normalisation process described for the Mesa instruction set in section 2.6.2. We should also note the disadvantages of using an instruction set such as the RISC one curiously found in the Java virtual machine, and that the introduction of a more CISC centred instruction set could reduce the significant amount of running time that instruction dispatch consumes.

Chapter 3

Requirements

To begin project development, a set of requirements are needed in order to structure and direct the development of the solution. Indeed, the literature covers many solutions to the problems of optimisation. All of these solutions have been considered and reviewed by the author to a point where a clear understanding of the development involved in producing a solution for the Java virtual machine instruction set has been achieved. Let us first state our project aims in order to clarify the objective.

Considering the ideas put forward in the literature, a method has been chosen to procure an optimised instruction set for the Java virtual machine. The work involved in this project is aimed to produce an automated method or methods that procures information to be used to create or edit an instruction set to render it more optimal than the original. The exact methods used in this process are detailed in sections 4 and 5.

The requirements in this chapter have been split into functional, performance and non-functional categories. Functional requirements are concerned with things that the proposed system must or should do, whereas performance requirements deal with how well the system must or should work under load. Non-functional requirements are concerned with the things that are acquired or obtained from the development, what they are and how they should or must work. To clarify requirement importance, requirements in the following sections that are critical to the completion of the project are identified by the word “must”. Requirements that are of high importance but are not critical to the project are indicated by the word “should”. The word “may” indicates requirements that can be met during the course of development but have no bearing on the success or failure of the project. These requirements are meant purely for interest or completeness.

3.1 Analysis

It is clear that the instruction set for the Java virtual machine is very RISC centred. This project proposes that semantically rich instructions would actually reduce the amount of time the JVM has to spend on instruction dispatch, resulting in a faster running program. By reducing the amount of instructions needed to execute a function, a CISC style instruction set implies a more compact encoding of Java applications, which gives rise to faster download[1] times.

3.2 Functional requirements

3.2.1 Data

The data used with the program in order to maintain information on instruction usage must be from recent applications that are both in current use and reflect a wide range of use of concepts and facilities provided by the Java programming language. Although there is no foolproof method of obtaining this kind of data, to generally ensure these characteristics, it is sensible not to choose very small Java programs for example, a "Hello World" program. These types of programs would not only skew the results, but because the method used for printing a line of text to the screen is present in the Java API[2], there is no guarantee that this code has not been compiled for native machine optimisations that could affect the results obtained. With this in mind, none of the data must include Java code from the Java API. If code from the API is used within an application, that section of code must be discarded.

3.2.2 Statistics capture

The system must collect statistics relevant to the data used in section 3.2.1. The system must also collect data for an entire application. This means every valid class file used within the application, excluding class files from the Java API for reasons detailed in section 3.2.1.

Similar to the analysis of the Mesa instruction set (Sweet & James G. Sandman, 1982), instructions within the data must be normalised by the system before collection, or dynamically at run-time. A number of instructions seen for use with the Java virtual machine are

[1] Or upload or transfer times, depending on how it is viewed

[2] Application Program Interface — A set of routines provided in libraries that extends a language's functionality.

pseudo specialised instructions (in that they are not specialised by statistical or evolutionary computing techniques). Of which there are two categories that need to be expanded:

1. Embedded operand values
   These are generic instructions that are specialised with respect to some common case operands. An example of a generic instruction could be "push", which pushes a byte onto the stack. A specialised generic instruction with an embedded operand value could be "push-zero" or "push-one", where the instruction has no operands and the values 0 and 1 would be pushed on the stack respectively.

2. Multi-functional instructions
   These instructions perform more than one operation and can be separated out into their component parts. An example of a multi-functional could be "branch-if-compare-not-equal". This could be replaced with "compare" and "branch-if-zero".

In order to determine the trends in the use of different instructions in various programs, an intuitive criteria similar to that used by Sweet & James G. Sandman (1982) must be utilised when analysing the information procured by the capture process.

1. Static instruction frequency
   The number of times a instruction appears in the data.

2. Operand values
   For each bytecode, compile a histogram of the operand values.

3. Pair frequency
   Frequent occurrences of pairs of instructions sorted by frequency.

In addition, statistics may be compiled based on the following criteria:

4. Triplets or quartets frequency
   Frequent occurrences of triplets or quartets of instructions sorted by frequency.

3.2.3 Statistics analysis

Instead of haphazardly suggesting new instructions based on simple analysis of the statistics, a program may be constructed to suggest new instructions based on some form of computed gain (saving) on the overall instruction set. This idea is not dissimilar to ISGEN, the

instruction set generator (Bennett, 1988). Also using Bennett's ideas for DL (a design language), it would be useful if the program(s) generated were general enough so that the output instruction set changed dependent on a specification provided by the user. An development of this sort may be implemented if time allows, but should be considered a project extension.

3.3 Performance requirements

3.3.1 Speed

The system will be processing a large amount of data in order to get a unbiased set of results and also in order to discount anomalies. It should be written in a language that is known to provide fast execution times due to either simplicity or advanced optimisation techniques available. It should, through the use of good programming techniques, also be written in a fashion that enables it to run at optimal speed. With this in mind, the system should also be written in such a way that facilitates maintenance and re-use and should be written in a modular fashion to ease understanding and complexity. This will allow the system to be used for other bytecoded languages (with some customisation).

3.3.2 Memory

Again, the system will be processing large amounts of data and will effectively be performing two jobs concurrently. It will be performing a kind of interpretation of the data to extract instructions one by one and it will also be compiling statistics based on these instructions. Considering the proposed amount of data that needs to be processed, it is not inconceivable to assume that allocation failures could occur due to a shortage of memory. Therefore, the system should minimise memory usage by using finite buffers. This could be done by reading content from files incrementally rather than all in one go, by explicitly freeing memory for reallocation as soon as it becomes unused, or by utilising/implementing some kind of garbage collection algorithm to retrieve unreachable allocated memory from the system. The latter of which must be considered carefully with respect to section 3.3.1.

3.4 Non-functional requirements

3.4.1 Size

The result of analysis of the statistics collected must result in a number of additional instructions to the Java virtual machine instruction set that are specialised in some way to perform more semantically rich operations. The final result of the project should make use of these instructions to create a smaller equivalent program. Both studies performed by Sweet & James G. Sandman (1982) and Bennett (1988) make use of a peephole optimiser for this purpose. An implementation of such an optimiser must first decompose pseudo specialised instructions identified in section 3.2.2 and then must proceed to replace pairs, triplets or quartets of occurring instructions with their semantically equivalent specialised instruction. The original application and specialised application must then be compared with respect to file size and conclusions drawn. This process must be repeated for each application used as data in the statistics collection process.

3.4.2 Semantics

Any changes to the instruction set of the Java virtual machine and consequently any peephole optimisations made to existing applications must not change the semantics of the program. Changing the semantics of a program could result in fatal program errors and erroneous output. Both of which are undesired effects.

3.4.3 Execution

After the requirement defined in section 3.4.1 has been fulfilled, the logical progression would be to investigate whether the more compact instruction set allows the Java virtual machine to spend less time on work to decide which instruction to execute. The specialised instruction set may then be used to implement either an entirely new Java virtual machine or a modified version that supports the new instructions. A new or modified version of the virtual machine may be used in conjunction with any of the methods described in chapter 2 to procure an even more optimised machine capable of significantly better performance in addition to faster download time achieved because of smaller program size.

Chapter 4

High level system design

This chapter is a description of the high level design decisions made during the planning and implementation phases of the project. It includes decisions such as choice of language and external utility libraries as well as an outline of the proposed system architecture and some pseudo code demonstrating rough designs for how modules of the system are envisaged to work.

4.1 Choice of programming language

The first choice to be made for the project was which programming language to use. The two programming languages considered were Java and C. It made little sense to add the additional workload of learning a new language to the project and using languages that were not as familiar as Java and C could have also added to development time.

Of course, there are huge differences between the two languages. Java being object orientated, represents a more modern day approach to development separating code into higher level abstractions known as objects which provide some desirable features such as encapsulation, inheritance and polymorphism. C on the other hand is very much more akin to older languages such as Pascal or Fortran but is somewhat more terse. This is seen by the precise control of input and output that C provides. It enables programs to be written with minimal code and maximum efficiency.

For the development of the project, the C programming language was chosen. Between Java and C it was an obvious choice to maximise speed of execution. Java is ultimately an interpreted language, although as can be seen from the discussions in section 2.3 JIT

compilation is also used. The main problem here with using Java, is the inherent use of a virtual machine to run code. A necessary abstraction which means the Java programs encoded in the standard class file format will never run as fast as natively compiled C code. In addition interpreted programs lose many of the optimisations[1] made possible at compile time.

C was also a sensible choice with respect to some of the open source libraries that were being considered at the time. Interoperability between these additional libraries (which are all written in C or C++) and the proposed system was made easier by choosing C as the primary language. In the case of the ZIP file reader library chosen, it is the same code that has been adapted for the equivalent library in Java (java.util.zip).

Admittedly, C does come with its disadvantages. The very nature of precise control seen within C can be a recipe for disaster for the inexperienced user! Memory management is the main problem, where the user is given full control over memory allocation and deallocation. Through the use of pointers, the undisciplined user can unwittingly create memory leaks and invoke the unhelpful "segmentation fault" error message where pointers address unallocated or deallocated memory. Often segmentation faults are the hardest to track down and fix. Explicit control over memory management however is not always a disadvantage. The requirement in section 3.3.2 to minimise memory usage is helped by C's explicit control over when memory is allocated and deallocated. By explicitly deallocating memory after its use, system performance is not compromised by intrusive garbage collection methods such as stop and copy or mark and sweep.

One final disadvantage of the C language is the inherent ability for the developer to create totally unreadable programs. This however, is avoidable by following simple modern programming practices and exercising a little common sense.

4.2 Modular system architecture

One important design choice in the planning phase, was to make the system as modular as possible. This was done for a number of reasons which are simply good programming practice. Modular design firstly aids understanding of the system as a whole. Naming the modules with meaningful names and appropriate descriptions means that a developer viewing a system for the first time can identify and group modules easily. Modules were chosen in such a way that should the system need to be ported to a different object oriented

[1] Such as loop unrolling, loop jamming, induction variable elimination, common sub-expression elimination etc.

language, the objects that need to be identified and defined would be obvious. It also means that modules largely exist without dependencies between them, allowing them to be both flexible and re-usable.

The JAR reader was designed around a harvester and processor of sorts. The harvester is designed to extract data from a class file (which can be more widely described as a collection of class files in a JAR file). The processor receives data that the harvester collects and processes it in order to compile statistics. There are however, many other aspects of the JAR reader that perform additional but necessary purposes. The following list of files is a complete overview of the files involved in collecting statistics on a set of Java files.

4.2.1 Program driver

The program driver is where everything starts. Its essential purpose is to call the methods involved in setting up and shutting down the JAR file processor. Additionally, it receives the first error code that was thrown when a fatal error occurs. This is then translated into a human readable form.

It is worth noting that every module that makes up the JAR reader has its own status flags that can change when errors occur (fatal or non fatal). Calls to methods between modules always have a status as the return type, which is checked after the method returns in case an error flag was set. If one was, a message is printed describing what the module was trying to achieve and the status flag is returned. In this way, an error that occurs three modules deep in the program (for example) filters back through to jr.c. Each module that it passes through prints a message, giving an accurate trace of what was happening when the error occurred. The following code is an example of how this methodology works:

```
lStatus = createInstrStruct(instr);
if(lStatus!=JR_OK)
{
    logError("failed to create internal instruction structure","");
    return JR_CONSTRUCT_FAIL;
}
```

For method calls between modules that require a return type other than a status flag, two methods have been used which allow us to work with our error flags without losing functionality:

1. The method returns a status flag but takes a pointer to the return type it can use as one of the parameters. The status flag is still tested and if an error flag was not raised, it is safe to assume that the pointer data passed to the method now contains the return type data.

2. The module contains a global variable and has an additional method used to return the global variable to the relevant module. The status flag is still tested and if an error flag is not raised, the `getXXX()` method is called. Clearly this method is the weaker of the two and can be considered bad programming practice as there is no gaurantee that `getXXX()` will return a meaningful value.

4.2.2 Utilities

A set of general utilities is used by all the modules. They deal with some basic data type functions like extracting bytes, shorts or integers from a byte stream. They also handle some functions for converting strings to unsigned bytes and integers. Both these functions are simplified versions of the function `strtoul` found in `stdlib.h`.

A simple error logger that prints out messages depending on the log level (set at the command line) is provided. Log level options are [-d -w -e -n], which stand for debug, warn, error and none respectively. There are four types of messages that can be passed to the error logger which reflect the log levels listed above. In addition, there is also an info message.

Structures used to internally represent a Java class file in memory are also part of the utilities package. They are an exact reflection of the structures defined in the Java virtual machine specification and the accompanying maintenance notes. The file containing these structures is separate from any of the other headers partly to separate out elements of the code that need to be changed for re-use with another language and partly because of the fact that quite a few modules need to know about the data it contains.

4.2.3 Processor

This module was designed as the hub of the system. It provides a simple and intuitive control mechanism with which JAR files are harvested. Code 1 shows the high level pseudo code demonstrating the design of the internal workings of the processor. The "harvest" function design is described in more detail in section 4.2.4.

The processor's end goal is to procure a valid and correct stream of instructions from every class file for every JAR file in the directory it is given. In reality, the processor's job

Code 1 Processor design

```
process()
{
    while(more JAR files in directory)
    {
        getNextJarFile(directory);

        while(not finished processing JAR file)
        {
            getNextClassFile(JAR);
            harvest(next class file in JAR);

            giveEvaluator(next class file);

            getNextInstruction();

            createStatistics(instruction);
        }
    }
}
```

is slightly more complicated. The processor spends much of its time switching between different modules, dealing with data it is given and data that is requested from it. Indeed, to obtain a steady stream of instructions, there is quite a bit of work that needs to be done with a number of modules in the system. Hence the processor spends a lot of its time traversing each JAR file passing classes and references to various modules.

4.2.4 Class harvester

Clearly there is a distinction between having a handle on a class file and actually having a representation of the class file in memory. The harvester is designed to read a single class file and return a structured internal representation of it. The internal structure is used in conjunction with the evaluator (section 4.2.7) to extract individual instructions. The harvester is designed to translate the data in a class file by utilising the decompression library to read the file incrementally. The amount of data read by the harvester into internal buffers at each point is determined by the Java virtual machine specification. The harvester is essentially a mapping between the class file specification, the class file and the decompression library. Pseudo code 2 shows the approximate design of the harvester.

The harvester's job is to read a compressed Java class file and convert it into an internal representation. It is similar to a lexer, but tokenises high level aspects of the class file and so

Code 2 Harvester design

```
harvest(class file)
{
    for each class file section
    {
        readNextSection(class file);

        createInternalStructure(section);
    }

    return internal structure;
}
```

does not include individual instructions[2]. Each class file is broken down into the component structures that create the minimum representation of a class file that supports version 49.0 (the most recent class file format version). The Java virtual machine specification and maintenance notes detail the exact structure of class files, which are followed precisely by the harvester to obtain a correct and valid representation. Implementation of the harvester is covered in more detail in section 5.3.

4.2.5 External instruction specification

Early on in the planning process it was decided to make the instruction specification both external and human readable. A specification using XML[3] was designed to incorporate features allowing it to be re-usable not only with extensions to the work presented here, but also in future work that may not be with the Java virtual machine.

The process presented in this document is predominantly a static profiling technique. However, the system has been designed to ease extensions allowing it to perform dynamic profiling. The XML instruction specification was designed to incorporate runtime operand stack information before and after the instruction has been executed in addition to information regarding its operands. Information conveyed in each instruction specification allows the system to recognise instructions, their operands types and the state of the runtime operand stack (with regards to types) before and after the instruction has been executed. This design was influenced by the observation that instructions for the Java virtual machine differ based solely on type information for each item of the runtime operand stack. If the JVM instruction set did not incorporate comparison based branch instructions, partial evalua-

[2] Although code is extracted from the class files, it is stored in "Code" attributes as a byte array

[3] Extensible Markup Language

tion of an entire program could be performed based entirely on type information built from instructions and their specification. This would have made the process of dynamic profiling JVM code much easier but is clearly a lot to ask as any non trivial program would use constructs requiring these types of branch instructions.

As the specification design essentially describes type information on the runtime stack, it seems reasonable to propose that with a few extensions to the system, stack data information could be procured. Dynamic profiling could then be performed using existing system modules[4].

The discussion in section 3.2.2 requires that instructions are normalised either before collection or at run-time. Either way, a machine readable rules set and process for replacing bytecodes with alternative instructions was needed. Since the XML specification was already a machine readable form used to identify bytecodes, the design was easily extended to include alternative information in each instruction specification.

The exact implementation details of the XML instruction specification including runtime operand stack elements not used in this system are detailed in section 5.4.

4.2.6 XML code constructor

The design of the code constructor was a natural progression from the design of the XML instruction specification. Obviously some kind of reader is required that not only understands the specification, but can construct a meaningful internal representation of it. The code constructor is designed to search through the XML instruction specification to find a bytecode. Once found, an internal structure is created and passed back to the caller. The design of the code constructor in this way was mainly due to the fact that instructions in the stream are sequential (within each method) and not delimited in any way. Each bytecode needs to be taken from the instruction stream and its operands need to be evaluated to determine where the next bytecode starts. The XML code constructor provides the mechanism for generating the data needed for the evaluation. Pseudo code 3 outlines the basic design of the code constructor.

The evaluator cannot extract a bytecode from the byte stream without some help from the XML code constructor. The structures returned by the code constructor are allocated and initialised but mostly not populated. Selective population occurs when expected values and alternative values are set within the XML specification (this is discussed further in section 5.4). Population is mostly performed by the evaluator and is necessarily this way because

[4]The design of which are covered in sections 4.2.7 and 4.2.8

Code 3 Code constructor design

```
codeConstruct(bytecode)
{
    while(not found)
    {
        if(getBytecodeFromElement(element)==bytecode)
        {
            createInstructionStructure(element);
            return instructionStructure;
        }

         if(EOF(XMLFile))
         {
            moveToStart(XMLFile);
         }

         getNextXMLInstrElement(XMLFile);
    }
}
```

of the non-static run-time environment information that is only known by the evaluator. Information on the implementation of the XML code constructor can be found in section 5.6.

4.2.7 Evaluator

The evaluator is the caller that was mentioned in section 4.2.6. The evaluator is designed to do everything necessary to provide the statistics collector (section 4.2.8) with a valid and complete instruction. This is a simple cycle of fetching an instruction structure from the code constructor and evaluating the operands in the current instruction stream. This is complicated somewhat by instructions that have variable numbers of operands. Observations of the current Java virtual machine instruction set show that there are a few instructions that have operands dictating how many more operands follow. In addition, some instructions have a variable length padding of zeroed out bytes in order to align operands on 4 byte boundaries. For every combination of padding bytes, there is a different instruction form specified by the XML file. The evaluator has the job of matching instruction forms to instruction data in the stream. Once the relevant form has been identified, operands can be extracted.

In addition, the evaluator is designed to replace instructions with their alternatives, should they have any. The XML specification allows for any number of alternatives and thus the

evaluator needs to create a string of these alternate instructions. This logic is the key part of the normalisation process discussed in section 5.5. Alternative instructions replace the original instruction and may also contain operands (specified in the XML file) that need to be promoted from alternate operands based in the orignal instruction to real instruction operands in each respective alternative. Pseudo code 4 shows the design of the evaluator.

4.2.8 Statistics collector

The presence of the last 7 modules made the design of the statistics collector very simple. Since instructions are being procured one at a time, the statistics collector is designed to collect statistics one instruction at a time. This becomes a simple exercise of decomposing the instruction structure to suit the needs of the type of statistics it is set up to collect. Due to the inherent simplicity of the system, this module is highly customisable and effective at procuring statistics on an incremental basis. This adds another advantage, in that even if the process fails mid way through execution, statistics up to that point have already been collected and are still usable.

Code 4 Evaluator design

```
stream;
streamOffset;

evaluate(class)
{
    if(finishedEvaluating(class)==TRUE)
    {
        return NULL; /* a new class is provided if returning NULL */
    }
    else if(streamOffset>=length(stream))
    {
        stream=getNextMethodStream(class);
        streamOffset=0;
    }

    getBytecode(stream);
    increment(streamOffset);
    instructionStructure=codeConstruct(bytecode);
    matchForm(instructionStructure);

    if(hasAlternatives(instructionStructure)==TRUE)
    {
        for each(alternative) /* create new alternative instruction for each alternative */
        {
            nextInstructionStructure=codeConstruct(bytecode);
            instructionStructure->nextInstruction=nextInstructionStructure;

            for each(alternative operand in instructionStructure) /* promote alternative operands into real operands */
            {
                nextInstructionStructure->operand=instructionStructure->alternativeOperand;
            }
        }
    }

    for each(instructionStructure) /* extract operands from stream */
    {
        for each(operand)
        {
            extractOperand(stream[streamOffset]);

            if(isAlternative(instruction)==FALSE)
            {
                increment(streamOffset);
            }
        }
    }
    return instruction; /* result of evaluating a bytecode is a complete instruction */
}
```

Chapter 5

Disassembling the Java class file

5.1 Background on technologies utilised

5.1.1 JAR files

JAR files, with respect to the technology they use, are identical to the popular ZIP file format in all but file name extension. They allow Java applications with multiple classes to be bundled within a convenient package. Inspection of Sun's JAR file specification[1] shows one important way JAR files differ from their ZIP counterpart. When a JAR file is created, an optional META-INF directory is placed in the inventory. This directory specifies various properties about the Java application that is packaged within. One important property it describes is the main Java class of the application. This in turn, allows JAR files to be executable, and thus easier to use, store and share.

5.1.2 Zlib and Minizip

Since a large number of applications now come packaged in the JAR file format, it makes sense to create an API that can read from them. The open source library zlib[2] was used for this purpose. Zlib is written by the same people responsible for the compression and decompression methods in gzip (Jean-loup Gailly and Mark Adler respectively). As such, many of the methods seen in zlib are essentially the same as those used for gzip. Zlib itself was also adopted in version 1.1 of the Java Development Kit (JDK), both as a raw class (namely java.util.zip) and as a component of the JAR archive format.

[1] Available for download at: http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html
[2] Available for download at: http://www.zlib.net/

With this in mind, it seemed sensible to use the zlib library as the projects decompression component. However, because the zlib library is primarily used for compressing and decompressing gzip files, the open source Minizip library[3] is used in addition to zlib in order to decompress ZIP files (and JAR files).

5.1.3 Java class files

Java class files are the basic data file used throughout the project. They are semantically equivalent versions of Java programs (although sometimes other languages) compiled for use with a Java virtual machine. Each class file's data is arranged in a particular format that is dictated by Sun's Java virtual machine specification (Lindholm & Yellin, 1999) and the maintenance document (Sun Microsystems Inc., 2006), which provides existing changes, clarifications and amendments to the current specification released in 1999. The most recent format at the time of writing is major version 49, minor version 0. The project has been specifically coded to work with class files compiled with this class file format. However, due to the nature of the task, it is possible (in theory) to use class file format versions with lower major and minor versions as not all changes to the format effect the projects ability to read the file. Indeed, if an irregularity is detected in the file, an attempt will be made to skip over the unrecognised data.

5.1.4 The Expat XML Parser

Bytecodes found in Java class files are identified by an XML specification (see section 5.4). Using XML for this purpose provides a number of benefits:

- Structured — XML documents are structured in an intuitive manner that makes them easily human-readable.
- Valid — providing a machine-readable grammar which specifies which tags and attributes are valid allows automatic validation of the document. For large documents (like the one found in this project), quick validation can save hours of code debugging.
- Re-usable — XML is not just a set of rules to enable generation of custom markup. It is part of a whole family of technologies that work well together to make working with XML documents both useful and easy.

[3] Available for download at: http://www.winimage.com/zLibDll/minizip.html

The fact that the bytecode specification is encoded in XML means that even if there were to be additional work on this project that used a different format instead of XML, there would be a high chance that there will be a translator that will transform the XML into the desired format, saving the developer a lot of time.

Expat[4] is an open source library that will parse XML documents in a stream oriented fashion. Although there are many other XML parsers available, Expat is quite well known, fast and widely used — Expat is the XML parser behind the open source Mozilla project and is also the driving force behind Pearls XML parser. It is also written by James Clark, who is the technical lead at the W3C's XML working group which produced the XML specification (World Wide Web Consortium - W3C, 2004).

5.2 Process overview

The evaluator's job is to provide the processor with a series of valid instructions. These instructions are constructed by the code constructor so that the evaluator knows how much to read ahead in the binary stream to get to the next instruction. The evaluator literally fills in the blanks that the code constructor creates when given a bytecode. Each instruction can take a number of forms and have stack and operand values set. The code constructor is designed to not be able to return a fully prepared instruction on its own, and requires the help of the evaluator.

Control is continually passed between the evaluator, harvester, code constructor, statistics collector and the processor during the process. Initially, the processor passes an open class file to the harvester, where an internal class file structure is created. This class file structure is then passed to the evaluator by the processor. The evaluator returns instructions to the processor one by one from the class file it was given. Each instruction the processor receives is passed to the statistics collector, where statistics are compiled. When the evaluator has finished with a class file it asks the processor for another. If another exists in the JAR file, it is opened and passed first onto the harvester etc. Execution continues in this manner until all class files in a JAR file have been processed. Once this has been completed, the entire process is repeated for every JAR file available to the program.

To aid understanding of the flow of data within the system, figure 5.1 illustrates the main modules and the types of data that flow between them.

[4]Available for download at: http://expat.sourceforge.net/

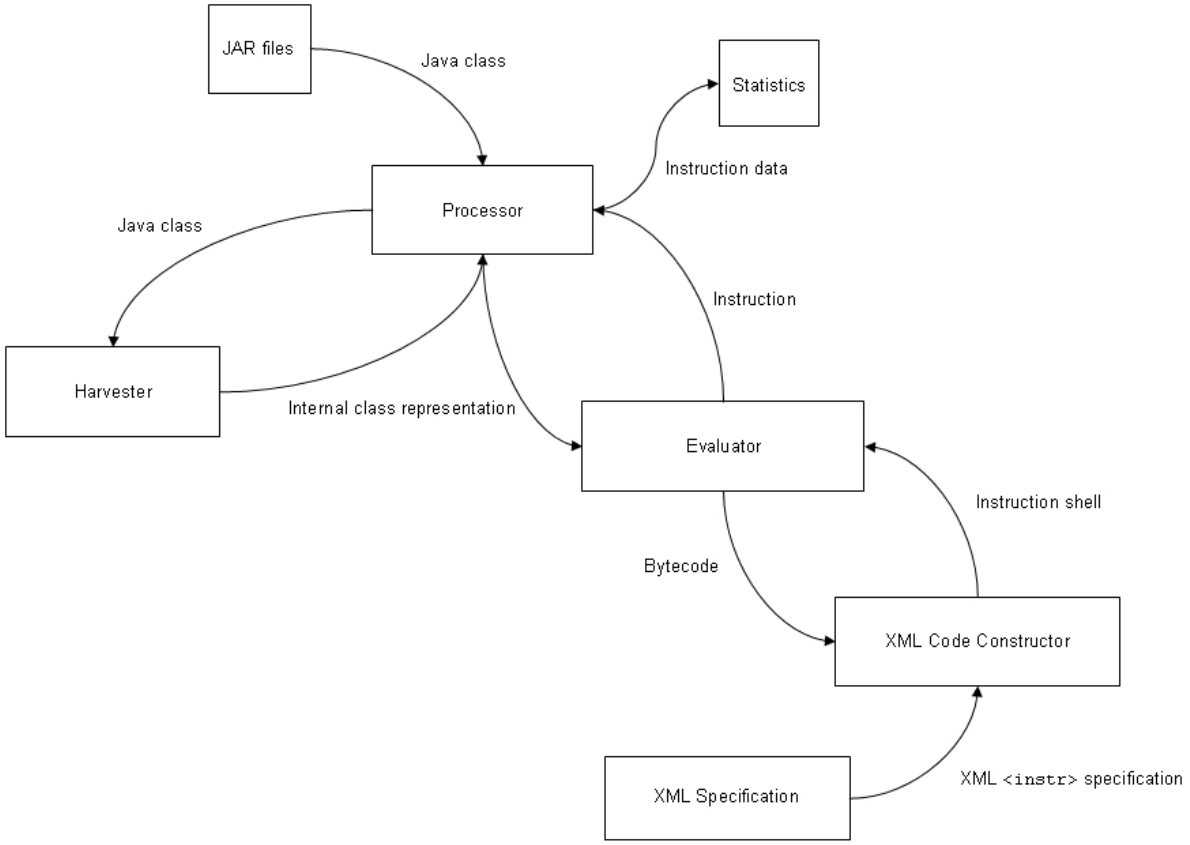

Figure 5.1: System architecture

5.3 Harvesting class files

The key component in class file decomposition is the processor. The goal of the processor is to obtain a stream of instructions complete with their operands. Only then can statistics collection begin. One of the main problems with obtaining this stream of instructions is that Java programs are made up of a number of class files, some or all of which are used during the execution of the program at one point or another. The problem is such that there is no means of easily extracting the data we require without a fair deal of processing beforehand.

A typical Java Class file is not a single stream of bytecodes and their arguments. The Java class file format is dictated by Sun's specification (Lindholm & Yellin, 1999). Each Class file is comprised of 7 main sections:

1. Versioning information — the first four bytes of a Java Class file contain the Hex value `0xCAFEBABE`, which uniquely identifies the file as a Java Class. There are also major and minor version numbers, which identify the class file format that is being used within the file. Implementations of JVMs can only normally work with a small subset of minor version numbered Class files as differing major versions identify significant structural changes.

2. Constant pool — the constant pool is a table which contains constant data used in the class at runtime. It cleanly separates instructions from layout information of classes, interfaces, class instances, and arrays. Hence instructions reference the symbolic information in this table. Some examples of the structure and information it represents are classes, fields, methods, integers, doubles and constant UFT8 strings.

3. Access flags — these are masks used to identify access permissions. Some familiar flags can signal public, private or final status. Although access flags are applicable to the class as a whole, often methods, fields and even some attributes also have access flags.

4. Interfaces — this is a simple list of direct super interfaces of the class or interface type. Note that each of these are a reference into the constant pool.

5. Fields — these are class or instance variables declared in a class.

6. Methods — each method present in the class, including class initialisation and instance initialisation methods, has a representation listed here.

7. Attributes — the attribute representations are the most widely used element in a class file. Attributes describe certain properties of representations and indeed are representations themselves. Hence attributes can also have attributes. Although attributes are listed here as a top level class element, they only describe attributes of the class as a whole, thus they can also belong to fields and methods to describe more in depth properties. There are nine predefined attributes, of which the Code attribute is most useful. This is because it specifies the stream of instructions associated with its parent.

We notice that the data contained within a class file is not always of fixed length. Some structures do not have the fixed width that a single short, integer or a double may have. A simple example is a UTF8 string, for example, "Hello World!". When reading the class file, we know where the string starts, but there is no way of knowing in advance how many bytes long the string is. The solution is that at compile time, the length of the data is computed and explicitly stated before the structure. This solves one problem but raises another. The problem is that although the length of the data is specified, these values are measured in uncompressed bytes, which is not useful when packaged as a compressed JAR. This means that even though we know how much data we want to skip over, we still need to read over a certain amount of compressed data to obtain the uncompressed data.

For the processor to obtain a stream of instructions from a JAR file, a mechanism for reading compressed class files is needed. The Harvester performs the job of extracting all the information out of a single class file into internal structures that are easier to navigate. The fact that a lot of reading is done because of the problems described above is reason enough to simply read a whole class file into memory. For the purposes of this project this was the easiest and least time consuming solution. Clearly there are a number of other solutions that allow the use of finite memory buffers and because of the modular structure of the JAR reader, implementation of this behaviour should be eased and can be implemented at a later date.

When reading a stream of bytes from a Code attribute, the structure of bytecodes is constructed from their XML specification. The code constructor performs this job. When given a bytecode, the XML file is queried until a match is found. The matching `<instr>` element is then parsed. For each element contained within, space is allocated, ready to be filled with the data from the instruction stream.

5.4 XML bytecode specification

The Extensible Markup Language (XML) Document Type Definition (DTD) describes a form of XML document that allows developers to specify a bytecode instruction set for processing by some machine. The machine is not dissimilar to a lexer. Data such as the number and size of operands are dictated by the XML file. Using this information, a stream of bytecodes is processed by referring to the XML file for the internal structure of each bytecode that is encountered. The main differences between this instruction specification and a lexer is that lexers tend to operate using delimiters, keywords and regular expressions. All of which are suitable for parsing a high level language but for bytecoded instructions there are no delimiters, keywords or regular expressions that can be processed by generic lexical analysers.

The simple DTD allows specification of the structure of instructions using XML. In addition to the aforementioned data included in the XML file, runtime operand stack information that many virtual machines now use is also stored. More specifically this information includes data such as sizes for each operand, the order in which they appear and the expected arguments, should they be known before runtime. One important feature of the instruction specification is the ability to specify one or more alternate instructions and their operands. Some languages have attempted to create more semantically rich instructions that can be broken down into a series of more simplistic commands. This is particularly true of the Java instruction set, which contains specialised instructions that appear to waste space and could be used for more useful instructions that will result from statistical analysis.

The DTD is intended to be general enough for use by other bytecoded languages whilst still providing a structured approach to specification of bytecodes. Elements and attributes are named in a general manner in order to be applicable to languages other than Java. Possible attribute values will be implementation specific, although doubtless some attribute values will be used for more than one implementation. A specific set of possible attribute values is presented in the following discussion of instruction specification using the XML format. Code 5 shows a simplified instruction specification which is intended to be more readable than the DTD. In keeping with tradition, square brackets denote elements that can optionally be left out.

Each instruction is contained in the `<instr>` tag which has three attributes. `bytecode` describes the byte coded value that represents the instruction that is being defined. Due to the very nature of bytecodes, it can have the values 0 through to 255. `name` is the human readable name given to the instruction and can contain any number and combination of alphanumeric

Code 5 Simplified XML instruction specification

```
<instr bytecode="" name="" [branch=""]>
  [<f1-f12>
    [<operand type="" [order=""]></operand>]
    [<prestack type="" [order=""]></prestack>]
    [<poststack type="" [order=""]></poststack>]
    [<alt bytecode="" [form=""] [order=""]>
       <operandval [order=""] [inherit=""]></operandval>
       <prestackval [order=""] [inherit=""]></prestackval>
       <poststackval [order=""] [inherit=""]></poststackval>
    </alt>]
  </f1-f12>]
</instr>
```

Table 5.1: `branch` attribute values for `<instr>` elements

attribute value	**description**
`always`	the instruction always invokes a jump
`eq`	the instruction invokes a jump when two operands are equal
`ne`	the instruction invokes a jump when two operands are not equal
`lt`	if one operand is less than the other a jump is invoked
`le`	if one operand is less than or equal to the other a jump is invoked
`gt`	if one operand is greater than another a jump is invoked
`ge`	if one operand is greater than or equal to the other a jump is invoked

characters and special symbol characters (such as underscore) that do not conflict with any characters used in XML documents. The `branch` attribute is a flag specifying the type of branch that occurs when this instruction is executed. This flag signifies the (potential) end of a code block, which cannot be considered in statistical analysis if a pair occurs over a branch boundary. Clearly an instruction immediately after a branch instruction cannot be concatenated with the branch instruction to form a single instruction. Table 5.1 shows the attribute values proposed for the Java language `branch` attribute.

The `f` elements 1 through 12 are used to specify various forms an instruction can take. Certainly in Java bytecode some instructions take more than one form depending on the type of the operands residing on the operand stack and in some cases the actual operands used by the instruction. In essence, the `f` elements separate the various forms an instruction can take. However, the numbering of the `f` elements affects the evaluation order of an instruction. Lower `f` numbers have priority over higher `f` numbers. This simply allows hints

Table 5.2: `type` attribute values for `<operand>` elements

attribute value	description
`byte`	eight bit byte
`ubyte`	eight bit unsigned byte
`nxubyte`	`n` * eight bit unsigned bytes

for more common instruction forms which although will not have any effect on statistical analysis, will speed up evaluation time if known. The largest number of forms a Java bytecoded instruction can take is twelve[5] although the actual number of instructions that utilise more than one `f` element is much less.

The `<operand>` tag specifies operands (or arguments) to the bytecode, of which there can be zero or more. Each operand must have a `type` attribute associated with it. The `type` attribute gives information on the size of the operand and thus where the next operand or instruction begins. Table 5.2 shows the values proposed for the `type` attribute when used with Java bytecoded instructions. An optional `order` attribute specifies the order in which operands appear and should be used at all times except when all operands are of the same type. If the `branch` flag is set in the parent `<instr>` element, ordering values `0` and `1` are used as the two arguments to be compared. Operand `0` is the first operand in the stream, operand `1` the second, and so on. The body of the `<operand>` tag[6] contains the value of the operand should it be known before runtime. For example, In Java bytecode this occurs when zero value padding bytes are required to guarantee four byte alignment of operands.

The `<prestack>` and `<poststack>` elements are very similar to `<operand>` elements. Instead of describing the operands to an instruction, they describe the state of the runtime operand stack used by the Java virtual machine. The difference between pre and post stack operands and operands declared by the `<operand>` tag is `type` values that are used. Table 5.3 shows these values. The `order` attribute works in much the same way as for any `<operand>` element. The highest ordering is on top of the stack and the lowest on the bottom. `<prestack>` elements describe operands popped from the stack during instruction execution and `<poststack>` elements describe operands pushed onto the stack as a result of execution of the instruction.

As mentioned previously, a number of alternate instructions can be specified so that spe-

[5] Hence only twelve elements have been defined for this purpose in the DTD.

[6] By this we mean the data between the two tags `<operand>` and `</operand>`. Empty operand tags (or indeed any XML tag) may have no closing tag and are encoded as `<operand />`. See the W3C XML specification at http://www.w3.org/TR/2004/REC-xml-20040204/

Table 5.3: `type` attribute values for `<prestack>` and `<poststack>` elements

attribute value	description
`ref`	reference to some object
`null`	reference to null object
`int`	integer
`nxint`	`n` * integer
`double`	double
`float`	float
`long`	long
`empty`	special post-condition that stack is empty (cleared)
`cat1`	category 1 computational type, either int, float, ref, ra
`cat2`	category 2 computational type, either long, double
`classvar`	fields of classes (static fields, known as class variables)
`instancevar`	fields of class instances (non-static fields, known as instance variables)
`nargs`	`n` * arguments from invocation of a method

cialised instructions can be expanded if necessary. `<alt>` elements must specify the bytecode of the instruction that will replace the instruction the `<alt>` element belongs too. If more than one instruction is needed, it is necessary to set the `order` attribute, so that each alternate instruction is executed in the correct order. The `form` attribute is a numeric value in the range 1 to 12 which can specify which form of the alternate instruction is needed. If no `form` value is set, `<f1>` is used.

Often when specifying an alternate instruction or set of instructions it is favourable to also specify the values for some or all of the operands. Some specialised Java bytecodes push constant values onto the stack with no operands. Instructions of this kind are expanded such that the general push instruction is called with a constant operand. The bodies of `<operandval>`, `<prestackval>` and `<poststackval>` each represent the values to be passed to the alternative instruction for the `<operand>`, `<prestack>` and `<poststack>` elements respectively. As with most other elements in the specification, these alternative values have an optional `order` attribute for when there is more than one operand value to be passed. `order` values of `<operandval>`, `<prestackval>` and `<poststackval>` are matched with `order` values of `<operand>`, `<prestack>` and `<poststack>`.

In some cases it is desirable for an alternative instruction to inherit its operand and operand stack values from its parent. This behaviour is seen when comparative branch instructions (whose target is calculated from its operands) are expanded into two alternative instructions.

The first of which performs a comparison and the second of which performs the branch. It is necessary for the second instruction to inherit the branch target from it's parent. For this purpose `<operandval>`, `<prestackval>` and `<poststackval>` each have an optional `inherit` attribute which specifies whether or not their values must be inherited from their parents. If the `inherit` attribute is not set, it is assumed to be false. It is worth noting here the distinction between values (or inherited values) expressed in the `<operandval>` tag and values expressed in the `<operand>` tag[7]. The former are used to specify values to be used instead of those present as operands in the instruction stream. The latter is used to express values that should appear as operands in the operand stream. They are meant to allow selection of instruction form based on operand values.

Finally, the total set of instruction specifications is encapsulated in the `<code>` tag which has only one required attribute: `lang` which specifies the bytecoded language that the containing elements represent.

5.5 Normalising the Java virtual machine bytecodes

Normalisation aids analysis by reducing the size of the instruction set and also increases the scope for more specialised instructions, that have their basis in concrete statistical analysis of the use of instructions in Java applications. One method of generating CISC style instructions is to start with a few instructions that have relatively little semantic content and build upon that. This will be the method used here.

The Sweet & James G. Sandman (1982) approach to normalisation appears similar conceptually to the processes used by the evaluation module. The idea was to transform the input stream into a normalised semantically equivalent output stream as a co-routine with their pattern matcher. Here the same operation is done, but statistics for the whole data set are generated concurrently as the input data is fed into the system, replacing the need for a pattern matcher.

The current Java virtual machine instruction set makes use of 201 of the 256 possible instructions. These are opcodes 0 through 201 (where opcode 186 is unused). In addition, opcodes 202, 254 and 255 are reserved for use by the Java virtual machine. This leaves 51 possible instructions that can be specialised before normalisation. The following discussion details the normalisation specification used in the implementation of the system to acquire a further 50 instructions from the original instruction set. In addition, further improvements

[7]This explaination applies also to `<prestackval>` and `<prestack>` as well as `<poststackval>` and `<poststack>`

Table 5.4: const bytecodes

Name	Semantics
iconst_m1	push integer constant -1 onto the operand stack
iconst_0	push integer constant 0 onto the operand stack
iconst_1	push integer constant 1 onto the operand stack
iconst_2	push integer constant 2 onto the operand stack
iconst_3	push integer constant 3 onto the operand stack
iconst_4	push integer constant 4 onto the operand stack
iconst_5	push integer constant 5 onto the operand stack
lconst_0	push long constant 0 onto the operand stack
lconst_1	push long constant 1 onto the operand stack
fconst_0	push float constant 0 onto the operand stack
fconst_1	push float constant 1 onto the operand stack
fconst_2	push float constant 2 onto the operand stack
dconst_0	push double constant 0 onto the operand stack
dconst_1	push double constant 1 onto the operand stack

to this figure are suggested but not used in this work.

5.5.1 const

The const bytecodes consist of 14 pseudo specialised instructions that push a constant onto the operand stack. As with most bytecodes in the Java virtual machine instruction set, the letter preceeding the instruction denotes the type information. i for integer, l for long f for float and d for double. These bytecodes and their respective semantic meaning are shown in table 5.4. For the remainder of this chapter, bytecodes similar in design that differ only by type may have their type information replaced by the string <t> to illustrate that the discussion relates to all types of the same instruction.

All of the <t>const bytecodes are pseudo specialised with respect to their operand. Due to the fact that each of these instructions perform variations of the same function, it should be possible to devise a singular expanded instruction for each type leaving us with 11 spare instructions. Ideally expansion of each of these instructions would be through the use of an existing general instruction that takes a single operand as its argument, for example <t>const [operand]. Unfortunately there is no such instruction in the current instruction set. Alternatively, we could keep the <t>const_1 instructions and use an add or subtract instruction(s) to increment or decrement the constant to the desired value. When comparing these two

methods for iconst, we see that the second of the two only requires less bytes to achieve the equivalent functionality of iconst_0, or iconst_2. These each use three bytes and could be compiled sequentially as: iconst_1, isub or iconst_1, iconst_1, iadd respectively. However, the equivalent of iconst_3 requires 5 bytes to encode in this way. iconst_4 requires 7 bytes and iconst_5 requires 9 bytes. We see that iconst_5 almost doubles the instruction size that would be needed for a iconst [operand] style instruction; 5 bytes (1 byte bytecode and 4 byte integer). In fact, this style instruction does not even need to take an integer operand. We observe that because the Java virtual machine does not provide this instruction, no values greater than 5 or less than -1 will ever be used. The iconst [operand] instruction can be modified so that the operand is a signed byte, and rely on the virtual machine to do the conversion to an integer. This has two main benefits — firstly, the instruction size is constantly 2 bytes, outperforming the iadd method in all but the most trivial case (when the integer value 1 needs to be pushed onto the stack). Secondly, it gives a larger range of constant integer values that can be pushed onto the stack (values -128 through 127). This same methodology (but with some modification) can be applied to const instructions of each different type.

For the system implementation, the new instructions <t>const [signed byte operand] were created and used as alternatives for each of the <t>const instructions. The result of the above analysis concludes that it makes little sense to implement the <t>add method as in the majority of cases it simply serves to grow the code base. We can also see that it does not aid statistical analysis — if the <t>add method is used as the equivalent for <t>const_2, and this particular instruction was used often, we would notice an increase in <t>const_1, <t>add pairs. This in turn would lead us to specialise <t>const_1 to (for example) <t>constadd_1 which is <t>const_2 expressed differently. The same conclusions can be drawn by using <t>const [operand] but instead we are looking for occurrences of <t>const with a particular operand; a simpler and easier process overall.

5.5.2 ldc_w, ldc2_w, goto_w and jsr_w

The interesting point about these instructions is that they are all variations of their respective equivalent instructions (their meanings are covered in table 5.5). The _w is used to signify that they take extra one byte or two byte operands. This, on first inspection seems reasonable enough but further investigation uncovers an instruction called wide, which is used to extend a local variable index by additional bytes. It modifies certain instructions, effectively making them [opcode]_w instructions.

Table 5.5: ldc_w, ldc2_w, goto_w and jsr_w bytecodes

Name	Semantics
ldc_w	push integer or float from run-time constant pool (wide index)
ldc2_w	push long or double from run-time constant pool (wide index)
goto_w	branch always (wide index)
jsr_w	push address of next instruction and jump to subroutine (wide index)

ldc instructions reference the run-time constant pool and goto and jsr both reference instruction addresses. The existing wide instruction extends a reference to a local variable on the stack in all cases, so from this point of view it is easy to see why it was separated in this way. Although the normalisation process is involved with reducing and simplifying the instruction set, it can be argued that changing the meaning of the wide instruction would mean an increase in time taken to decide what route to take when the JVM is given this instruction. It would also be a significant change to the workings of the Java virtual machine — something which the author is hesitant to invoke.

For initial implementation, these instructions were not expanded into a wide [opcode]_w pair, pending analysis of the frequency these instructions occur and the need for extra instructions (should we not have enough spare).

5.5.3 load and store

<t>load and <t>store instructions load or store a variable in the current frame. They are expanded in the same way as <t>const expressions, but require no new instruction to be created due to the fact that <t>load [unsigned byte operand] and <t>store [unsigned byte operand] already exist as instructions in the full instruction set. Specialised <t>load and <t>store instructions all have alternatives set to their respective <t>load [unsigned byte operand] or <t>store [unsigned byte operand] instructions. In total, an extra 20 instructions are gained by expanding these pseudo specialised instructions.

5.5.4 pop

Both pop instructions, pop and pop2, pop items off the run-time operand stack as per the description in table 5.6. We see that pop is the generic instruction and pop2, the specialised one. Expanding pop2 becomes complicated because of the different forms it can take. So

Table 5.6: `pop` bytecodes

Name	Semantics
`pop`	pop the top operand stack value
`pop2`	form 1: pop top two operand stack values form 2: pop the top operand stack value

far, we have only considered instructions with one form for specialisation, making semantic preservation almost trivial. With `pop2` however, the operation of the instruction changes, dependent on the run-time operand stack types. Fortunately our XML specification (by design) can express the different forms an instruction can take and describe suitable alternatives based on each form. Thus the semantics for `pop2` can be preserved over expansion.

Forms are chosen based entirely on type information (see section 4.2.5). `pop` will pop the top operand from the operand stack if it is a category 1 computational type[8] (an integer, float, reference or return address). `pop2` will pop the top two operand stack items if they are computational 1 type, but will pop the top operand stack item if it is a category 2 computational type (a long or double). This is easy to express in our XML specification, code 6 shows how it can be achieved.

Code 6 Expressing `pop` and `pop2` bytecodes in the XML specification

```
<instr bytecode="87" name="pop">
  <f1>
    <prestack type="cat1"/>
  </f1>
</instr>
<instr bytecode="88" name="pop2">
  <f1>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
  </f1>
  <f2>
    <prestack type="cat2"/>
  </f2>
</instr>
```

Our expansion of `pop2` to `pop` instruction(s) needs alternative instructions for both forms it

[8]Computational types are defined in the Java virtual machine specification.

can take. The implementation of this alone is incomplete, as there is no pop form that can deal with operand stack items of type category 2. Hence we add a new form to `pop` and specify which `pop` form each alternate instruction should take for `pop2`. The result is shown in code 7. In actual fact, by making this small change to the number of forms an instruction can take, we are explicitly changing the inner workings of the Java virtual machine — something which was avoided in section 5.5.2.

Code 7 Expressing `pop2` bytecode alternatives in the XML specification

```
<instr bytecode="87" name="pop">
  <f1>
    <prestack type="cat1"/>
  </f1>
  <f2>
    <prestack type="cat2"/>
  </f2>
</instr>
<instr bytecode="88" name="pop2">
  <f1>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <alt bytecode="87" form="1" order="0"/>
    <alt bytecode="87" form="1" order="1"/>
  </f1>
  <f2>
    <prestack type="cat2"/>
    <alt bytecode="87" form="2"/>
  </f2>
</instr>
```

There are other larger issues with performing this expansion that are concerned primarily with the method of choice for statistics collection. This manifests itself within the fact that we do not know operand stack type information at all parts of the program. Clearly we could build a method by which operand stack type information could be procured (this is trivial given the XML specification) but it is destroyed whenever a branch instruction is hit. In response to this we could suggest evaluating every possible path through the program to ensure we always have the correct stack type information whenever we encounter a `pop2` instruction. As the reader can probably imagine, this is completely unfeasible for anything but trivial applications.

Sadly the expansion of `pop2` does not appear possible with the statistical profiling method used for this system. The above analysis shows that it could be possible but requires a

stronger evaluation mechanism to succeed.

5.5.5 if_icmp, if_acmp, ifnull and ifnonnull

There are in total 16 separate `if` instructions (table 5.7) that can cause a branch on comparison in the Java virtual machine instruction set. The expansions specified here for `if_icmp` and `if_acmp` appear to be slightly unintuitive as no instructions are salvaged in the process. This is due to the fact that half the component instructions derived from the original instructions do not already exist in the instruction set. This has occurred before with the `<t>const` instructions (section 5.5.1) but still resulted in a number of salvaged instructions. The difference with `if_icmp` and `if_acmp` instructions, is that they are specialised with respect to function, not value. By this we mean that `<t>const` instructions were combined with their actual operands, meaning that when expanded, the resulting `<t>const [operand]` instruction could be re-used for each specialised instruction. Now we are expanding `if_icmplt` (for example) to get the two instructions `icmplt` and `ifeq`. The former of which does not exist, and needs to be defined in the XML specification. This example instruction (`icmplt`) is envisaged to push 0 onto the operand stack if one value on the operand stack is less than another and 1 if the converse.

For the Java virtual machine instruction set, a new comparison instruction needs to be created for each of the `if_icmp` and `if_acmp` instructions. This is necessary since no generic integer comparison instructions exist. Hence the newly crafted instructions take back the instructions that could have been gained if we were dealing with instructions specialised by operand value.

`if_null` and `if_nonnull` are interesting because although they are specialised by function (a branch and a comparison), they can both use the result of the `NULL` comparison function when expanded, meaning we still salvage one instruction. A new instruction `isnull` is crafted, which is envisaged to push 0 onto the stack if the top operand stack value is `NULL` and 1 otherwise. The formal definitions of `isnull`, `if_icmp` variants and `if_acmp` variants are shown in code 8.

5.6 Constructing instructions from the XML specification

The code constructor provides a mechanism to determine bytecode operands and operand stack data. A simple matching algorithm (discussed in section 5.7) that determines the current state of the machine and matches it with the relevant instruction form allows for

Code 8 Example XML specification for integer and reference comparison instructions

```
<instr bytecode="203" name="icmpeq">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="204" name="icmpne">
  <!-- Instruction body same as instruction 203 //-->
</instr>
<instr bytecode="205" name="icmplt">
  <!-- Instruction body same as instruction 203 //-->
</instr>
<instr bytecode="206" name="icmpge">
  <!-- Instruction body same as instruction 203 //-->
</instr>
<instr bytecode="207" name="icmpgt">
  <!-- Instruction body same as instruction 203 //-->
</instr>
<instr bytecode="208" name="icmple">
  <!-- Instruction body same as instruction 203 //-->
</instr>
<instr bytecode="209" name="acmpeq">
  <f1>
    <prestack type="ref"/>
    <prestack type="ref"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="210" name="acmpne">
  <!-- Instruction body same as instruction 209 //-->
</instr>
<instr bytecode="211" name="isnull">
  <f1>
    <prestack type="ref"/>
    <poststack type="integer"/>
  </f1>
</instr>
```

Table 5.7: The 16 branch on comparison bytecodes

Name	**Semantics**
`ifeq`	branch if value(integer) = 0
`ifne`	branch if value(integer) $\neq$ 0
`iflt`	branch if value(integer) < 0
`ifge`	branch if value(integer) $\geq$ 0
`ifgt`	branch if value(integer) > 0
`ifle`	branch if value(integer) $\leq$ 0
`if_icmpeq`	branch if value1(integer) = value2(integer)
`if_icmpne`	branch if value1(integer) $\neq$ value2(integer)
`if_icmplt`	branch if value1(integer) < value2(integer)
`if_icmpge`	branch if value1(integer) $\geq$ value2(integer)
`if_icmpgt`	branch if value1(integer) > value2(integer)
`if_icmple`	branch if value1(integer) $\leq$ value2(integer)
`if_acmpeq`	branch if value1(reference) = value2(reference)
`if_acmpne`	branch if value1(reference) $\neq$ value2(reference)
`ifnull`	branch if value(reference) = NULL(reference)
`ifnonnull`	branch if value(reference) $\neq$ NULL(reference)

this. The number of bytecode operands procured via this method means that the evaluator knows how much to increment the stream in order to get to the next bytecode. This is how the evaluator can keep providing the code constructor with a stream of bytecodes.

The code constructor also provides a mechanism for storing instruction data after evaluation has taken place. The structure of the corresponding instruction has to be found in the XML specification for each instruction received by the conde constructor. The Expat parser requires handler functions to be registered in order to deal with certain events such as the start of a tag or a character data section. As the document is passed, these handler functions are invoked. On receipt of a new bytecode, the document is parsed (from the last parse point reached[9]) until a start element handler function is called that matches the `<instr>` tag and has a bytecode equal to the bytecode that is being looked up. Once found, handler functions are invoked for each element within the `<instr>` tag. The handler functions allow the instruction to be incrementally constructed until the closing tag (`</instr>`) is found.

By following the code DTD, the code constructor remains language independent although regrettably, the DTD is not parsed by Expat and hence the XML specification is not vali-

[9]Should EOF be reached during this process, the file is wrapped back to the start

dated. code constructor assumes the code is valid and well formed.

5.7 Instruction evaluation

A single method in the evaluator is responsible for (attempting to) procure another instruction for the processor. The evaluator can either supply a new instruction or not. Each time the evaluator returns to the processor, the evaluator's status is queried to deduce whether the process has been successful or some action is needed to be taken by the processor to ensure the next call to the evaluator succeeds. Normally this is a request for a class (as methods are invoked) but can also be a "finished" status flag to indicate that all the instructions for the current class have been procured. This translates into a request for the next class file in the JAR. If there are no more classes, the next JAR file is obtained and a harvest of the first class in it. The new class is then passed to the evaluator.

The evaluator has an internal state[10]. As instructions are requested by the processor, the internal state changes depending on (for example), whether a new method is being evaluated or whether the evaluator is just continuing to evaluate the current instruction stream.

On the initial receipt of a class file, it is necessary for the evaluator to compile a reference list of all the methods with instruction streams. Before processing can take place on each of these streams, a preliminary pass over every method is performed. This preliminary pass is needed to find the targets of all branch instructions. The reason for this being that an instruction preceding a branch target cannot be considered as a common pair with the branch target. The reader may notice that this reason is the inverse of why instructions that cause branches cannot be considered as common instruction pairs with their successors.

Branches between methods are not permitted in the Java virtual machine except by special "invoke" instructions, which always start execution at byte zero of a new method and hence do not have predecessors to consider. Thus calculated branch targets are done on a per method basis. For each method, a list is constructed of all branch targets and checked against prior to returning a completed instruction to the processor. If the instruction starts at a branch target byte, a flag is raised and later considered by the statistics collector when compiling statistics on instruction pairs.

The first pass is necessary due to the nature of the statistics collection process. The system is required to operate within finite buffers, meaning that memory consumption should not grow indefinitely as instructions are acquired and statistics collected. Capturing statistics

[10]Of which, most possible states are known to the processor

one at a time allows this to happen, but means that the system can only consider the current instruction (and possibly other future instructions) at any given point in the process. The implications of branch instructions are that the target can be anywhere in the current method, before or after the current instruction. The latter is a problem to the system as it means that an instruction previously processed needs to be flagged as a branch target and then possibly not considered for statistics capture. Clearly this would be difficult to implement, if not impossible.

Evaluation is restricted by the processor to just the class files in each JAR. Although this could easily be extended to all class files including those from the Java API, it would not give a true reflection of the programs written by everyday users of Java. This restriction is made possible as the evaluator "doesn't care" which class it is currently dealing with. Indeed class and method information is encoded in the instructions it creates, but this is solely for the statistics collection module. Evaluation would proceed in exactly the same way if a completely random set of classes was used instead.

If the evaluator were to be used for dynamic profiling then it would have to "care" which class it was dealing with. The class given to the evaluator would have to contain the stream of instructions for a particular method in a particular class. We also lose our ability to discard Java API classes in this way as we need to know actual return values from methods to be able to continue executing instructions.

Chapter 6

Statistical analysis

6.1 Test data

Our first functional requirement (section 3.2.1) concerns the data to be used with the program, which has to be recent, varied, widely used and sizeable. All the applications used as test data were recent, if not the most recent versions at the time of writing. In addition, they are all, relatively large Java applications with reasonably varied functions.

1. JADE (Java Agent DEvelopment Framework) 3.4 — an open source platform for peer to peer agent based applications. The code for JADE includes a codec package which contains simple encoder and decoders for various formats such as Base64 and Hexadecimal.
2. Protege 3.1.1 — a free, open source ontology editor and knowledge-base framework.
3. JEdit 4.2 — a mature and well-designed programmer's text editor with 7 years of development behind it.
4. Apache Tomcat 5.5.16 — server libraries from the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies.
5. BCEL (Byte Code Engineering Library) 5.1 — a toolkit for the static analysis and dynamic creation or transformation of Java class files.
6. ANTLR (ANother Tool for Language Recognition) 2.7.5 — a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions.

7. Azureus 2.4.0.2 — an implementation of the BitTorrent protocol using the Java language.

In total 26 separate JAR files totalling over 15MB of compressed programs were used. It is important to remember that even though this seems like a lot of code, a class file is not entirely executable code and other files such as the META-INF directory found in most JAR files also take up space (although the majority of files in the JARs we obtained were class files). This translates into a much smaller code base for each JAR, and in retrospect, more Java applications could have been used. Our subset equates roughly to the size in bytes that was used by Sweet & James G. Sandman (1982), so perhaps 24 years later a larger set should be used. Certainly this would help to eliminate anomalies, but from initial observations with a data set half the size of the current, similar trends were seen. Note the author does not suggest procuring a larger data would not be beneficial, merely that the benefit of doing so would remain to be seen.

An initial run was compiled which recorded statistics without the expansions used by the normalisation process[1]. The test data contained a total of 1,488,710 individual instructions totalling just over 3.1 million bytes of instruction data. After normalisation we see a rise in both of these figures as should be expected. Total individual instructions rises to 1,521,718 and the total bytes of instruction data rises to over 3.6 million bytes. The figure for total individual instructions gains 0 or more instructions for each expanded instruction, but we see a more dramatic rise in the figure for number of bytes of instructions. This is because bytes from additional instructions as well as additional bytes from explicit operands increase this figure.

6.2 Instruction frequency

Instruction frequency can be portrayed in two different senses. Firstly instruction frequency is presented as the times an instruction has occurred in the data set. This has limited functionality as it can only give us information on which instructions are used often. Following the instruction analysis performed by Bennett (1988), we see that a more important statistic to us is frequency of instruction size as a percentage of the total size of all instructions. This results in larger instructions gaining more prominence than smaller instructions. In hindsight, this is more than acceptable as it provides us with information relating to our objective — reduce the static size of a program. Knowing instruction frequency as a func-

[1]This is done by executing the program with the `-x` flag.

tion of instruction size can direct us to places where instruction specialisation should be primarily focused. Unless otherwise stated, frequencies used in this chapter are of this second kind.

Table 6.1 shows the frequency distribution of our normalised instruction set. To make the table more readable, instruction frequencies of less than 0.1% are omitted. A sample of the full results is presented in Appendix C.1. We see that only a few instructions make up the bulk of the frequency distribution and that they are all instructions used in assignment, procudure call and conditional statements. It is worth noting that the top 5 instructions make up over 50% of the code size and that specialising these instructions with respect to their operands alone would result in massive savings.

Contrasting these results with those obtained by Sweet & James G. Sandman (1982) and Bennett (1988), we see a striking resemblance. Indeed, experimental observation by Bennett showed that assignment, procedure call and conditional statements made up 82.7% of BCPL instructions, and we see similar figures here.

All but 5 instructions were used (these are seen in table 6.2), which considering the amount of data processed tends to suggest that they are not needed. We do however see that discarding some of these instructions altogether may limit the Java virtual machine. The last three instructions (`wide, goto_w and jsr_w`) all reference addresses outside of their usual range. The standard `goto` instruction takes a signed short as it's argument[2]. In addition, the Code attribute in the Java class file is limited to 65535 bytes (an unsigned short). It is not implausible that a standard `goto` statement could not express the address it needs to reach with a signed short operand. The same type of argument can be applied to `wide` and `jsr_w`.

The fact that these instructions were unused means that our decision not to expand these instructions in section 5.5.2 was the correct one to make, but poses the question whether our test data was adequate. One can argue that adequate test data would have shown full usage of all instructions. However, considering modern day programming practices (that favour short and many methods) and the object oriented nature of Java, it is far from surprising that `wide`, `goto_w` and `jsr_w` are not used at all.

[2]This is actually two unsigned bytes used to construct a 16 bit signed short.

Table 6.1: Normalised instruction set frequency distribution

Name	Frequency	Semantics
aload	20.11%	Load ref from local variable
invokevirtual	14.09%	Invoke instance method; dispatch based on class
getfield	7.94%	Fetch field from object
invokespecial	5.41%	Invoke instance method; special handling for certain methods
invokeinterface	4.94%	Invoke interface method
ifeq	4.43%	Branch if equal to zero
iconst	3.63%	Push int constant
new	3.50%	Create new object
astore	3.15%	Store ref into local variable
iload	3.15%	Load int from local variable
ldc	3.03%	Push from constant pool
putfield	2.96%	Set field in object
goto	2.79%	Branch always
invokestatic	2.77%	Invoke a class (static) method
getstatic	1.69%	Get static field
dup	1.62%	Duplicate top stack value
bipush	1.35%	Push byte
istore	1.17%	Store int into local variable
checkcast	1.04%	Check object type
ifne	0.72%	Branch if not equal to zero
ldc_w	0.51%	Push from constant pool (wide index)
iinc	0.50%	Increment local variable by constant
sipush	0.50%	Push short
putstatic	0.45%	Set static field
isnull	0.44%	Test for null value
pop	0.41%	Pop top stack value
ldc2_w	0.36%	Push long or double from constant pool (wide index)
lload	0.32%	Load long from local variable
return	0.30%	Return void from method
aconst_null	0.30%	Push null
instanceof	0.29%	Determine if object is of given type
anewarray	0.24%	Create new array of reference
iadd	0.23%	Add int
aastore	0.19%	Store into reference array
lconst	0.18%	Push long constant
athrow	0.17%	Throw exception or error
aaload	0.14%	Load reference from array
areturn	0.14%	Return reference from method
arraylength	0.14%	Get length of array
ifle	0.13%	Branch if less than zero
icmpge	0.13%	Int comparison greater than or equal
icmpne	0.12%	Int comparison less than or equal
ireturn	0.12%	Return int from method
isub	0.11%	Subtract int
lstore	0.10%	Store long into local variable

Table 6.2: Unused Java virtual machine instructions

Name	Semantics
dup2_x2	Duplicate the top one or two operand stack values and insert two, three, or four values down
swap	Swap the top two operand stack values
wide	Extend local variable index by additional bytes
goto_w	Branch always (wide index)
jsr_w	Jump subroutine (wide index)

6.3 Instruction to operand frequency

Section 6.2 has set the scene for further analysis. Frequency of instruction data now shows us where to focus our optimisations. The normalisation process in section 5.5 has resulted in a total of 101 instruction spaces that can be filled with more specialised instructions. This section analyses a number of interesting instructions from our frequency distribution (table 6.1) with a view to creating specialised instructions with respect to their argument. At this point, we are also looking at instruction arguments so that a new instruction can be created with a smaller argument (as per Bennett's design rules discussed in section 2.6.2).

It should be noted that predicted savings in this section are based solely on instruction size with a particular operand. We have not yet considered instruction pairs, which may lower the values of these figures. To that end, the predicted savings expressed here should be viewed as an absolute maximum that could theoretically be procured. Predicted savings are calculated by the following equation:

$$instruction\ frequency * (instruction\ size - specialised\ instruction\ size)$$

6.3.1 aload

aload loads a reference from a local variable and is by far the most dominant instruction found in our frequency distribution. Figure 6.1 shows the frequency distribution of operands for aload, derived from the sample data seen in Appendix C.1. We see a very steep but smooth curve indicating scope for a uniform operand value specialisation, which supports some of the previous instruction specialisation (up to an operand value of 3) as seen in the instruction set before normalisation. Bennett observed the same trait when analysing the stack offsets (which is equivalent to the aload operand value) to local variables in BCPL. Our main concern here is that frequency for an operand value of 3 is less than a quarter of the frequency for an operand value of 0. Specialising with respect to the operand value 3

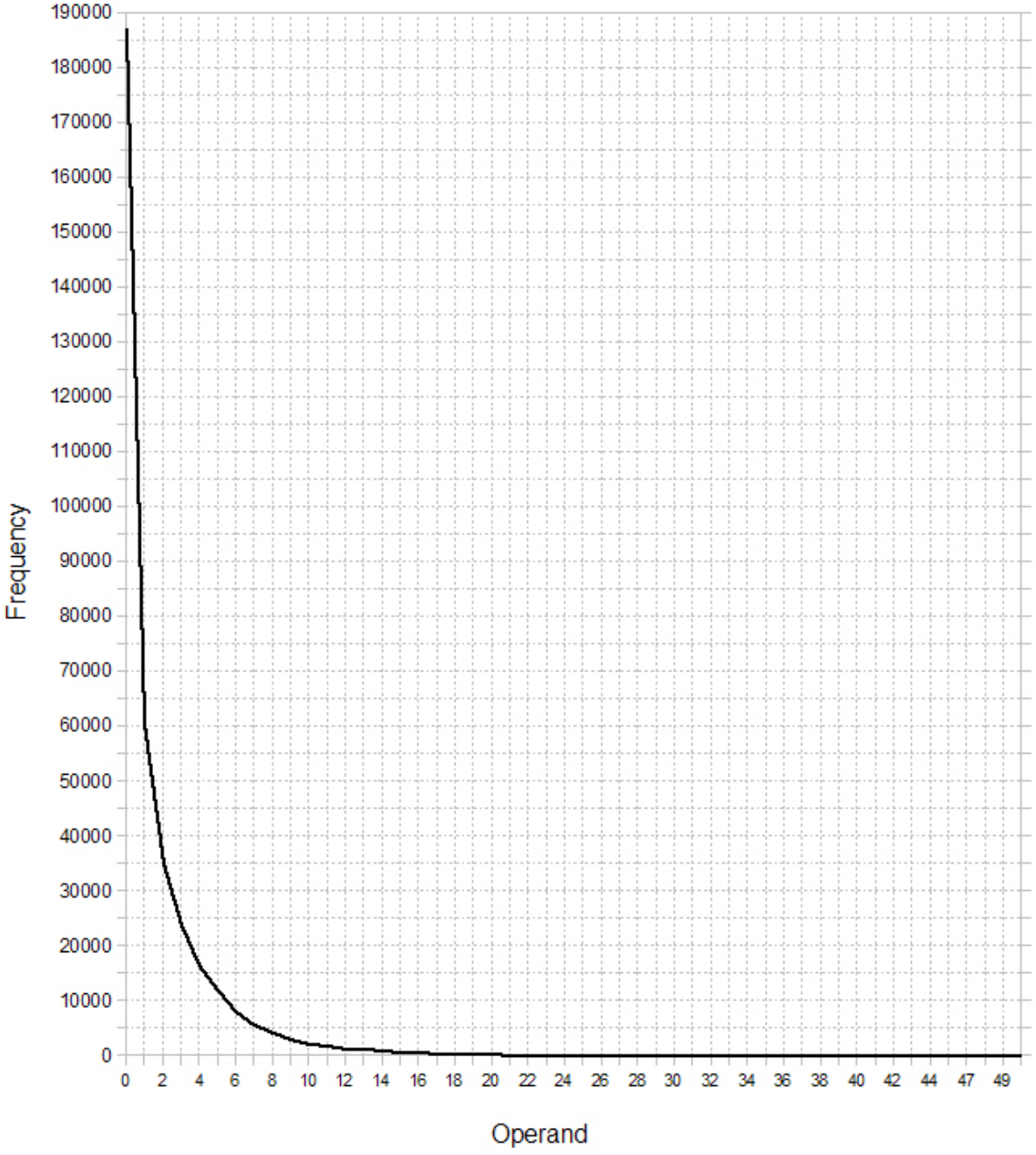

Figure 6.1: `aload` operand frequency distribution

heralds a predicted saving of 0.65%. Approximately just a fifth of what is predicted to be saved by specialising with respect to 0 — a more respectable 5.11%. The reader's attention is drawn to the fact that because this is the most frequent instruction (with respect to the total instruction data), this figure is quite possibly going to be the highest single saving we obtain. We have seen here that specialising with respect to 3 has given a much lower predicted precentage saving, but in the long run we do have 101 instructions that we can use. Specialising with respect to values such as 4 or 5 may also procure low predicted savings (0.45% and 0.32% respectively), but combined applications will incrementially bring our total predicted percentage saving up.

The frequency of `iconst` seems to suggest that it would also be a good candidate for specialisation. Predictably, we only see operand values of those expressed by the specialised version of `iconst`, which is because there was no generic `iconst` instruction prior to normalisation, allowing us to push an arbitrary integer constant onto the stack. The normalisation process for this variety of instructions was primarily concerned with procuring more instructions for specialisation. Here we have found that the specialisation of `iconst` was justified (as it appeared highly in our frequency distribution) and so the specialisation for this instruction will be reinstated.

Although `aload` and `iconst` appear to support the specialisations seen before normalisation, almost all of the other specialisations with respect to value do not. The exceptions being `astore` and `iload`. `astore`'s most frequent operand is 2, which if specialised would exhibit a 0.29% predicted saving. `iload`'s most frequent operand is also 2, exhibiting a predicted saving of 0.26%. The reader is drawn to the fact that our predicted savings for `aload` with respect to 4 or 5 outperform both of these values, and in fact, `aload` specialised with respect to 6 heralds a predicted saving similar to `iload`. Now, it appears reasonable to change the specialisation of `aload` to include values 4 and 5, and maybe 6. Although if at the end of our analysis we find that we need more instruction spaces, specialisation with respect to 6 is probably expendable.

Other values previously specialised with respect to operand are almost not even worth considering. The next most frequent instruction of this variety is `istore`, whose most frequent instruction could only save us 0.08% — a tiny proportion of what is seen to be possible by `aload`. Previously specialised instructions lower down in the frequency distribution are just not used frequently enough to warrant specialisation with respect to operand.

6.3.2 invokevirtual

`invokevirtual` is the main (and most used) method call instruction. It falls into a class of instructions that the author can only describe as "may operands, low frequency". This means that although this is seen to be the second most frequent instruction in our instruction frequency distribution, operand frequency is spread sparsely between 818 recorded operands. Figure 6.2 consequently does not have labelling for the x-axis as there was no space. Unfortunately due to the huge amount of possible operands, it is hard to know where to begin specialising with respect to operand. The graph peaks are more erratic for lower operand values and lower operand values are generally more frequent. This is something we do not want to see and it means that we cannot decisively say which value(s) to specialise. Coupled with this we see that even the most frequent operand only occurs 3386 times, giving rise to a predicted saving of just 0.19%. If we specialised the top 5 of these instructions then it could in theory be possible to gain almost an extra percent saving, but we would ideally like to find a better use for these instructions.

A number of graphs were compiled for top ranking instructions that had not been previously specialised showing that most of them also fall into this class. This is a shame considering that they make up a large proportion of all instruction data. One saving grace is that 95.71% of instruction values for `invokevirtual` fall in the range 0-255 (an unsigned byte). The normal `invokevirtual` operand is a short, so we can see more than enough scope to create a new instruction with a smaller argument. The maximum predicted saving for this specialisation would be 4.94%. We see the same trait in the `getfield` instruction, which has an even greater 97.82% of instruction values but because of it's lower frequency has a less predicted saving of 2.59%.

A slight variation on this comes in the form of `ifeq` which has a spread of operand values from -5712 to 5732. Its operand was originally considered to be a signed byte, which can express 94.33% of the operand values recorded (a 1.39% saving). This was later discarded in favour of the observation that by using an unsigned byte we still expressed a greater precentage of the operand values recorded (95.46%), yielding a saving of 1.41%. This decision is justified by the further observation that values in the range 0 to -128 make up only 2.20% of the total instruction values.

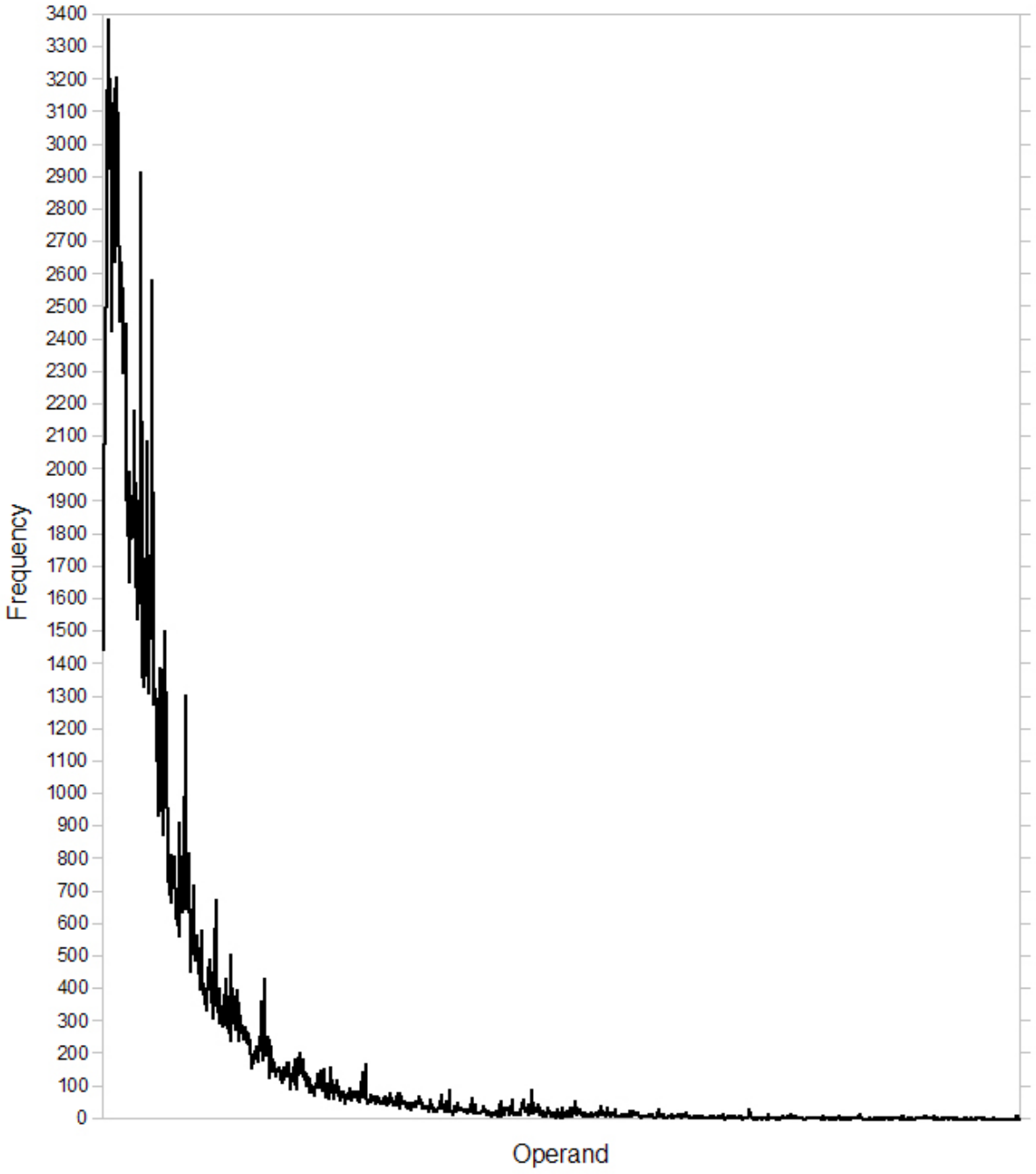

Figure 6.2: `invokevirtual` operand frequency distribution

Table 6.3: Normalised common instruction pair frequency distribution

Name	Pair	Frequency
aload	getfield	12.31%
aload	invokevirtual	7.68%
aload	aload	6.59%
new	dup	4.67%
astore	aload	4.35%
putfield	aload	3.85%
ldc	invokevirtual	3.66%
aload	invokeinterface	3.60%
getfield	invokevirtual	3.36%
getfield	aload	3.16%
aload	invokespecial	2.95%
dup	invokespecial	2.04%
aload	iconst	1.90%
isnull	ifeq	1.78%
aload	iload	1.74%
aload	ldc	1.53%
aload	new	1.47%
aload	putfield	1.32%
getfield	invokeinterface	1.10%
iconst	invokevirtual	1.09%
dup	aload	1.08%
aload	invokestatic	1.06%
iconst	istore	1.00%

6.4 A refined instruction set

Table 6.3 shows the distribution of pairs in terms of pair size frequency as a percentage of the total data size. Again, to make the table more readable, pairs whose frequency accounts for less than 1% of the total data size have been omitted. The full results can be seen in Appendix C.2. Common instruction pairs allow us to see places where instructions (and space) can be saved if two instructions are concatenated together to make just one. A quick look at table 6.3 shows impressively that `aload` appears in well over half of the 23 instruction pairs listed, which is not surprising given that it is the most frequent instruction.

We immediately notice that there are very few instructions in this table that do not have any operands, and for those that do we see that some of them could be specialised according to our discussion in section 6.3. We could continue and create specialised versions of the

pairs in table 6.3, but we notice that a more powerful optimisation would be to not only consider these pairs, but also pairs of the instructions already specialised by operand value. This would allow us to increase the specificity of the generated instructions, possibly saving on specialised instructions that are not needed. Bennett's methodology did not take this first step, but instead assumed independence of instructions, leading to unfavourable combinations. One such example was the concatenation of IMMEDIATE and MINUS, with a specific operand value. The observation was that both instructions appeared often and that a common operand to IMMEDIATE was 0. The system concluded that the most common immediate value to be subtracted is 0. Common sense tells us otherwise.

A refined instruction set that incorporated the features discussed in section 6.3 was submitted to the system. Admittedly, some of the optimisations were simply putting back specialisation that was present before normalisation. However, specialisations that were not needed were not replaced. The refined instruction set included our `aload_4` and `aload_5` instructions and our system was modified to use unsigned byte operand values for certain instructions. These new instructions were postfixed with `_n` to signify narrow operand values.

The exact actions taken were as follows:

1. Remove bytecode 203. This was our general `iconst` instruction that was derived from the normalisation process to express arbitrary values.
2. Reinstate the following specialised instructions as they have proved to be space saving (`<v>` is used here to signify all specialised values previously existent in the Java virtual machine instruction set):
   - `iconst_<v>` (bytecodes 2-8)
   - `astore_<v>` (bytecodes 75-78)
   - `iload_<v>` (bytecodes 26-29)
3. Add the following new bytecodes with specialised operand values:
   - `aload_4` (bytecode 203)
   - `aload_5` (bytecode 216)
4. Add the following new bytecodes with smaller arguments:
   - `invokevirtual_n` (bytecode 217)
   - `getfield_n` (bytecode 218)

- `invokespecial_n` (bytecode 219)
- `invokeinterface_n` (bytecode 220)
- `ifeq_n` (bytecode 221)

6.5 Common instruction pairs

Our first observation was a dramatic decrease in the total instruction bytes collected. Our refined instruction set bytes collected were 76.33% of the total bytes collected for the normalised instruction set and more importantly 87.81% of the total bytes collected for the original instruction set. It is striking that although our normalisation process has removed a number of supposidly specialised instructions, the edition of just 7 more specialised instructions has lead to an over 12% saving in program size, and we are yet to properly consider commmon instruction pairs.

Table 6.4 (continued in table 6.5) shows the new frequency distribution for our refined instruction set. Appendix C.3 shows a sizeable sample of the actual results obtained. They show us instantly the effect our instructions with smaller arguments has had on the distribution. The frequency of each of these relative to their wider indexed equivalents is quite large, so we do not see a huge change in the relative rankings of these instructions. With the `aload` family of instructions however, we see that frequency is spread predominantly over three instructions; `aload_0`, `aload` and `aload_1`. Hence the change in ranking is to be expected.

It is nice to see that most of our optimisations, with respect to value have appeared within our >0.1% boundary. `iconst_4` and `iconst_5` are the noticeable absentees which suggest that our decision to reinstate all of the `iconst` instruction specialisations was misinformed.

Clearly our refined instruction set has shown us that even with minimal effort we can derive a more CISC style instruction set by simply considering just two of Bennett's design rules. Table 6.3 showed us that whilst it was possible to concatenate a number of operands, our statistics were only strong enough to express the most general cases where these instructions could be used i.e. by instructions that accepted arbitrary operand values. We noticed that if we could procure data on which operands these instructions effected, we could create a super specialised instruction which could be the concatenation of two instructions and their operands.

The data in table 6.6 shows the distribution of successor pairs in terms of pair size frequency as a percentage of the total data size, for our refined instruction set. Appendix C.4 shows the complete results. Our criterion for selecting valuable pairs for specialisation

Table 6.4: Refined instruction set frequency distribution (Part 1)

Name	Frequency	Semantics
invokevirtual_n	11.78%	Invoke instance method (narrow index); dispatch based on class
getfield_n	6.78%	Fetch field from object (narrow index)
aload_0	6.70%	Load ref from local variable at index 0
invokeinterface_n	4.91%	Invoke interface method (narrow index)
new	4.59%	Create new object
invokespecial_n	4.51%	Invoke instance method (narrow index); special handling for certain methods
ldc	3.97%	Push from constant pool
putfield	3.88%	Set field in object
ifeq_n	3.69%	Branch if equal to zero (narrow index)
goto	3.65%	Branch always
invokestatic	3.62%	Invoke a class (static) method
aload	2.36%	Load ref from local variable
getstatic	2.21%	Get static field
aload_1	2.18%	Load ref from local variable at index 1
dup	2.12%	Duplicate top stack value
iload	2.12%	Load int from local variable
astore	2.09%	Store ref into local variable
bipush	1.77%	Push byte
istore	1.53%	Store int into local variable
checkcast	1.37%	Check object type
aload_2	1.26%	Load ref from local variable at index 2
iconst_0	1.00%	Push int constant 0
ifne	0.94%	Branch if not equal to zero
aload_3	0.85%	Load ref from local variable at index 3
iconst_1	0.81%	Push int constant 1
invokevirtual	0.79%	Invoke instance method; dispatch based on class
ldc_w	0.66%	Push from constant pool (wide index)
iinc	0.66%	Increment local variable by constant
sipush	0.65%	Push short
aload_4	0.59%	Load ref from local variable at index 4
putstatic	0.59%	Set static field
isnull	0.58%	Test for null value

Table 6.5: Refined instruction set frequency distribution (Part 2)

Name	Frequency	Semantics
pop	0.53%	Pop top stack value
ldc2_w	0.47%	Push long or double from constant pool (wide index)
aload_5	0.42%	Load ref from local variable at index 5
lload	0.42%	Load long from local variable
return	0.39%	Return void from method
aconst_null	0.39%	Push null
instanceof	0.38%	Determine if object is of given type
astore_2	0.38%	Store ref into local variable at index 2
astore_3	0.35%	Store ref into local variable at index 3
iload_2	0.34%	Load int from local variable at index 2
invokespecial	0.33%	Invoke instance method; special handling for certain methods
iload_1	0.33%	Load int from local variable at index 2
invokeinterface	0.33%	Invoke interface method
anewarray	0.32%	Create new array of reference
iload_3	0.31%	Load int from local variable at index 3
iadd	0.31%	Add int
astore_1	0.27%	Store ref into local variable at index 1
ifeq	0.26%	Branch if equal to zero
aastore	0.24%	Store into reference array
lconst	0.24%	Push long constant
athrow	0.23%	Throw exception or error
getfield	0.23%	Fetch field from object
iconst_2	0.21%	Push int constant 2
aaload	0.19%	Load reference from array
areturn	0.19%	Return reference from method
arraylength	0.18%	Get length of array
ifle	0.17%	Branch if less than zero
icmpge	0.17%	Int comparison greater than or equal
icmpne	0.16%	Int comparison less than or equal
ireturn	0.16%	Return int from method
isub	0.15%	Subtract int
lstore	0.13%	Store long into local variable
iconst_m1	0.13%	Push int constant -1
newarray	0.11%	Create new array
iastore	0.11%	Store into int array
iconst_3	0.11%	Push int constant 3

Table 6.6: Refined common instruction pair frequency distribution

Name	Pair	Frequency
aload_0	getfield_n	8.94%
new	dup	6.12%
ldc	invokevirtual_n	3.81%
putfield	aload_0	3.61%
getfield_n	invokevirtual_n	2.74%
dup	invokespecial_n	1.98%
isnull	ifeq_n	1.72%
aload_1	invokevirtual_n	1.47%
aload_0	invokevirtual_n	1.41%
getfield_n	aload_0	1.29%
aload_0	invokespecial_n	1.21%
aload_0	aload_1	1.18%
ldc	invokestatic	1.15%
aload	invokevirtual_n	1.10%
getfield_n	invokeinterface_n	1.01%
aload_0	new	1.00%

is ideally to choose frequently occurring pairs containing large instructions that occur frequency individually. Fortunately by creating smaller arguments for frequently occurring large instructions, we have already satisfied some of our criterion. Notice also that the most frequent instruction pairs are predominantly comprised of the most frequent individual instructions. Out of the 4000+ pairs that were recorded, this makes our task of selecting candidates for specialisation much easier.

The fact that there are less pairs that each make up >1% of the total instruction data tells us that by using our refined instruction set when considering pairs has actually increased the specificity, and thus enabled us to create specialised instructions that can have more impact on the static size of a program. Considering this, and the fact that much of our criterion for selecting instructions seems to have already been fulfilled, it makes sense to simply select the most frequent instruction pairs in table 6.6 for concatenation. We have more than enough instruction spaces to do this and thus there is no reason why any extension of this work cannot utilise our statistics and the remaining spare instructions to create further specialised instruction suggestions. However, the remaining instructions would probably not exhibit the same amount of saving due to their lower frequency (although collectively they may do). Our goal to demonstrate significant savings is the reason why further instructions have not been considered.

6.5.1 New instructions

This section is simply a formal definition of the new instructions derived from the instruction pair statistics in table 6.6. Instructions are created as a simple concatenation of their names and an underscore. As all of these instructions are only specialised statically, the run-time operand stack information does not change and is not included here. Changes to the run-time stack data, as a result of executing the instruction, should be considered to be the result of sequentially executing it's two component instructions separately. The details of which are described in detail in the JVM specification.

1. `aload_0_getfield_n`
   Size: 2 bytes
   Operands: 1 unsigned byte
   Semantics: Load ref from local variable at index 5 then fetch field from object at narrow index described by the operand.
   Comments: This is an advanced concatenation of an instruction specialised by operand and an instruction with a smaller argument. If these were executed as their normalised equivalents (`aload` and `getfield`), they would have a total size of 5 bytes.

2. `new_dup`
   Size: 1 bytes
   Operands: none
   Semantics: Create new object then duplicate top stack value
   Comments: This is a simple concatenation of two frequent instructions and indeed frequent pairs.

3. `ldc_invokevirtual_n`
   Size: 3 bytes
   Operands: 2, both unsigned bytes
   Semantics: Push item from run-time constant pool at the index of the first operand then invoke instance method based on the narrow indexed second operand.

4. `putfield_aload_0`
   Size: 3 bytes
   Operands: 1 unsigned short
   Semantics: Set field in object at the constant pool index of the first operand then load ref from local variable at index 0

5. `getfield_n_invokevirtual_n`

   Size: 3 bytes

   Operands: 2, both unsigned bytes

   Semantics: Fetch field from object at the narrow index of the first operand in the constant pool then invoke instance method based on the narrow indexed second operand.

   Comments: Executed as their normalised equivalents (`getfield` and `invokevirtual`), they would have a total size of 6 bytes.

6. `dup_invokespecial_n`

   Size: 2 bytes

   Operands: 1 unsigned byte

   Semantics: Duplicate top stack value then invoke special instance method based on the narrow indexed operand.

7. `isnull_ifeq_n`

   Size: 2 bytes

   Operands: 1 unsigned byte

   Semantics: Branch to address described by the operand (relative to the address of the instruction) if reference is null.

   Comments: As their normalised equivalents these instructions would have a size of 4 bytes. Note this instruction could be expressed as `ifnull_n` if we use the pseudo specialised notation seen in the original instruction set.

8. `aload_1_invokevirtual_n`

   Size: 2 bytes

   Operands: 1 unsigned byte

   Semantics: Load ref from local variable at index 1 then invoke instance method based on the narrow indexed operand.

9. `aload_0_invokevirtual_n`

   Size: 2 bytes

   Operands: 1 unsigned byte

   Semantics: Load ref from local variable at index 0 then invoke instance method based on the narrow indexed operand.

10. `getfield_n_aload_0`

    Size: 2 bytes

    Operands: 1 unsigned byte

Semantics: Fetch field from object at the narrow index of the operand in the constant pool then load ref from local variable at index 0.

11. `aload_0_invokespecial_n`
    Size: 2 bytes
    Operands: 1 unsigned byte
    Semantics: Load ref from local variable at index 0 then invoke special instance method based on the narrow indexed operand.

12. `aload_0_aload_1`
    Size: 1 byte
    Operands: none
    Semantics: Load ref from local variable at index 0 then load ref from local variable at index 1.

13. `ldc_invokestatic`
    Size: 4 bytes
    Operands: 1 unsigned byte, 1 unsigned short
    Semantics: Push item from run-time constant pool at the index of the first operand then invoke class (static) method at the constant pool index of the second operand.

14. `aload_invokevirtual_n`
    Size: 3 bytes
    Operands: 2, both unsigned bytes
    Semantics: Load ref from local variable at index of the first operand then invoke instance method based on the narrow indexed second operand.

15. `getfield_n_invokeinterface_n`
    Size: 5 bytes
    Operands: 4, all unsigned bytes
    Semantics: Fetch field from object at the narrow index of the first operand in the constant pool then invoke interface method based on the narrow indexed second operand. The third operand is the count and the forth is always zero (see the JVM specification).

16. `aload_0_new`
    Size: 3 bytes
    Operands: 2, both unsigned bytes
    Semantics: Load ref from local variable at index 0 then create a new object.

Predicting savings for these new instructions is not as easy as analysing the instruction size and the frequency the pair occurred because many of them utilise the same instructions, leading to problems with overlaps between pairs. If anything, the predicted saving would be the absolute maximum that could be saved using these instructions, which would of course be assuming that none of them overlap. To obtain a slightly more accurate savings figure, a peephole optimiser is needed. A rudimentary peephole optimiser was written to perform this job. It does not output an instruction stream, but performs equivalent substitutions in terms of instructions used and their size, in order to gain statistics on savings made. It is referred to as rudimentary because it does not consider the instruction data as a whole, but merely substitutes instructions when it is seen possible. It is essentially an extension to the system's statistics collector, and started life as the mechanism for swapping instructions with long arguments for instructions with short arguments.

6.6 Deriving a new Java virtual machine instruction set

The results from the peephole optimiser were quite impressive, our 16 new instructions made a total code size reduction of 11.91% from the code size of our refined instruction set. This means that the code size acquired from the operand value specialisations and our 16 new instructions is 77.36% of the size of the original Java code (a total reduction of 22.64%). The results from the peephole optimiser are displayed in table 6.7 and shows the predicted saving and actual saving that was acquired. The bytes saved are expressed as a percentage of the total size of instructions collected when using the refined instruction set. Appendix C.5 shows the actual results obtained from the peephole optimiser.

Predicted bytes saved are calculated using the following formula:

$$pair\ frequency * (individual\ instruction\ sizes - specialised\ instruction\ size)$$

Peephole savings are calculated in a similar way:

$$instruction\ frequency * (individual\ instruction\ sizes - specialised\ instruction\ size)$$

The distinction is that in the peephole equation, instruction frequency is the actual times a pair was replaced with an instruction, whereas pair frequency is just the number of times a pair occurred in the total data. The upshot is that instruction frequency gives the actual saving, because it eliminates overlaps. Individual instruction sizes here should not be confused with the individual instruction sizes from the original instruction set.

Table 6.7: New instructions — bytes saved as a percentage of the total size of instructions

Instruction		Bytes saved	
#	Name	Predicted	Peephole
222	aload_0_getfield_n	2.97%	2.71%
223	new_dup	1.53%	1.37%
224	ldc_invokevirtual_n	0.95%	0.95%
225	putfield_aload_0	0.90%	0.90%
228	isnull_ifeq_n	0.57%	0.57%
233	aload_0_aload_1	0.59%	0.50%
230	aload_0_invokevirtual_n	0.47%	0.45%
229	aload_1_invokevirtual_n	0.49%	0.42%
232	aload_0_invokespecial_n	0.40%	0.31%
235	aload_invokevirtual_n	0.28%	0.28%
234	ldc_invokestatic	0.23%	0.23%
237	aload_0_new	0.25%	0.16%
231	getfield_n_aload_0	0.43%	0.12%
226	getfield_n_invokevirtual_n	0.68%	0.10%
227	dup_invokespecial_n	0.66%	0.08%
236	getfield_n_invokeinterface_n	0.17%	0.02%

getfield_n_invokeinterface_n would have a total size of 4 bytes individually, not 6 bytes. Hence we are expressing savings on top of the savings found from our refined instruction set.

As predicted, we see a distinct drop between some predicted values and actual values. This is not surprising given that a number of our instructions begin and end with the same instruction. Given instructions are matched to be combined in the order expressed in section 6.5.1, and aload_0_getfield_n is the most common instruction, it does not come as a surprise that both getfield_n_aload_0, getfield_n_invokevirtual_n and getfield_n_invokeinterface_n have suffered because of it. In fact the failure of these three instructions to live up to predictions shows that there is scope for a triple instruction such as aload_0_getfield_n_aload_0, aload_0_getfield_n_invokevirtual_n or aload_0_getfield_n_invokeinterface_n, or all of them. It is safe to assume that the lower the frequency of an instruction shadowed by another, the higher the probably of it being a good candidate for a triple instruction.

Without modifying the system to collect these statistics, this is hard to tell for certain. We notice that dup_invokespecial_n (a highly ranked pair) is almost completely overshadowed by new_dup. A quick search through the code generated by the system[3] suggests that the

[3]As statistics are collected, exact details of every instruction encountered are also output to a file. These are details of byte offsets, bytecodes, instruction names, operands and class and method names.

combination of `new`, `dup` and then `invokevirtual_n` is reasonably frequent. As is the combination `new`, `dup` and then `invokestatic`. In retrospect, the construction of the peephole optimiser in this way is actually more of a help than a hindrance as it provides us with extra information that perhaps would not be present if a more intelligent peephole optimiser had been utilised.

One drawback of our peephole optimiser is that it only records statistics for instructions that have been peephole optimised, not for all instructions. Ideally we would like to analyse the effect of these instructions on the frequency of their component instructions, to derive whether or not they still warrant inclusion in the final instruction set.

The differences between the savings observed here and the the savings seen with Mesa and in Bennett's work are apparent. It is important to remember that the results seen in table 6.7 are comparisons to our already specialised "refined" instruction set, and hence we are seeing an incremental picture rather than an entire overview. Clearly the findings are not quite as dramatic as those found in either papers, but it should be noted that we have started with an instruction set which had far more instructions, meaning that there is less chance of common pairs occurring and hence less likelihood of finding a specialised instruction that makes a significant impact on code size. It should also be noted that our saving of over 20% is reasonable, considering the few numbers of new instructions and limited statistical analysis that was undertaken.

To derive an instruction set solely from the results presented here would be ill advised. Whilst we do demonstrate quite sizeable savings, not all can be proved to be as beneficial as they could be. Clearly there is much more work to be done deriving a new Java virtual machine instruction set, and where possible, these avenues have been identified. Our research has been slightly simplistic, not considering how code is generated, the history of the instruction set or the expected future trends of the language. We do however, observe that the results captured and analysed here are a reasonably good starting point for further investigation, and that our savings show promise for even greater savings if more intelligent methods of optimisation are used.

Chapter 7

Conclusions

7.1 Evaluation against requirements

To ascertain whether the performance of the system developed lives up to what was expected of it, it is necessary to review our requirements to see if, and at what level our requirements have been met.

7.1.1 Functional requirements

Data

The data used with the system, in the author's view was adequate. The reason why some instructions were unused was discussed in section 6.2, but we can still conclude that even if they were used, the chances of them being frequent enough to warrant any kind of optimisation are slim. The only other explanation is that we have missed some major application that extensively uses these instructions. Although, considering the variety of our chosen test data, the chances of that happening are also slim. Indeed, we mention the necessity of some of the unused instructions but should also consider that two of the instructions (`dup2_x2` and `swap`) can be expressed by other instructions, and considering that they occur so infrequently, should be removed.

Whether enough instruction data was used is debatable. Certainly we use more than was used by Sweet & James G. Sandman (1982), and we observed similar trends even when half the data set was used. This tends to suggest that we have used more than enough instruction data for analysis.

As required, data from the Java API was not considered. The method chosen to harvest our JAR files for statistics did not need to even consider anything outside of the JAR files. A simple method of obtaining just the code from the JARs was used. Calls to classes inside the Java API (or to any class) were inconsequential to the collection of instructions, but did give us information about which pairs to consider and which not to consider. Our observation is that a dynamic profiling method would need to consciously exclude classes from the Java API but would need to evaluate them anyway, in order to obtain the stream of instructions that would be executed by a Java virtual machine. The question that arises is whether or not excluding these classes is an informed decision. If the class needs to be evaluated anyway, why not include the data in the statistics? After all, there is no guarantee that our test data is compiled by the same compiler or even hand written.

For our implementation, a dynamic profiling method was avoided due to the unnecessary complications. Either extensions to some virtual machine or an entirely new type of virtual machine that collected dynamic statistics would have had to have been built. Our statistics collected on individual instruction frequency would have been more powerful in this case, and it is true that the results presented in this dissertation could well have been gained with this method. However, what we gain by using dynamic profiling is the ability to dynamically optimise a program — a desirable trait but one that is out of the scope of this dissertation.

Statistics capture

Our statistics capturing process worked mostly as was required of it, although the statistics collector did not specifically know how to expand pseudo specialised instructions. This is done through the XML specification, which has been instrumental in the process of re-designing the instruction set incrementally to suit optimisations suggested through analysis of previous statistics. The nature of the XML specification allowed the code for processing to be written once and then statistics collected easily and simply for a variety of different instruction set designs — something that could not have easily been done if the instruction set was hard coded into the system.

Largely our criteria for statistics to be collected was fulfilled, and is described in detail in chapter 6. One regrettable statistic that should have been recorded in light of other statistics is the triples, and possibly quartets. Development time issues were the only boundary to the implementation of this functionality, but from the outset, this requirement was only of “may” priority.

Statistics analysis

Regrettably and perhaps predictably, no mechanical method of suggesting newly optimised instructions was created. Our two step process of refining the instruction set before considering common pairs proved useful and is possibly a method to eliminate (or at least minimise) the problems that Bennett (1988) observed with some of the instructions generated by ISGEN (discussed in section 6.4), by reducing the amount of deductions that can possibly be made.

7.1.2 Performance requirements

Speed

The speed of the system is overall pretty fast, but was initially let down by one module — the code constructor. The implementation of the code constructor described in section 5.6 is correct, but some optimisation has been performed so that if the instruction has been parsed once before, it does not have to be parsed again. This is made possible by the fact that the XML code constructor simply constructs instruction "shells" (which the system calls templates). Each time a template is created (this happens lazily, once for each instruction), the structure is stored for future use. Every subsequent call for an instruction "shell" invokes a process whereby the template for that particular instruction bytecode is copied. If the instruction with that bytecode has not been templated yet, the XML document is parsed and a template created.

Of course, this method of caching instruction "shells" increases overhead but the saving in speed of execution through not parsing the XML document sequentially every single time is phenomenal and thus justified. To process all our test data, our unoptimised code constructor took approximately 2 hours to complete, and could not be run on the University of Bath Unix machines due to the excessive processor time needed. With "shell" caching, our data is processed in about 10 minutes.

Memory

Admittedly, the amount of memory the program utilised could be reduced somewhat. Firstly, memory allocated for various structures is not always explicitly deallocated when it needs to be. The short development time of the project has meant that edges such as this have not been rounded off as cleanly as they would have been for a development to be

used in a production environment. It was observed that our system allocated in excess of 200MB of memory whilst processing our test data. The author believes almost all of which is due to the lack of deallocation that occurs.

Secondly, our class harvester is relatively inefficient at extracting data from a class file. This is simply because the class file is read into the system in its entirety. In retrospect, we see that our system only deals with code from various methods in the class, and does not need to know details of the data present in, for example, the constant pool of the class. Admittedly this part of the system was developed very early on in the project life cycle when the exact contents that were needed from the class file were not decisively known. The harvester performs everything that is needed of it, but also a bit more.

Clearly our shortfalls in program size are not unsolvable, and the reader is drawn to the fact that the design of our solution — permitting the incremental collection of statistics, would certainly allow our system to operate within smaller memory constraints if the issues described above are solved.

7.1.3 Non-functional requirements

Size

As is evident from section 6.6, we have enabled a code size saving of over 20%. In this respect, we have succeeded in fulfilling this requirement. What is evident from our analysis is that the methods used have been only weakly based on those from previous work. Our requirements state that instructions should be suggested based on some computed gain on the overall instruction set. Clearly a formally defined method of how this should be done really needs to be described. Analysis similar to that seen in chapter 6 can only go so far with suggesting instructions that enable significant savings based on simple observation of statistics. A mechanised solution could consider a vast number of possibilities where saving may be made and would select the optimal sub set — almost certainly superseding the savings seen here. One problem that can be identified with this is the problem of more specialised instructions affecting the frequency distribution of the instruction set. New instructions added in an incremental way may affect the significance of other specialised instructions. This causes undesirable complexity through the fact that each specialised instruction should be careful not to make other specialised instructions redundant. Both Sweet & James G. Sandman (1982) and Bennett (1988) describe occurrences of this and we saw this trait when analysing some of our pair specialisations.

Semantics

The two processes that could have changed the semantics of the program were normalisation and suggestion of more specialised instructions. In both, new instructions were specified and described as fully as possible. Actual semantic content of instructions specialised by operand was never actually changed, but "copied" from the generic form and altered to accommodate a particular value. Where generic forms did not exist (for example `iconst_0`), the generic forms were created so that other values could still be expressed. In this way semantic content is added to some instructions in the instruction set but not withdrawn. In much the same way, semantic content for specialisation of common instruction pairs is, for lack of a better description, the concatenation of "copied" semantic content from individual instructions.

Some instructions that were pseudo specialised effectively had their semantic content split into 2 or more instructions that could be executed sequentially to achieve the same effect. This expansion was seen with the instruction `if_icmpne`, which was expanded into the two instructions `icmpne` and `ifeq`. The semantics of `if_icmpne` are preserved due to the ability of expression through intermediaries.

Execution

The execution requirements were also of "may" priority and unfortunately time did not allow such a development to take place. The fact that a finalised new instruction set was not derived also hindered our ability to test how execution could be affected by our specialised instruction set. As is stated in the requirements, the development of a virtual machine that utilises the new instruction set is the next natural progression. Since our savings have been concerned simply with changes to the instruction set, many of the ideas put forward in chapter 2 concerning optimisation of the actual machine are still applicable.

7.2 Implementation issues

Although a lot of the implementation issues were covered in the previous section, it was noticed that there are a few other areas that need improving. Firstly, the peephole optimiser does not integrate with the system as well as it should do. Internally, the peephole optimiser recognises instances when two instructions can be replaced by one, but does not actually perform the process. Instead the peephole optimiser logs the information about what could

have been done producing separate statistics from the statistics collector. This was done to save time, but after analysis it was apparent that although it provided us with immediate data about the specialised instructions, it did not provide us with the "bigger picture" about the frequency distribution of all instructions.

Secondly, the system's statistics collector can only really deal with instructions of one operand. All instruction operands in Java bytecode are 1 byte long, and although many instructions do have many more than one byte sized operand, they are usually transformed into a single operand with a length of 1 byte, 2 bytes (short) or 4 bytes (integer). For the rest of this discussion, we shall call this transformed operand an argument. The conversion process typically takes a chain of operands and converts them into the appropriate type, based on how many there are. For almost all instructions this process is acceptable, but for instructions with more than one argument, the transformation process does not work correctly, because the string of operands has no indication of which operands belong to which argument. This can be put down to some ambiguity the XML code specification, that does not allow us to express operand groups.

For Java virtual machine instructions, the effect of this is largely inconsequential to the statistics collector. Only 6 Java bytecodes have more than one argument (`iinc`, `invokeinterface`, `lookupswitch`, `multianewarray`, `tableswitch` and `wide`). 5 of which do not occur frequently as either a single instruction or as part of a pair, but the reader may notice that `invokeinterface` was observed to be quite frequent and thus posed a problem. Due to time constraints on the project it was decided not to implement a full method for grouping operands in the XML specification. Instead we made `invokeinterface` a special case and hard coded the system to recognise it. The `invokeinterface` instruction works much like the other `invoke` instructions. It has two extra arguments, a count and a constant value 0, the exact reasons for this are covered in the JVM specification (Lindholm & Yellin, 1999). When specialising this instruction in section 6.4, we specialise `invokeinterface` to `invokeinterface_n`, in much the same way as the other `invoke` instructions — a narrower index. The two extra arguments are ignored (in both specialisation and our output files), but the size of the instruction is still logged correctly.

7.3 Further developments

Although we recognise the benefits from using a peephole optimiser that does not intelligently optimise code, clearly a better peephole optimiser is needed. Ideally the peephole optimiser would not only integrate better with the system as a whole, but would also perform intelligent peephole optimisation on the entire data set (or at least on a per method

basis). Unfortunately this poses a problem concerning the incremental method in which statistics are collected. The program essentially performs two passes over the data already (one to calculate branch targets and one to collect statistics). A third pass would be feasible, but would have an adverse effect on overall performance, although arguably, this does not matter when collecting statistics.

Another functional component that the author would like to include in a future peephole optimiser for the system would be the ability to consider arbitrary numbers of instructions to be concatenated into some specialised instruction. Our research based on commonly paired instructions has shown there is scope for the consideration of triples. It would be favourable for a peephole optimiser to be able to consider these larger numbers as future analysis may uncover scope for specialisation based on quartets or greater numbers of instructions. The prerequisites of this are that the statistics collector will also need to collect information based on triples, quartets etc. so that substitutions can be defined.

The definition of substitutions for the peephole optimiser also needs to have some formal description similar to that of the XML code specification. Currently the instructions and their replacements are hard coded into the system, which is not particularly maintainable. A sketch of an XML specification for the peephole optimiser is shown in code 9. It is a set of `<sequence>` and `<replacement>` tags that describe arbitrary instruction sequences and the instruction that should replace them respectively. The following list shows the properties of this sketch specification:

- `<sequence>` elements are given an `id` attribute that uniquely identifies them as a sequence of instructions that can be specialised.
- `<replacement>` elements are given a `for` attribute that identifies which `<sequence>` element they belong to.
- Instructions within a sequence are given an order to identify the order in which they should occur.
- The `inherit` attribute for replacement instructions is redefined to mean the corresponding instruction (identified by the `bytecode` attribute) in the sequence from which data should be inherited.
- Inherited `operands`, `prestacks` or `poststacks` are matched by order.
- Operands that are present in the instruction stream, but that have been assimilated by the specialised instruction are not given an `inherit` attribute so that they are discarded but still skipped over in the instruction stream.

Code 9 Sketch of peephole XML specification

```
<sequence id="1">
  <instr bytecode="138" order="0">
    <f1>
      <operand type="ubyte" order="0"/>
      <operand type="ubyte" order="1"/>
      <prestack type="ref" order="0"/>
      <poststack type="ref" order="0"/>
    </f1>
  </instr>
  <instr bytecode="139" order="1">
    <f1>
      <poststack type="ref" order="0"/>
    </f1>
  </instr>
</sequence>
<replacement for="1">
  <instr bytecode="25" name="foobar">
    <f1>
      <operand type="ubyte" order="0" inherit="138"/>
      <operand type="ubyte" order="1"/>
      <prestack type="ref" order="0" inherit="138"/>
      <poststack type="ref" order="0" inherit="138"/>
      <poststack type="ref" order="0" inherit="139"/>
    </f1>
  </instr>
</replacement>
```

Despite these further and other developments covered in this chapter, the author believes that the system is in essence a well developed and thought out piece of software. It would be interesting to refine and polish the system to some state that allowed it to be reusable with other byte coded languages. Certainly, the foundations laid here will make that possible.

7.4 Final thoughts

Through argument, or statistical evidence, this dissertation has shown a number of properties of the Java virtual machine instruction set and how it is used, these conclusions are presented below:

1. The existing instruction set is large, RISC centred and that instructions contain little semantic content.

2. Existing pseudo specialisations to the instruction set are largely ineffective in encoding

compact Java applications.

3. Language use in Java, in terms of the types of instructions and their relative frequencies, is very similar to previous work relating to analysis of instruction sets.

4. Application of formal processes from previous work can create significant savings in total application size.

5. Incrementally specialising instruction sets by operand value before common instruction pairs makes the process of selecting instructions for concatenation easier. If a mechanical method for this was to be used, it would also limit numbers of specialised instructions that are never used.

6. Through analysis of popular instruction pairs, there is scope for consideration of triples or quartets.

7. Our changes to the Java virtual machine instruction set have shown an over 20% saving in the size of instruction bytes collected.

In addition, there are a number of items that we would like to be able to conclude on, but cannot, due to restrictions on the project such as scope and time:

1. A dynamic profiling method would be beneficial in knowing additional areas where a CSIC style instruction set would lend itself. Our method is predominantly concerned with static size, but we could go further to make specialised suggestions to instructions that are executed frequently, because of loops or method calls. Clearly it would be beneficial to specialise instructions that are found to be executed frequently, to optimise the dynamic size of the program.

2. We would ideally like to know the frequency distribution of the final instruction set presented in this dissertation. As mentioned previously, this would enable us to ascertain whether or not our optimisations have been effective, and remained effective through further specialisation.

3. Through analysis of common instruction pairs, we can conclude that there is scope for consideration of common instruction triples or quartets. It would be favourable to extend our peephole optimiser to allow this to happen. This does not appear to be a lot of work, considering the current state of the peephole optimiser. However if the method described in section 7.3 is to be used, this time will obviously increase.

The implementation of statically and dynamically optimised virtual machines for resource constrained devices does not appear easy without trade offs between functionality, performance and memory use. Java virtual machines that interpret Java bytecode appear to be the answer, but are inherently slow. This dissertation has shown that even through simplistic methodology's, significant savings in the size of byte coded instruction data can be achieved. This reduction in code size has obvious benefits for bandwidth and download speed and the rich semantic content of instructions should facilitate faster executing times. The author's personal thought is that the statistics presented here are significant proof of the usage of instructions within the Java virtual machine. It is astonishing to find the level of ineffectiveness portrayed by most of the pseudo optimised instructions in the Java virtual machine instruction set. This very observation was conclusive enough to suggest further optimisations were possible and the fact that our statistics have allowed us to make the savings that we did implies at least some correct methodical direction has been found.

Bibliography

Aït-Kaci, H. (1991), *Warren's abstract machine: a tutorial reconstruction*, MIT Press, Cambridge, MA, USA.

Alpern, B., Attanasio, D., Barton, J., Burke, M., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Ngo, T., Mergen, M., Sarkar, V., Serrano, M., Shepherd, J., Smith, S., Sreedhar, V. C., Srinivasan, H. & Whaley, J. (2000), 'The Jalapeño virtual machine', *IBM Systems Journal, Java Performance Issue*.

Beatty, A., Casey, K., Gregg, D. & Nisbet, A. (2003), An optimized java interpreter for connected devices and embedded systems, *in* 'Proceedings of the 2nd international conference on Principles and practice of programming in Java', ACM Press, pp. 692–697.

Bell, J. R. (1973), 'Threaded code', *Commun. ACM* **16**(6), 370–372.

Bennett, J. (1988), A methodology for automated design of computer instruction sets, Technical report, University of Cambridge.

Bush, B., Simon, D. & Taivalsaari, A. (1999), The spotless system: Implementing a Java(TM) system for the palm connected organizer, Technical report, Sun Microsystems, Inc.

Cousot, R. & Cousot, P. (1977), Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *in* 'Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages', ACM Press, pp. 238–252.

Ertl, M. A. & Gregg, D. (2001), The behaviour of efficient virtual machine interpreters on modern architectures, *in* 'Europar 2001 European conference on parallel computing', pp. 403–413.

Hannan, J. (1994), Operational semantics-directed compilers and machine architectures, *in* 'ACM Transactions on Programming Languages and Systems (TOPLAS)', ACM Press, pp. 1215–1247.

Hoogerbrugge, J., Augusteijn, L., Trum, J. & van de Wiel, R. (1999), 'A code compression system based on pipelined interpreters', *Software Practice and Experience* **29**(11), 1005–1023.

Jones, N. D., Gomard, C. K. & Sestoft, P. (1993), *Partial Evaluation and Automatic Program Generation (Prentice Hall International Series in Computer Science)*, Prentice Hall.

Jørring, U. & Scherlis, W. L. (1986), Compilers and staging transformations, *in* 'Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages', ACM Press, pp. 86–96.

Kaffe (1998), 'Kaffe — an open source java virtual machine'. http://www.kaffe.org [Accessed 05 December 2005].

Lindholm, T. & Yellin, F. (1999), *The Java(TM) Virtual Machine Specification (second addition)*, Addison-Wesley.

Miranda, E. (1987), Brouhaha- a portable smalltalk interpreter, *in* 'OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications', ACM Press, New York, NY, USA, pp. 354–365.

Padget, J. (n.d.), Staging the mechanical evaluation of expressions, University of Bath.

Piumarta, I. & Riccardi, F. (1998), Optimizing direct threaded code by selective inlining, *in* 'PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation', ACM Press, New York, NY, USA, pp. 291–300.

Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H. & Nakatani, T. (2005), Design and evaluation of dynamic optimizations for a java just-in-time compiler, *in* 'ACM Transactions on Programming Languages and Systems (TOPLAS)', ACM Press.

Sun Microsystems Inc. (2002), The java hotspot virtual machine, Technical report. Currently available via: http://java.sun.com/products/hotspot/.

Sun Microsystems Inc. (2005), The K virtual machine (KVM), Technical report. Currently available via: http://java.sun.com/products/cldc/wp/.

Sun Microsystems Inc. (2006), 'Existing changes, clarifications and amendments to the JVM specification (second addition)'. http://java.sun.com/docs/books/vmspec/2nd-edition/jvms-maintenance.html.

Sweet, R. E. & James G. Sandman, J. (1982), Empirical analysis of the mesa instruction set, *in* 'Proceedings of the first international symposium on Architectural support for programming languages and operating systems', ACM Press, pp. 158–166.

World Wide Web Consortium - W3C (2004), 'Extensible markup language (XML) 1.0 (third edition)'. http://www.w3.org/TR/2004/REC-xml-20040204/ [Accessed 03 April 2006].

Zhang, L. & Krintz, C. (2005), The design, implementation, and evaluation of adaptive code unloading for resource-constrained devices, *in* 'ACM Transactions on Architecture and Code Optimization (TACO)', ACM Press, pp. 131–164.

Appendix A

User manual

A.1 Hardware requirements

None that are known of, but the system as it is at the moment is quite processor hungry. There should be no reason why the system cannot be compiled and run on both 32 and 64 bit architectures. Indeed the author ran all results seen in the report from the compiled system running on a 64 bit architecture.

The disk space needed varies depending on the amount of data being considered. Currently (for debugging purposes) the statistics collector dumps out information on each instruction it sees, including byte offsets, bytecodes, instruction names, operands and class and method names. For the amount of data used in the project this results in a code file of about 100MB. The actual statistics files generated are commonly less than 1MB in comparison.

A.2 Software requirements

The software requirements for the project are as follows:

1. A C/C++ compiler. The system is written in C but the Minizip library is written in C++.
2. Standard C libraries.
3. The zlib compression/decompression library
4. The Minizip compression/decompression library (included with the project source)

5. The Expat XML parser library.

Many Linux systems come with most of these libraries already installed (or readily available).

A.3 Building the project

On the Bath University Unix machines the Expat library is not installed such that it is in any default directories the C compiler looks for header files in. The Makefile included in the project source adds the relevant directory, however you may need to add `/opt/packages/expat/1.95.8/libs` to the `LD_LIBRARY_PATH` environment variable so the executable can find the library at run-time.

1. Change to the directory you have copied the system source.
2. Execute the Makefile by typing "make"

A.4 Running the JAR reader

1. First of all it is necessary to procure a number of JAR files that you require processing. Place them all in the same directory (as the system does not search directories recursively). The system will ask you to put in the path of that directory when the program is run.
2. Secondly an XML specification is needed. Although the system will pass a specification for a different instruction set, it only understands the attribute values as specified in the report. Three example XML specifications are provided with the project source. One for the normalised instruction set, one for the refined instruction set and one last one which was reinstate the `ifnull` bytecode from the refined instruction set after it was found to be relatively useful.
3. Due to time constraints on the project, the peephole optimiser has it's optimisations hard coded. To remove peephole optimisation from the system you will have to remove all occurrences of `setupPeephole()`, `peephole()` and `terminatePeephole()` from the `jrProcessor.c` file and re-compile.
4. To run the system once it is compiled, change to the directory where the system was compiled and type "./jr".

The JAR reader takes certain arguments to enable and disable certain functions (These are mostly the different types of logging available to the user). Executing with the `-h` flag shows a list of all these options.

Usage:
d - enable logging of debug messages
w - enable logging of warning messages
e - enable logging of error messages (default)
n - disable all logging (even info messages)
x - don't expand instructions

The `-x` is probably the only non-trivial flag in the list. It simply means that alternatives within the XML specification are ignored.

For the logging options, Info messages are always logged unless the -n flag is set. Logging of error messages is selected as the default if no options are provided. Log levels are set up such that:

- log level of debug logs: debug, warn, error and info messages.
- log level of warn logs: warn, error and info messages.
- log level of error logs: error and info messages.
- log level of info logs: info messages.
- log level of none logs: no messages.

After the process has completed, four files will be output to the statistics directory and one file in the peephole directory (both directories are asked as input from the user before processing took place). Statistics files are named `code`, `freq.csv`, `successor.csv` and `predecessor.csv`. The `code` file holds instruction about the instructions processed (as described in section A.1), `freq.csv` holds information about single instruction frequencies and `successor.csv` and `predecessor.csv` hold information on popular pairs. The peephole file `peep.csv` holds information collected on the pairs that were peephole optimised during the process.

Appendix B

XML specification files

B.1 XML Code DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT code (instr+)>
<!ATTLIST code
    lang CDATA #REQUIRED
>
<!ELEMENT instr (f1?, f2?, f3?, f4?, f5?, f6?, f7?, f8?, f9?, f10?, f11?, f12?)>
<!ATTLIST instr
    bytecode CDATA #REQUIRED
    name CDATA #REQUIRED
    branch CDATA #IMPLIED
>
<!ELEMENT f1 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f2 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f3 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f4 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f5 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f6 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f7 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f8 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f9 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f10 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f11 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT f12 (operand*, prestack*, poststack*, alt*)>
<!ELEMENT operand (#PCDATA)>
<!ATTLIST operand
    type CDATA #REQUIRED
    order CDATA #IMPLIED
>
<!ELEMENT prestack (#PCDATA)>
<!ATTLIST prestack
```

```
    type CDATA #REQUIRED
    order CDATA #IMPLIED
>
<!ELEMENT poststack (#PCDATA)>
<!ATTLIST poststack
    type CDATA #REQUIRED
    order CDATA #IMPLIED
>
<!ELEMENT alt (operandval*, prestackval*, poststackval*)>
<!ATTLIST alt
    bytecode CDATA #REQUIRED
    form CDATA #IMPLIED
    order CDATA #IMPLIED
>
<!ELEMENT operandval (#PCDATA)>
<!ATTLIST operandval
    order CDATA #IMPLIED
    inherit CDATA #IMPLIED
>
<!ELEMENT prestackval (#PCDATA)>
<!ATTLIST prestackval
    order CDATA #IMPLIED
    inherit CDATA #IMPLIED
>
<!ELEMENT poststackval (#PCDATA)>
<!ATTLIST poststackval
    order CDATA #IMPLIED
    inherit CDATA #IMPLIED
>
```

B.2 Normalised XML Java bytecode specification

Due to the length of the XML bytecode specification files, only the normalised version is presented here. All of the XML specifications used during this project are available on the CD (under the directory `system/XML/`), which was handed in with the dissertation.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "code.dtd">
<!-- Begin Java Bytecodes //-->
<code lang="java">
  <instr bytecode="50" name="aaload">
    <f1>
      <prestack type="ref" order="0"/>
      <prestack type="integer" order="1"/>
      <poststack type="ref"/>
    </f1>
```

```
</instr>
<instr bytecode="83" name="aastore">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <prestack type="ref" order="2"/>
  </f1>
</instr>
<instr bytecode="1" name="aconst_null">
  <f1>
    <poststack type="nullref"/>
  </f1>
</instr>
<instr bytecode="25" name="aload">
  <f1>
    <operand type="ubyte"/>
    <poststack type="ref"/>
  </f1>
</instr>
<instr bytecode="42" name="aload_0">
  <f1>
    <poststack type="ref"/>
    <alt bytecode="25">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="43" name="aload_1">
  <f1>
    <poststack type="ref"/>
    <alt bytecode="25">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="44" name="aload_2">
  <f1>
    <poststack type="ref"/>
    <alt bytecode="25">
      <operandval>2</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="45" name="aload_3">
  <f1>
    <poststack type="ref"/>
    <alt bytecode="25">
      <operandval>3</operandval>
    </alt>
  </f1>
</instr>
```

```
<instr bytecode="189" name="anewarray">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
    <poststack type="ref"/>
  </f1>
</instr>
<instr bytecode="176" name="areturn" branch="always">
  <f1>
    <prestack type="ref"/>
    <poststack type="empty"/>
  </f1>
</instr>
<instr bytecode="190" name="arraylength">
  <f1>
    <prestack type="ref"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="58" name="astore">
  <f1>
    <operand type="ubyte"/>
    <prestack type="ref"/>
  </f1>
  <f2>
    <operand type="ubyte"/>
    <prestack type="ra"/>
  </f2>
</instr>
<instr bytecode="75" name="astore_0">
  <f1>
    <prestack type="ref"/>
    <alt bytecode="58" form="1">
      <operandval>0</operandval>
    </alt>
  </f1>
  <f2>
    <prestack type="ra"/>
    <alt bytecode="58" form="2">
      <operandval>0</operandval>
    </alt>
  </f2>
</instr>
<instr bytecode="76" name="astore_1">
  <f1>
    <prestack type="ref"/>
    <alt bytecode="58" form="1">
      <operandval>1</operandval>
    </alt>
  </f1>
```

```
  <f2>
    <prestack type="ra"/>
    <alt bytecode="58" form="2">
      <operandval>1</operandval>
    </alt>
  </f2>
</instr>
<instr bytecode="77" name="astore_2">
  <f1>
    <prestack type="ref"/>
    <alt bytecode="58" form="1">
      <operandval>2</operandval>
    </alt>
  </f1>
  <f2>
    <prestack type="ra"/>
    <alt bytecode="58" form="2">
      <operandval>2</operandval>
    </alt>
  </f2>
</instr>
<instr bytecode="78" name="astore_3">
  <f1>
    <prestack type="ref"/>
    <alt bytecode="58" form="1">
      <operandval>3</operandval>
    </alt>
  </f1>
  <f2>
    <prestack type="ra"/>
    <alt bytecode="58" form="2">
      <operandval>3</operandval>
    </alt>
  </f2>
</instr>
<instr bytecode="191" name="athrow" branch="always">
  <f1>
    <prestack type="ref"/>
    <poststack type="ref"/>
  </f1>
</instr>
<instr bytecode="51" name="baload">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="84" name="bastore">
  <f1>
    <prestack type="ref" order="0"/>
```

```
    <prestack type="integer" order="1"/>
    <prestack type="integer" order="2"/>
  </f1>
</instr>
<instr bytecode="16" name="bipush">
  <f1>
    <operand type="byte"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="52" name="caload">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="85" name="castore">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <prestack type="integer" order="2"/>
  </f1>
</instr>
<instr bytecode="192" name="checkcast">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref"/>
    <poststack type="ref"/>
  </f1>
</instr>
<instr bytecode="144" name="d2f">
  <f1>
    <prestack type="doubler"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="142" name="d2i">
  <f1>
    <prestack type="doubler"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="143" name="d2l">
  <f1>
    <prestack type="doubler"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="99" name="dadd">
```

```
  <f1>
    <prestack type="doubler"/>
    <prestack type="doubler"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="49" name="daload">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="82" name="dastore">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <prestack type="doubler" order="2"/>
  </f1>
</instr>
<instr bytecode="152" name="dcmpg">
  <f1>
    <prestack type="doubler"/>
    <prestack type="doubler"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="151" name="dcmpl">
  <f1>
    <prestack type="doubler"/>
    <prestack type="doubler"/>
    <poststack type="integer"/>
  </f1>
</instr>
<!-- Decomposed dconst //-->
<instr bytecode="206" name="dconst">
  <f1>
    <operand type="ubtye"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="14" name="dconst_0">
  <f1>
    <poststack type="doubler">0</poststack>
    <alt bytecode="206">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="15" name="dconst_1">
  <f1>
```

```
    <poststack type="doubler">1</poststack>
    <alt bytecode="206">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="111" name="ddiv">
  <f1>
    <prestack type="doubler"/>
    <prestack type="doubler"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="24" name="dload">
  <f1>
    <operand type="ubyte"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="38" name="dload_0">
  <f1>
    <poststack type="doubler"/>
    <alt bytecode="24">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="39" name="dload_1">
  <f1>
    <poststack type="doubler"/>
    <alt bytecode="24">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="40" name="dload_2">
  <f1>
    <poststack type="doubler"/>
    <alt bytecode="24">
      <operandval>2</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="41" name="dload_3">
  <f1>
    <poststack type="doubler"/>
    <alt bytecode="24">
      <operandval>3</operandval>
    </alt>
  </f1>
</instr>
```

```
<instr bytecode="107" name="dmul">
  <f1>
    <prestack type="doubler"/>
    <prestack type="doubler"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="119" name="dneg">
  <f1>
    <prestack type="doubler"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="115" name="drem">
  <f1>
    <prestack type="doubler"/>
    <prestack type="doubler"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="175" name="dreturn" branch="always">
  <f1>
    <prestack type="doubler"/>
    <poststack type="empty"/>
  </f1>
</instr>
<instr bytecode="57" name="dstore">
  <f1>
    <operand type="ubyte"/>
    <prestack type="doubler"/>
  </f1>
</instr>
<instr bytecode="71" name="dstore_0">
  <f1>
    <prestack type="doubler"/>
    <alt bytecode="57">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="72" name="dstore_1">
  <f1>
    <prestack type="doubler"/>
    <alt bytecode="57">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="73" name="dstore_2">
  <f1>
    <prestack type="doubler"/>
```

```
    <alt bytecode="57">
      <operandval>2</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="74" name="dstore_3">
  <f1>
    <prestack type="doubler"/>
    <alt bytecode="57">
      <operandval>3</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="103" name="dsub">
  <f1>
    <prestack type="doubler"/>
    <prestack type="doubler"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="89" name="dup">
  <f1>
    <prestack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
  </f1>
</instr>
<instr bytecode="90" name="dup_x1">
  <f1>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
  </f1>
</instr>
<instr bytecode="91" name="dup_x2">
  <f1>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
  </f1>
  <f2>
    <prestack type="cat2" order="0"/>
    <prestack type="cat1" order="1"/>
    <poststack type="cat1" order="0"/>
    <poststack type="cat2" order="1"/>
```

```
    <poststack type="cat1" order="2"/>
  </f2>
</instr>
<instr bytecode="92" name="dup2">
  <f1>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
  </f1>
  <f2>
    <prestack type="cat2"/>
    <poststack type="cat2"/>
    <poststack type="cat2"/>
  </f2>
</instr>
<instr bytecode="93" name="dup2_x1">
  <f1>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
  </f1>
  <f2>
    <prestack type="cat1" order="0"/>
    <prestack type="cat2" order="1"/>
    <poststack type="cat2" order="0"/>
    <poststack type="cat1" order="1"/>
    <poststack type="cat2" order="2"/>
  </f2>
</instr>
<instr bytecode="94" name="dup2_x2">
  <f1>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
    <poststack type="cat1"/>
  </f1>
  <f2>
```

```
    <prestack type="cat1" order="0"/>
    <prestack type="cat1" order="1"/>
    <prestack type="cat2" order="2"/>
    <poststack type="cat2" order="0"/>
    <poststack type="cat1" order="1"/>
    <poststack type="cat1" order="2"/>
    <poststack type="cat2" order="3"/>
  </f2>
  <f3>
    <prestack type="cat2" order="0"/>
    <prestack type="cat1" order="1"/>
    <prestack type="cat1" order="2"/>
    <poststack type="cat1" order="0"/>
    <poststack type="cat1" order="1"/>
    <poststack type="cat2" order="2"/>
    <poststack type="cat1" order="3"/>
    <poststack type="cat1" order="4"/>
  </f3>
  <f4>
    <prestack type="cat2" order="0"/>
    <prestack type="cat2" order="1"/>
    <poststack type="cat2" order="0"/>
    <poststack type="cat2" order="1"/>
    <poststack type="cat2" order="2"/>
  </f4>
</instr>
<instr bytecode="141" name="f2d">
  <f1>
    <prestack type="floater"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="139" name="f2i">
  <f1>
    <prestack type="floater"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="140" name="f2l">
  <f1>
    <prestack type="floater"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="98" name="fadd">
  <f1>
    <prestack type="floater"/>
    <prestack type="floater"/>
    <poststack type="floater"/>
  </f1>
</instr>
```

```
<instr bytecode="48" name="faload">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="81" name="fastore">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <prestack type="floater" order="2"/>
  </f1>
</instr>
<instr bytecode="150" name="fcmpg">
  <f1>
    <prestack type="floater"/>
    <prestack type="floater"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="149" name="fcmpl">
  <f1>
    <prestack type="floater"/>
    <prestack type="floater"/>
    <poststack type="integer"/>
  </f1>
</instr>
<!-- Decomposed fconst //-->
<instr bytecode="205" name="fconst">
  <f1>
    <operand type="ubtye"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="11" name="fconst_0">
  <f1>
    <poststack type="floater">0</poststack>
    <alt bytecode="205">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="12" name="fconst_1">
  <f1>
    <poststack type="floater">1</poststack>
    <alt bytecode="205">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
```

```
<instr bytecode="13" name="fconst_2">
  <f1>
    <poststack type="floater">2</poststack>
    <alt bytecode="205">
      <operandval>2</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="110" name="fdiv">
  <f1>
    <prestack type="floater"/>
    <prestack type="floater"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="23" name="fload">
  <f1>
    <operand type="ubyte"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="34" name="fload_0">
  <f1>
    <poststack type="floater"/>
    <alt bytecode="23">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="35" name="fload_1">
  <f1>
    <poststack type="floater"/>
    <alt bytecode="23">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="36" name="fload_2">
  <f1>
    <poststack type="floater"/>
    <alt bytecode="23">
      <operandval>2</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="37" name="fload_3">
  <f1>
    <poststack type="floater"/>
    <alt bytecode="23">
      <operandval>3</operandval>
    </alt>
```

```
  </f1>
</instr>
<instr bytecode="106" name="fmul">
  <f1>
    <prestack type="floater"/>
    <prestack type="floater"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="118" name="fneg">
  <f1>
    <prestack type="floater"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="114" name="frem">
  <f1>
    <prestack type="floater"/>
    <prestack type="floater"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="174" name="freturn" branch="always">
  <f1>
    <prestack type="floater"/>
    <poststack type="empty"/>
  </f1>
</instr>
<instr bytecode="56" name="fstore">
  <f1>
    <operand type="ubyte"/>
    <prestack type="floater"/>
  </f1>
</instr>
<instr bytecode="67" name="fstore_0">
  <f1>
    <prestack type="floater"/>
    <alt bytecode="56">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="68" name="fstore_1">
  <f1>
    <prestack type="floater"/>
    <alt bytecode="56">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="69" name="fstore_2">
```

```
  <f1>
    <prestack type="floater"/>
    <alt bytecode="56">
      <operandval>2</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="70" name="fstore_3">
  <f1>
    <prestack type="floater"/>
    <alt bytecode="56">
      <operandval>3</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="102" name="fsub">
  <f1>
    <prestack type="floater"/>
    <prestack type="floater"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="180" name="getfield">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref"/>
    <poststack type="ref"/>
  </f1>
</instr>
<instr bytecode="178" name="getstatic">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <poststack type="classvar"/>
  </f1>
</instr>
<instr bytecode="167" name="goto" branch="always">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
  </f1>
</instr>
<instr bytecode="200" name="goto_w" branch="always">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
  </f1>
</instr>
```

```
<instr bytecode="145" name="i2b">
  <f1>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="146" name="i2c">
  <f1>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="135" name="i2d">
  <f1>
    <prestack type="integer"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="134" name="i2f">
  <f1>
    <prestack type="integer"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="133" name="i2l">
  <f1>
    <prestack type="integer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="147" name="i2s">
  <f1>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="96" name="iadd">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="46" name="iaload">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="126" name="iand">
```

```
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="79" name="iastore">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <prestack type="integer" order="2"/>
  </f1>
</instr>
<!-- decomposed iconst //-->
<instr bytecode="203" name="iconst">
  <f1>
    <operand type="ubyte"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="2" name="iconst_m1">
  <f1>
    <poststack type="integer">-1</poststack>
    <alt bytecode="203">
      <operandval>-1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="3" name="iconst_0">
  <f1>
    <poststack type="integer">0</poststack>
    <alt bytecode="203">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="4" name="iconst_1">
  <f1>
    <poststack type="integer">1</poststack>
    <alt bytecode="203">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="5" name="iconst_2">
  <f1>
    <poststack type="integer">2</poststack>
    <alt bytecode="203">
      <operandval>2</operandval>
    </alt>
  </f1>
```

```
</instr>
<instr bytecode="6" name="iconst_3">
  <f1>
    <poststack type="integer">3</poststack>
    <alt bytecode="203">
      <operandval>3</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="7" name="iconst_4">
  <f1>
    <poststack type="integer">4</poststack>
    <alt bytecode="203">
      <operandval>4</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="8" name="iconst_5">
  <f1>
    <poststack type="integer">5</poststack>
    <alt bytecode="203">
      <operandval>5</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="108" name="idiv">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<!-- Start decomposed if instructions //-->
<instr bytecode="207" name="icmpeq">
<f1>
  <prestack type="integer"/>
  <prestack type="integer"/>
  <poststack type="integer"/>
</f1>
</instr>
<instr bytecode="208" name="icmpne">
<f1>
  <prestack type="integer"/>
  <prestack type="integer"/>
  <poststack type="integer"/>
</f1>
</instr>
<instr bytecode="209" name="icmplt">
<f1>
  <prestack type="integer"/>
  <prestack type="integer"/>
```

```
  <poststack type="integer"/>
</f1>
</instr>
<instr bytecode="210" name="icmpge">
<f1>
  <prestack type="integer"/>
  <prestack type="integer"/>
  <poststack type="integer"/>
</f1>
</instr>
<instr bytecode="211" name="icmpgt">
<f1>
  <prestack type="integer"/>
  <prestack type="integer"/>
  <poststack type="integer"/>
</f1>
</instr>
<instr bytecode="212" name="icmple">
<f1>
  <prestack type="integer"/>
  <prestack type="integer"/>
  <poststack type="integer"/>
</f1>
</instr>
<instr bytecode="213" name="acmpeq">
  <f1>
    <prestack type="ref"/>
    <prestack type="ref"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="214" name="acmpne">
  <f1>
    <prestack type="ref"/>
    <prestack type="ref"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="215" name="isnull">
  <f1>
    <prestack type="ref"/>
    <poststack type="integer"/>
  </f1>
</instr>
<!-- End decomposed if instructions //-->
<instr bytecode="165" name="if_acmpeq" branch="eq">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref"/>
    <prestack type="ref"/>
```

```
    <!-- acmpeq then ifeq //-->
    <alt bytecode="213" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
<instr bytecode="166" name="if_acmpne" branch="ne">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref"/>
    <prestack type="ref"/>
    <alt bytecode="214" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
<instr bytecode="159" name="if_icmpeq" branch="eq">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <alt bytecode="207" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
<instr bytecode="160" name="if_icmpne" branch="ne">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <alt bytecode="208" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
<instr bytecode="161" name="if_icmplt" branch="lt">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
```

```
    <prestack type="integer"/>
    <prestack type="integer"/>
    <alt bytecode="209" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
<instr bytecode="162" name="if_icmpge" branch="ge">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <alt bytecode="210" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
<instr bytecode="163" name="if_icmpgt" branch="gt">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <alt bytecode="211" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
<instr bytecode="164" name="if_icmple" branch="le">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <alt bytecode="212" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
<instr bytecode="153" name="ifeq" branch="eq">
  <f1>
    <operand type="ubyte"/>
```

```
    <operand type="ubyte"/>
    <prestack type="integer"/>
  </f1>
</instr>
<instr bytecode="154" name="ifne" branch="ne">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
  </f1>
</instr>
<instr bytecode="155" name="iflt" branch="lt">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
  </f1>
</instr>
<instr bytecode="156" name="ifge" branch="ge">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
  </f1>
</instr>
<instr bytecode="157" name="ifgt" branch="gt">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
  </f1>
</instr>
<instr bytecode="158" name="ifle" branch="le">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="integer"/>
  </f1>
</instr>
<instr bytecode="199" name="ifnonnull" branch="ne">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref"/>
    <alt bytecode="215" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
```

```
<instr bytecode="198" name="ifnull" branch="eq">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref"/>
    <alt bytecode="215" order="0"/>
    <alt bytecode="153" order="1">
      <operandval inherit="true"/>
      <operandval inherit="true"/>
    </alt>
  </f1>
</instr>
<instr bytecode="132" name="iinc">
  <f1>
    <operand type="ubyte" order="0"/>
    <operand type="byte" order="1"/>
  </f1>
</instr>
<instr bytecode="21" name="iload">
  <f1>
    <operand type="ubyte"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="26" name="iload_0">
  <f1>
    <poststack type="integer"/>
    <alt bytecode="21">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="27" name="iload_1">
  <f1>
    <poststack type="integer"/>
    <alt bytecode="21">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="28" name="iload_2">
  <f1>
    <poststack type="integer"/>
    <alt bytecode="21">
      <operandval>2</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="29" name="iload_3">
  <f1>
    <poststack type="integer"/>
```

```
    <alt bytecode="21">
      <operandval>3</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="104" name="imul">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="116" name="ineg">
  <f1>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="193" name="instanceof">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="185" name="invokeinterface" branch="always">
  <f1>
    <operand type="ubyte" order="0"/>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
    <operand type="byte" order="3">0</operand>
    <prestack type="ref" order="0"/>
    <prestack type="nargs" order="1"/>
  </f1>
</instr>
<instr bytecode="183" name="invokespecial" branch="always">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref" order="0"/>
    <prestack type="nargs" order="1"/>
  </f1>
</instr>
<instr bytecode="184" name="invokestatic" branch="always">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="nargs"/>
  </f1>
</instr>
```

```
<instr bytecode="182" name="invokevirtual" branch="always">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref" order="0"/>
    <prestack type="nargs" order="1"/>
  </f1>
</instr>
<instr bytecode="128" name="ior">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="112" name="irem">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="172" name="ireturn" branch="always">
  <f1>
    <prestack type="integer"/>
    <poststack type="empty"/>
  </f1>
</instr>
<instr bytecode="120" name="ishl">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="122" name="ishr">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="54" name="istore">
  <f1>
    <operand type="ubyte"/>
    <prestack type="integer"/>
  </f1>
</instr>
<instr bytecode="59" name="istore_0">
  <f1>
    <prestack type="integer"/>
```

```
    <alt bytecode="54">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="60" name="istore_1">
  <f1>
    <prestack type="integer"/>
    <alt bytecode="54">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="61" name="istore_2">
  <f1>
    <prestack type="integer"/>
    <alt bytecode="54">
      <operandval>2</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="62" name="istore_3">
  <f1>
    <prestack type="integer"/>
    <alt bytecode="54">
      <operandval>3</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="100" name="isub">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="124" name="iushr">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="130" name="ixor">
  <f1>
    <prestack type="integer"/>
    <prestack type="integer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="168" name="jsr" branch="always">
```

```
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <poststack type="ra"/>
  </f1>
</instr>
<instr bytecode="201" name="jsr_w" branch="always">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <poststack type="ra"/>
  </f1>
</instr>
<instr bytecode="138" name="l2d">
  <f1>
    <prestack type="longer"/>
    <poststack type="doubler"/>
  </f1>
</instr>
<instr bytecode="137" name="l2f">
  <f1>
    <prestack type="longer"/>
    <poststack type="floater"/>
  </f1>
</instr>
<instr bytecode="136" name="l2i">
  <f1>
    <prestack type="longer"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="97" name="ladd">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="47" name="laload">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="127" name="land">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
```

```
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="80" name="lastore">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <prestack type="longer" order="2"/>
  </f1>
</instr>
<instr bytecode="148" name="lcmp">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <prestack type="integer"/>
  </f1>
</instr>
<!-- Decomposed lconst //-->
<instr bytecode="204" name="lconst">
  <f1>
    <operand type="ubtye"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="9" name="lconst_0">
  <f1>
    <poststack type="longer">0</poststack>
    <alt bytecode="204">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="10" name="lconst_1">
  <f1>
    <poststack type="longer">1</poststack>
    <alt bytecode="204">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="18" name="ldc">
  <f1>
    <operand type="ubyte"/>
    <poststack type="floater"/>
  </f1>
  <f2>
    <operand type="ubyte"/>
    <poststack type="integer"/>
  </f2>
  <f3>
    <operand type="ubyte"/>
```

```
    <poststack type="ref"/>
  </f3>
</instr>
<instr bytecode="19" name="ldc_w">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <poststack type="floater"/>
  </f1>
  <f2>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <poststack type="integer"/>
  </f2>
  <f3>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <poststack type="ref"/>
  </f3>
</instr>
<instr bytecode="20" name="ldc2_w">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <poststack type="cat2"/>
  </f1>
</instr>
<instr bytecode="109" name="ldiv">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="22" name="lload">
  <f1>
    <operand type="ubyte"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="30" name="lload_0">
  <f1>
    <poststack type="longer"/>
    <alt bytecode="22">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="31" name="lload_1">
  <f1>
    <poststack type="longer"/>
```

```
    <alt bytecode="22">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="32" name="lload_2">
  <f1>
    <poststack type="longer"/>
    <alt bytecode="22">
      <operandval>2</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="33" name="lload_3">
  <f1>
    <poststack type="longer"/>
    <alt bytecode="22">
      <operandval>3</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="105" name="lmul">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="117" name="lneg">
  <f1>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="171" name="lookupswitch" branch="always">
  <f1>
    <operand type="byte" order="0">0</operand>
    <operand type="byte" order="1">0</operand>
    <operand type="byte" order="2">0</operand>
    <operand type="ubyte" order="3"/>
    <operand type="ubyte" order="4"/>
    <operand type="ubyte" order="5"/>
    <operand type="ubyte" order="6"/>
    <operand type="ubyte" order="7"/>
    <operand type="ubyte" order="8"/>
    <operand type="ubyte" order="9"/>
    <operand type="ubyte" order="10"/>
    <operand type="nxubyte" order="11"/>
    <prestack type="integer"/>
  </f1>
  <f2>
```

```
    <operand type="byte" order="0">0</operand>
    <operand type="byte" order="1">0</operand>
    <operand type="ubyte" order="2"/>
    <operand type="ubyte" order="3"/>
    <operand type="ubyte" order="4"/>
    <operand type="ubyte" order="5"/>
    <operand type="ubyte" order="6"/>
    <operand type="ubyte" order="7"/>
    <operand type="ubyte" order="8"/>
    <operand type="ubyte" order="9"/>
    <operand type="nxubyte" order="10"/>
    <prestack type="integer"/>
  </f2>
  <f3>
    <operand type="byte" order="0">0</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
    <operand type="ubyte" order="3"/>
    <operand type="ubyte" order="4"/>
    <operand type="ubyte" order="5"/>
    <operand type="ubyte" order="6"/>
    <operand type="ubyte" order="7"/>
    <operand type="ubyte" order="8"/>
    <operand type="nxubyte" order="9"/>
    <prestack type="integer"/>
  </f3>
  <f4>
    <operand type="ubyte" order="0"/>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
    <operand type="ubyte" order="3"/>
    <operand type="ubyte" order="4"/>
    <operand type="ubyte" order="5"/>
    <operand type="ubyte" order="6"/>
    <operand type="ubyte" order="7"/>
    <operand type="nxubyte" order="8"/>
    <prestack type="integer"/>
  </f4>
</instr>
<instr bytecode="129" name="lor">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="113" name="lrem">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
```

```
  </f1>
</instr>
<instr bytecode="173" name="lreturn" branch="always">
  <f1>
    <prestack type="longer"/>
    <poststack type="empty"/>
  </f1>
</instr>
<instr bytecode="121" name="lshl">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="123" name="lshr">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="55" name="lstore">
  <f1>
    <operand type="ubyte"/>
    <prestack type="longer"/>
  </f1>
</instr>
<instr bytecode="63" name="lstore_0">
  <f1>
    <prestack type="longer"/>
    <alt bytecode="55">
      <operandval>0</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="64" name="lstore_1">
  <f1>
    <prestack type="longer"/>
    <alt bytecode="55">
      <operandval>1</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="65" name="lstore_2">
  <f1>
    <prestack type="longer"/>
    <alt bytecode="55">
      <operandval>2</operandval>
    </alt>
  </f1>
```

```
</instr>
<instr bytecode="66" name="lstore_3">
  <f1>
    <prestack type="longer"/>
    <alt bytecode="55">
      <operandval>3</operandval>
    </alt>
  </f1>
</instr>
<instr bytecode="101" name="lsub">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="125" name="lushr">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="131" name="lxor">
  <f1>
    <prestack type="longer"/>
    <prestack type="longer"/>
    <poststack type="longer"/>
  </f1>
</instr>
<instr bytecode="194" name="monitorenter">
  <f1>
    <prestack type="ref"/>
  </f1>
</instr>
<instr bytecode="195" name="monitorexit">
  <f1>
    <prestack type="ref"/>
  </f1>
</instr>
<instr bytecode="197" name="multianewarray">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="nxint"/>
    <poststack type="ref"/>
  </f1>
</instr>
<instr bytecode="187" name="new">
  <f1>
```

```
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <poststack type="ref"/>
  </f1>
</instr>
<instr bytecode="188" name="newarray">
  <f1>
    <operand type="ubyte"/>
    <prestack type="integer"/>
    <poststack type="ref"/>
  </f1>
</instr>
<instr bytecode="0" name="nop"/>
<instr bytecode="87" name="pop">
  <f1>
    <prestack type="cat1"/>
  </f1>
</instr>
<instr bytecode="88" name="pop2">
  <f1>
    <prestack type="cat1"/>
    <prestack type="cat1"/>
  </f1>
  <f2>
    <prestack type="cat2"/>
  </f2>
</instr>
<instr bytecode="181" name="putfield">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <prestack type="ref" order="0"/>
    <prestack type="cpool" order="1"/>
  </f1>
</instr>
<instr bytecode="179" name="putstatic">
  <f1>
    <operand type="ubyte" />
    <operand type="ubyte" />
    <prestack type="cpool" />
  </f1>
</instr>
<instr bytecode="169" name="ret" branch="always">
  <f1>
    <operand type="ubyte"/>
  </f1>
</instr>
<instr bytecode="177" name="return" branch="always">
  <f1>
    <poststack type="empty"/>
  </f1>
```

```
</instr>
<instr bytecode="53" name="saload">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="86" name="sastore">
  <f1>
    <prestack type="ref" order="0"/>
    <prestack type="integer" order="1"/>
    <prestack type="integer" order="2"/>
  </f1>
</instr>
<instr bytecode="17" name="sipush">
  <f1>
    <operand type="ubyte"/>
    <operand type="ubyte"/>
    <poststack type="integer"/>
  </f1>
</instr>
<instr bytecode="95" name="swap">
  <f1>
    <prestack type="cat1" order="0"/>
    <prestack type="cat1" order="1"/>
    <poststack type="cat1" order="0"/>
    <poststack type="cat1" order="1"/>
  </f1>
</instr>
<instr bytecode="170" name="tableswitch" branch="always">
  <f1>
    <operand type="ubyte" order="0">0</operand>
    <operand type="ubyte" order="1">0</operand>
    <operand type="ubyte" order="2">0</operand>
    <operand type="ubyte" order="3"/>
    <operand type="ubyte" order="4"/>
    <operand type="ubyte" order="5"/>
    <operand type="ubyte" order="6"/>
    <operand type="ubyte" order="7"/>
    <operand type="ubyte" order="8"/>
    <operand type="ubyte" order="9"/>
    <operand type="ubyte" order="10"/>
    <operand type="ubyte" order="11"/>
    <operand type="ubyte" order="12"/>
    <operand type="ubyte" order="13"/>
    <operand type="ubyte" order="14"/>
    <operand type="nxubyte" order="15"/>
    <prestack type="integer"/>
  </f1>
  <f2>
```

```
    <operand type="ubyte" order="0">0</operand>
    <operand type="ubyte" order="1">0</operand>
    <operand type="ubyte" order="2"/>
    <operand type="ubyte" order="3"/>
    <operand type="ubyte" order="4"/>
    <operand type="ubyte" order="5"/>
    <operand type="ubyte" order="6"/>
    <operand type="ubyte" order="7"/>
    <operand type="ubyte" order="8"/>
    <operand type="ubyte" order="9"/>
    <operand type="ubyte" order="10"/>
    <operand type="ubyte" order="11"/>
    <operand type="ubyte" order="12"/>
    <operand type="ubyte" order="13"/>
    <operand type="nxubyte" order="14"/>
    <prestack type="integer"/>
  </f2>
  <f3>
    <operand type="ubyte" order="0">0</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
    <operand type="ubyte" order="3"/>
    <operand type="ubyte" order="4"/>
    <operand type="ubyte" order="5"/>
    <operand type="ubyte" order="6"/>
    <operand type="ubyte" order="7"/>
    <operand type="ubyte" order="8"/>
    <operand type="ubyte" order="9"/>
    <operand type="ubyte" order="10"/>
    <operand type="ubyte" order="11"/>
    <operand type="ubyte" order="12"/>
    <operand type="nxubyte" order="13"/>
    <prestack type="integer"/>
  </f3>
  <f4>
    <operand type="ubyte" order="0"/>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
    <operand type="ubyte" order="3"/>
    <operand type="ubyte" order="4"/>
    <operand type="ubyte" order="5"/>
    <operand type="ubyte" order="6"/>
    <operand type="ubyte" order="7"/>
    <operand type="ubyte" order="8"/>
    <operand type="ubyte" order="9"/>
    <operand type="ubyte" order="10"/>
    <operand type="ubyte" order="11"/>
    <operand type="nxubyte" order="12"/>
    <prestack type="integer"/>
  </f4>
</instr>
```

```
<instr bytecode="196" name="wide">
  <f1>
    <!-- iload //-->
    <operand type="ubyte" order="0">21</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
  </f1>
  <f2>
    <!-- fload //-->
    <operand type="ubyte" order="0">23</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
  </f2>
  <f3>
    <!-- aload //-->
    <operand type="ubyte" order="0">25</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
  </f3>
  <f4>
    <!-- lload //-->
    <operand type="ubyte" order="0">22</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
  </f4>
  <f5>
    <!-- dload //-->
    <operand type="ubyte" order="0">24</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
  </f5>
  <f6>
    <!-- istore //-->
    <operand type="ubyte" order="0">54</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
  </f6>
  <f7>
    <!-- fstore //-->
    <operand type="ubyte" order="0">56</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
  </f7>
  <f8>
    <!-- astore //-->
    <operand type="ubyte" order="0">58</operand>
    <operand type="ubyte" order="1"/>
    <operand type="ubyte" order="2"/>
  </f8>
  <f9>
    <!-- lstore //-->
```

```
      <operand type="ubyte" order="0">55</operand>
      <operand type="ubyte" order="1"/>
      <operand type="ubyte" order="2"/>
    </f9>
    <f10>
      <!-- dstore //-->
      <operand type="ubyte" order="0">57</operand>
      <operand type="ubyte" order="1"/>
      <operand type="ubyte" order="2"/>
    </f10>
    <f11>
      <!-- ret //-->
      <operand type="ubyte" order="0">169</operand>
      <operand type="ubyte" order="1"/>
      <operand type="ubyte" order="2"/>
    </f11>
    <f12>
      <!-- iinc //-->
      <operand type="ubyte" order="0">132</operand>
      <operand type="ubyte" order="1"/>
      <operand type="ubyte" order="2"/>
      <operand type="ubyte" order="3"/>
      <operand type="ubyte" order="4"/>
    </f12>
  </instr>
</code>
```

Appendix C

Statistics

All of the following statistics as well as others can be found on the CD handed in with the dissertation.

C.1 Normalised instruction set frequency and operand value distribution

The following statistical data is a small sample of the whole frequency distribution for the normalised instruction set. The full version of the file can be found on the CD under the directory `system/results/normalised/`.

C.2 Normalised common instruction pair frequency distribution

C.3 Refined instruction set frequency and operand value distribution

The following statistical data is a small sample of the whole frequency distribution for the refined instruction set. The full version of the file can be found on the CD under the directory `system/results/refined/`.

C.4 Refined common instruction pair frequency distribution

C.5 New instructions frequency distribution

Appendix D

Source code

The following files make up the source code for the entire system:

- `jr.c` — program driver
- `jrUtils.c`, `jrUtils.h` — general utilities
- `jrLogger.c`, `jrLogger.h` — error logger
- `jrProcessor.c`, `jrProcessor.h` — JAR processor
- `jrHarvester.c`, `jrHarvester.h` — class harvester
- `jrXMLCodeConstructor.c`, `jrXMLCodeConstructor.h` — code constructor
- `jrEval.c`, `jrEval.h`, `jrEvalUtils.c` — evaluator and utilities for the evaluator
- `jrCapture.c`, `jrCapture.h` — statistics collector
- `jrPeephole.c`, `jrPeephole.h` — peephole optimiser
- `jrJava.h` — Java class file structures

All source code for the system is not included here due to it's excessive length. It can be found on the CD handed in with the dissertation under the directory `system/`. The following code is a subset of the source for the whole system, showing the workings of interesting components.

D.1 jrProcessor.c

```
#include "jrJava.h"
#include "jrUtils.h"
#include "jrEval.h"
#include "unzip.h"
#include "jrProcessor.h"
#include "jrHarvester.h"
#include <dirent.h>

/* chain of java classes */
jr_class *javaClasses;
jr_class *class;

jr_jar   *currentJar = NULL; /* current file being processed */

int      pStatus = JR_OK; /* process status flag (0 = OK) */

/* jar setup ------------------------------------------------------------- */

int setupProcessor()
{
    char jarPath[JR_MAX_PATH];
    logInfo("Enter JAR file(s) path WITH trailing forward slash:","");
    scanf("%s", &jarPath);

    /* debug dir codes */
    if(jarPath[0]=='u')
    {
        strcpy(jarPath, "/u/cs/2/cs2ajss/final/semester2/cm30082/code/jarReader/javaTest/JARS/");
    }
    else if(jarPath[0]=='h')
    {
        strcpy(jarPath, "/home/alan/final/semester2/cm30082/code/jarReader/javaTest/JARS/");
    }

    pStatus = getJarFiles(jarPath);
    if(pStatus!=JR_OK)
    {
        logError("failed to get file list", "");
        return pStatus;
    }

    /* see if we can open this archive to check if it exists */
    currentJar->jarHandle = unzOpen(currentJar->jarPath);
    if(currentJar->jarHandle != NULL)
    {
        jrGlobalInfo = realloc(jrGlobalInfo, sizeof(unz_global_info));
        /* get global info */
        pStatus = unzGetGlobalInfo(currentJar->jarHandle, jrGlobalInfo);
        if(pStatus != UNZ_OK)
        {
            logError("failed to get global info from ", currentJar->jarPath);
            return pStatus;
        }
    }
    else
    {
        logError(currentJar->jarPath, ": file not found");
        return JR_FILE_NOT_FOUND;
    }

    /* init class file list */
    javaClasses = NULL;

    /* setup the XML parser */
    pStatus = setupCodeConstructor();
    if(pStatus != JR_OK)
    {
        logError("failed to setup code file parser ", "");
        return pStatus;
    }

    /* setup the statictics collector */
    pStatus=setupCapture(jarPath);
    if(pStatus != JR_OK)
    {
        logError("failed to setup statistics collector ", "");
        return pStatus;
    }

     /* setup the peephole optimiser */
    pStatus=setupPeephole();
    if(pStatus != JR_OK)
    {
        logError("failed to setup peephole optimiser ", "");
        return pStatus;
    }

    return JR_OK;
}

/* gets all jar files in a dir and creates internal objects representing them */
int getJarFiles(char *jarPath)
{
    DIR *dir = opendir(jarPath);
```

```
struct dirent *dirEntry;
jr_jar *newJar = NULL;
jr_jar *tmp = NULL;

if(dir == NULL)
{
    logError(jarPath, " : No such directory");
    return JR_FILE_NOT_FOUND;
}

dirEntry = readdir(dir);

while(dirEntry != NULL)
{
    /* check this file is a jar */
    if(strstr(dirEntry->d_name,".jar") != NULL)
    {
        newJar = (jr_jar*)malloc(sizeof(jr_jar));

        if(newJar!=NULL)
        {
            /* copy the path into our buffer */
            strcpy(newJar->jarPath,jarPath);
            if((strlen(newJar->jarPath) + strlen(dirEntry->d_name)) <= JR_MAX_PATH)
            {

                /* concatinate the filename onto the end */
                strcat(newJar->jarPath,dirEntry->d_name);

                /* now link up the current and tmp */
                if(currentJar==NULL)
                {
                    currentJar = newJar;
                }
                else
                {
                    tmp=currentJar;

                    while(tmp->nextJar)
                        tmp=tmp->nextJar;

                    tmp->nextJar = newJar;
                }
            }
            else
            {
                logWarn("discarding file: path longer than max path", "");
                free(newJar);
            }
        }
        else
        {
            closedir(dir);
            logError("out of memory (new jar file)","");
            return JR_OUT_OF_MEMORY;
        }
    }

    /* get the next entry */
    dirEntry = readdir(dir);
}

closedir(dir);

return JR_OK;
}

/* gets next internal JAR file object */
int getNextJarFile()
{
    jr_jar *tmp;

    /* move onto the next jar file */
    tmp = currentJar;
    currentJar = currentJar->nextJar;

    /* clean up the old */
    pStatus = unzClose(tmp->jarHandle);
    if(pStatus != JR_OK)
    {
        logError("failed jar archive close", "");
        return pStatus;
    }
    free(tmp);

    if(currentJar!=NULL)
    {
        /* see if we can open this archive to check if it exists */
        currentJar->jarHandle = unzOpen(currentJar->jarPath);

        if(currentJar->jarHandle != NULL)
        {
            pStatus = unzGoToFirstFile(currentJar->jarHandle);
            if(pStatus!=JR_OK)
            {
                logError("failed to move to start of jar file", "");
                return pStatus;
            }
```

```
            jrGlobalInfo = realloc(jrGlobalInfo, sizeof(unz_global_info));

            /* get global info */
            pStatus = unzGetGlobalInfo(currentJar->jarHandle, jrGlobalInfo);
            if(pStatus != UNZ_OK)
            {
                logError("failed to get global info from ", currentJar->jarPath);
                return pStatus;
            }
        }
        else
        {
            logError(currentJar->jarPath, ": file not found");
            return JR_FILE_NOT_FOUND;
        }
    }
    else
    {
        logWarn("end of JAR file list", "");
    }

    return JR_OK;
}

/* gets internal representation of the next class file in a JAR */
int getNextClassFile(int gotoNext)
{
    int foundClassFile = FALSE;

    /* while we still don't have a class file */
    while(foundClassFile==FALSE)
    {
        /* move to next file */
        if(gotoNext==TRUE)
        {
            pStatus = unzGoToNextFile(currentJar->jarHandle);
            if(pStatus==UNZ_END_OF_LIST_OF_FILE)
                return pStatus;
        }


        pStatus = unzOpenCurrentFile(currentJar->jarHandle);
        if(pStatus != JR_OK)
        {
            logError("failed to open next file","");
            return pStatus;
        }
```



```
/* reallocate memory */
jrFileInfo = (unz_file_info*)realloc(jrFileInfo,
                                     sizeof(unz_file_info));

jrFileName = (char*)realloc(jrFileName,
                            sizeof(char[JR_FILENAMEBUF]));

jrExtraField = (char*)realloc(jrExtraField,
                              sizeof(char[JR_EXTRABUF]));

jrComment = (char*)realloc(jrComment,
                           sizeof(char[JR_COMMENTBUF]));

if(jrFileInfo    == NULL |
   jrFileName    == NULL |
   jrExtraField == NULL |
   jrComment     == NULL)
{
   logError("out of memory (new file properties)", "");
   return JR_OUT_OF_MEMORY;
}

/* get file info */
pStatus = unzGetCurrentFileInfo(currentJar->jarHandle,
                                jrFileInfo,
                                jrFileName,
                                JR_FILENAMEBUF,
                                jrExtraField,
                                JR_EXTRABUF,
                                jrComment,
                                JR_COMMENTBUF);

if(pStatus != UNZ_OK)
{
   logError("failed to get file info from ", jrFileName);
   return pStatus;
}

/* now create internal representation of this class file */
pStatus = jarvest(currentJar->jarHandle, jrFileName);
if(pStatus == JR_OK)
{
   /* found a class file */
   foundClassFile = TRUE;

   /* move to end of class file list */
   if(javaClasses==NULL)
   {
       javaClasses=getHarvestedFile();
```

```
                class=javaClasses;
            }
            else
            {
                class=javaClasses;
                while(class->nextClass)
                    class=class->nextClass;

                /* append harvested class on the end */
                class->nextClass=getHarvestedFile();
                class=class->nextClass;
            }

        }
        else if(pStatus == JR_INVALID_MAJOR ||
                pStatus == JR_INVALID_MINOR ||
                pStatus == JR_INVALID_MAGIC ||
                pStatus == JR_INVALID_EXT)
        {
            logWarn(jrFileName, " is not a valid code file");
            gotoNext = TRUE;
        }
        else
        {
            return pStatus;
        }

        /* close file to ensure CRC ok */
        pStatus = unzCloseCurrentFile(currentJar->jarHandle);
        if(pStatus != JR_OK)
        {
            logError("failed CRC for file ", jrFileName);
            return pStatus;
        }
    }

    return pStatus;
}


/* jar processing ------------------------------------------------------------ */

int process()
{
    int evalStatus;

    /* move to the start of the file (if we're not there already) */
    pStatus = unzGoToFirstFile(currentJar->jarHandle);
    if(pStatus != JR_OK)
{
    logError("failed to move to start of jar file", "");
    return pStatus;
}

/* are we working with packaged zip/jar files? */
if(JR_USE_JAR == JR_OK)
{
    /* while we still have programs to process */
    while(currentJar!=NULL)
    {
        logInfo("----------------------------------------------------","");
        logInfo("processing: ", currentJar->jarPath);

        pStatus = getNextClassFile(FALSE);
        if(pStatus != JR_OK & pStatus != UNZ_END_OF_LIST_OF_FILE)
        {
            logError("failed to get next class file","");
            return pStatus;
        }

        pStatus = resetEvaluator();
        if(pStatus!=JR_EVAL_OK)
        {
            logError("failed to reset evaluator","");
            return JR_EVAL_FAIL;
        }

        evalStatus = getEvalStatus();

        /* while the lexer hasn't finished processing the program */
        while(evalStatus!=JR_EVAL_FINISHED)
        {
            /* process the next instruction */
            pStatus = jrlex(class);

            if(pStatus!=JR_EVAL_OK && pStatus!=JR_EVAL_FINISHED)
            {
                logError("failed to get next instruction","");
                return pStatus;
            }

            /* check internal status */
            evalStatus = getEvalStatus();

            if(evalStatus==JR_EVAL_CONTINUE)
            {
                /* lexing process was successful, retrieve next instruction(s) */
                peephole(getNextInstr());
```

```
            capture(getNextInstr());
        }
        else if(evalStatus==JR_EVAL_NEXT_CLASS)
        {
            peephole(getNextInstr());
            capture(getNextInstr());
            logInfo("successfully evaluated: ", getClassName(class));

            /* read another class into memory */
            pStatus = getNextClassFile(TRUE);
            if(pStatus==UNZ_END_OF_LIST_OF_FILE)
            {
                /* finished with this JAR */
                setNextEvalStatus(JR_EVAL_FINISHED);

                /* fast shut down */
                evalStatus=JR_EVAL_FINISHED;
            }
            else if(pStatus!=JR_OK)
            {
                logError("failed to get next class file","");
                return pStatus;
            }
        }
        else if(evalStatus!=JR_EVAL_NEW_STATUS &
                evalStatus!=JR_EVAL_FINISHED)
        {
            /* we've pinged our lexer at the wrong point and/or something
               terrible has happened */
            logError2("unexpected state reported from evaluator: ", evalStatus);
            return JR_INVALID_EVAL_STATUS;
        }
    }

    logInfo("successfully processed: ", currentJar->jarPath);

    /* finished with this JAR */
    pStatus = getNextJarFile();
    if(pStatus!=JR_OK)
    {
        logError("failed to get next JAR file","");
        return pStatus;
    }
}

/* clean up */
terminatePeephole();
terminateCapture();
terminateCodeConstructor();
    }
    else
    {
        /* no zip/jar files */
        /* todo :) */
    }

    return JR_OK;
}

/* clean up ------------------------------------------------------------- */

int terminateProcessor()
{
    /* de-allocate our global structs */
    free(jrGlobalInfo);
    free(jrFileInfo);

    return JR_OK;
}
```

D.2 jrEval.c

```
#include "jrJava.h"
#include "jrUtils.h"
#include "jrEval.h"
#include "jrXMLCodeConstructor.h"
#include "unzip.h"
#include "jrHarvester.h"

jr_instr             *instr = NULL; /* most recent evaluated instruction */

jr_eval_class_method *classMethods = NULL;

int evalStatus = JR_EVAL_OK; /* lexer status */
int nextEvalStatus = JR_EVAL_GET_METHODS;

int applyExpansion=TRUE;

int eStatus = JR_EVAL_OK; /* error status */
```

```
void setExpansion(int value)
{
    applyExpansion=value;
}


int resetEvaluator()
{
    jr_eval_class_method *ecm;

    /* tell lexer what it needs to do next */
    setNextEvalStatus(JR_EVAL_GET_METHODS);

    ecm=classMethods;
    while(classMethods)
    {
        ecm=classMethods;
        classMethods=classMethods->nextEcm;
        free(ecm);
    }

    return JR_EVAL_OK;
}


/* get/return local information ------------------------------------------ */
int getEvalStatus()
{
    return evalStatus;
}

void setEvalStatus(int status)
{
    evalStatus = status;
}

void setNextEvalStatus(int status)
{
    nextEvalStatus = status;
    evalStatus = JR_EVAL_NEW_STATUS;
}

jr_instr *getNextInstr()
{
    return instr;
}
/* ----------------------------------------------------------------------- */
```

```
int getClassMethods(jr_class *class)
{
    jr_constant          *constant=NULL;
    jr_method            *method=NULL;
    jr_attribute         *attribute=NULL;
    jr_eval_class_method *ecm=NULL;
    char                 *thisClassName;

    /* get class name */
    thisClassName=getClassName(class);

    logDebug("methods in class: ", thisClassName);
    method=class->methods;

    if(method==NULL)
    {
        logWarn("no methods in class: ", thisClassName);
        return JR_NO_CLASS_METHODS;
    }

    indent();

    while(method)
    {
        /* get code attribute first (because native and abstract methods dont
           have a code attribute) */
        attribute=method->attributes;
        while(attribute)
        {
            constant=getConstant(class, attribute->nameIndex);

            if(strcmp(constant->info.UTF8.bytes, JR_ATTR_CODE)==0)
                break;

            attribute=attribute->nextAttribute;
        }

        if(attribute==NULL)
        {
            logWarn("code attribute not found, assumed native or abstract","");
        }
        else
        {

            if(ecm==NULL)
```

```
            {
                ecm=newEvalClassMethod();
                classMethods=ecm;
            }
            else
            {
                ecm->nextEcm=newEvalClassMethod();
                ecm=ecm->nextEcm;
            }

            if(ecm==NULL)
            {
                logError("out of memory (new ecm)", "");
                return JR_OUT_OF_MEMORY;
            }

            ecm->className=thisClassName;

            /* get nameIndex */
            constant=getConstant(class, method->nameIndex);
            ecm->methodName=constant->info.UTF8.bytes;

            /* get descriptor */
            constant=getConstant(class, method->descriptorIndex);
            ecm->descriptor=constant->info.UTF8.bytes;

            /* get code */
            ecm->stream=attribute->info.code.code;
            ecm->streamLength=attribute->info.code.codeLength;

            logDebug("", ecm->methodName);
        }

        method=method->nextMethod;
    }

    unindent();

    if(classMethods==NULL)
    {
        logWarn("no methods in class: ", thisClassName);
        return JR_NO_CLASS_METHODS;
    }

    return JR_EVAL_OK;
}

int getBranchTargets(jr_class *class)
{
    jr_eval_class_method *ecm=classMethods;
    char *stream=NULL;
    int   streamOffset=0;
    int   streamLength=0;
    jr_instr *instr=NULL;
    jr_operand *operand=NULL;
    int count=0;
    jr_form *form=NULL;
    int i=0;
    jr_branch_target *bt=NULL;

    while(ecm)
    {
        stream=ecm->stream;
        streamOffset=ecm->streamOffset;
        streamLength=ecm->streamLength;

        while(streamOffset<streamLength)
        {
            if(instr!=NULL)
                free(instr);

            instr = newInstr();
            instr->bytecode = convertToUnsignedByte((stream +
                                                     streamOffset));

            streamOffset+=JR_BYTEBUF;

            /* if this is a branch instruction, check where it lands -
                the only branches we need to take into acount that always
                branch are goto(167), goto_w(200), jsr(168) and jsr_w(201) */
            if(instr->bytecode==200 || instr->bytecode==201)
            {
                /* read int from stream and add to offset */
                bt=newBranchTarget();
                if(bt==NULL)
                {
                    logError("out of memory (new branch target)","");
                    return JR_OUT_OF_MEMORY;
                }

                bt->target=streamOffset+convertToInt((stream+streamOffset))-1;
                if(ecm->branchTargets==NULL)
                {
                    ecm->branchTargets=bt;
                }
                else
                {
                    bt->nextBranchTarget=ecm->branchTargets;
```

```
        ecm->branchTargets=bt;
    }

}
else if((instr->branch!=always ||
        instr->bytecode==167   ||
        instr->bytecode==168) &
        instr->branch!=never)
{

logError2("found ", instr->bytecode);

    bt=newBranchTarget();
    if(bt==NULL)
    {
        logError("out of memory (new branch target)","");
        return JR_OUT_OF_MEMORY;
    }

    bt->target=streamOffset+convertToShort((stream+streamOffset))-1;
    if(ecm->branchTargets==NULL)
    {
        ecm->branchTargets=bt;
    }
    else
    {
        bt->nextBranchTarget=ecm->branchTargets;
        ecm->branchTargets=bt;
    }

}



/* now do the stuff necessary to get us to the next instruction */
eStatus = createInstrStruct(instr);
if(eStatus!=JR_OK)
{
    logError("failed to create internal instruction structure","");
    return JR_CONSTRUCT_FAIL;
}

eStatus = matchForm(class, instr, stream, streamOffset);
if(eStatus!=JR_OK)
{
    logError("failed to match form to instruction","");
    return eStatus;
}
```

```
        form = getForm(instr->forms, instr->eData.formId);

        if(form!=NULL)
        {
            /* get operands */
            operand = form->operands;
            count = countOperands(operand);

            for(i = 0; i<count; i++)
            {
                operand = getOperand(form->operands, i);

                /* should never get these - they should be removed by matchForm */
                if(operand->type==nxubyte)
                {
                    logError("found nxubyte in instruction: ",instr->name);
                    return JR_EVAL_FAIL;
                }
                else
                {
                    if(instr->eData.operandvals==NULL)
                    {
                         streamOffset+=JR_BYTEBUF;
                    }
                    else
                    {
                        /* something very wrong has happened */
                        logError("operand values found?!?","");
                        return JR_EVAL_FAIL;
                    }
                }
            }
        }

        ecm=ecm->nextEcm;
    }

    return JR_OK;
}



int matchForm(jr_class *class,
              jr_instr *instr,
              char     *stream,
              int       streamOffset)
{
    jr_form    *form;
```

```
jr_operand *operand;
int         formId;
int         order;
int         foundForm=FALSE;

int paddingBytes;
int defaultBytes;
int highBytes;
int lowBytes;
int totalOffsets;
int npairs;

/* no forms */
if(instr->forms==NULL)
{
    formId = 0;
}
/* special case for lookupswitch(171), tableswitch(170) and wide(196) */
else if(instr->bytecode==170)
{
    logDebug("found tableswitch", "");
    indent();

    form=instr->forms;

    if((streamOffset % 4)==0)
        paddingBytes=0;
    else
        paddingBytes=4-(streamOffset%4);

    logDebug2("streamOffset=", streamOffset-JR_BYTEBUF);
    logDebug2("padding bytes=", paddingBytes);
    streamOffset+=paddingBytes;

    /* find form with paddingBytes padding bytes */
    while(form)
    {
        order=0;

        /* padding bytes should be the first paddingBytes operands,
           therefore we try to get operands with order 0 through
           paddingBytes -1 */
        while(order<countOperands(form->operands))
        {
            operand=getOperand(form->operands, order);

            if(operand==NULL ||
               operand->operandVal==NULL ||
               operand->operandVal->value[0]!=0)
                break;


            order++;
        }

        if(order==paddingBytes)
        {
            foundForm=TRUE;
            break;
        }

        form=form->nextForm;
    }

    if(foundForm!=TRUE)
    {
        logDebug("failed to match form to tableswitch", "");
        return JR_FORM_MATCH_ERROR;
    }

    formId=form->id;
    logDebug2("chosen form id=", formId);

    /* now get to the total offsets section so we can alter the
       instruction to include this now known data */
    defaultBytes=convertToInt(stream+streamOffset);
    logDebug2("default bytes=", defaultBytes);

    streamOffset+=JR_INTBUF;
    lowBytes=convertToInt(stream+streamOffset);
    logDebug2("low bytes=", lowBytes);

    streamOffset+=JR_INTBUF;
    highBytes=convertToInt(stream+streamOffset);
    logDebug2("high bytes=", highBytes);

    totalOffsets=highBytes-lowBytes+1;
    logDebug2("total offsets=", totalOffsets);


    /* get the nxubyte type operand */
    order=0;
    while(order<=countOperands(form->operands))
    {
        operand=getOperand(form->operands,order);

        if(operand->type==nxubyte)
            break;
```

```
        order++;
    }

    if(order>countOperands(form->operands))
    {
        logDebug("failed to get nxubyte for tableswitch","");
        return JR_FORM_MATCH_ERROR;
    }

    /* change this operand from nxubyte to ubyte */
    operand->type=ubyte;

    /* convert the offsets from 32 bit to 8 bit bytes, and minus 1
       because we have converted the nxubyte */
    totalOffsets = (totalOffsets*4)-1;

    /* go to last operand in chain */
    while(operand->nextOperand)
        operand=operand->nextOperand;

    /* nxubyte should be the last operand so the next operands we
       add to the instruction will be taken incrementally from the
       order of the nxubyte */
    while(totalOffsets>0)
    {
        order++;

        /* create a new operand */
        operand->nextOperand=newOperand();

        /* move to new operand */
        operand=operand->nextOperand;

        /* set data */
        operand->type=ubyte;
        operand->order=order;

        totalOffsets--;
    }

    unindent();
}
else if(instr->bytecode==171)
{
    logDebug("found lookupswitch", "");
    indent();

    form=instr->forms;

if((streamOffset % 4)==0)
    paddingBytes=0;
else
    paddingBytes=4-(streamOffset%4);

logDebug2("streamOffset=", streamOffset-JR_BYTEBUF);
logDebug2("padding bytes=", paddingBytes);
streamOffset+=paddingBytes;

/* find form with paddingBytes padding bytes */
while(form)
{
    order=0;

    /* padding bytes should be the first paddingBytes operands,
       therefore we try to get operands with order 0 through
       paddingBytes -1 */
    while(order<countOperands(form->operands))
    {
        operand=getOperand(form->operands, order);

        if(operand==NULL ||
           operand->operandVal==NULL ||
           operand->operandVal->value[0]!=0)
            break;

        order++;
    }

    if(order==paddingBytes)
    {
        foundForm=TRUE;
        break;
    }

    form=form->nextForm;
}

if(foundForm!=TRUE)
{
    logDebug("failed to match form to lookupswitch", "");
    return JR_FORM_MATCH_ERROR;
}

formId=form->id;
logDebug2("chosen form id=", formId);

/* now get to the total offsets section so we can alter the
```

```
   instruction to include this now known data */
defaultBytes=convertToInt(stream+streamOffset);
logDebug2("default bytes=", defaultBytes);

streamOffset+=JR_INTBUF;
npairs=convertToInt(stream+streamOffset);
logDebug2("npairs=", npairs);

/* get the nxubyte type operand */
order=0;
while(order<=countOperands(form->operands))
{
    operand=getOperand(form->operands,order);

    if(operand->type==nxubyte)
        break;

    order++;
}

if(order>countOperands(form->operands))
{
    logDebug("failed to get nxubyte for lookupswitch","");
    return JR_FORM_MATCH_ERROR;
}

/* change this operand from nxubyte to ubyte */
operand->type=ubyte;

/* convert the npairs from 32 bit to 8 bit bytes, * 2 because they are
   pairs and minus 1 because we have converted the nxubyte */
npairs = ((npairs*4)*2)-1;

/* go to last operand in chain */
while(operand->nextOperand)
    operand=operand->nextOperand;

/* nxubyte should be the last operand so the next operands we
   add to the instruction will be taken incrementally from the
   order of the nxubyte */
while(npairs>0)
{
    order++;

    /* create a new operand */
    operand->nextOperand=newOperand();

    /* move to new operand */
    operand=operand->nextOperand;

        /* set data */
        operand->type=ubyte;
        operand->order=order;

        npairs--;
    }

    unindent();
}
else if(instr->bytecode==196)
{
    logError2("special case ", instr->bytecode);
    printf("special case %d", instr->bytecode);
    return JR_FORM_MATCH_ERROR;
}
/* otherwise just take first form */
else
{
    formId=1;

    /* need to check for expected values (only one ever happens here) */
    /* check value for aload_4 and aload_5 */
    if(instr->bytecode==25)
    {
        form=instr->forms;

        while(form)
        {
            operand=form->operands;

            if(operand->operandVal==NULL)
            {
                formId=form->id;
            }
            else if(operand->operandVal->value[0]==convertToUnsignedByte(stream+streamOffset))
            {
                formId=form->id;
                break;
            }


            form=form->nextForm;
        }

    }

}
```

```
    instr->eData.formId = formId;
    return JR_EVAL_OK;
}


int jrlex(jr_class *class)
{
    jr_instr             *instrList;
    jr_form              *form;
    jr_operand           *operand;
    jr_operandval        *operandval;
    jr_poststack         *poststack;
    jr_alt               *alt;
    jr_alt               *altList;
    int                   count;
    int                   i;
    jr_eval_class_method *ecm;

    /* update lexer status */
    if(evalStatus!=JR_EVAL_CONTINUE)
        evalStatus = nextEvalStatus;

    /* switch on current action */
    switch(evalStatus)
    {
        case JR_EVAL_GET_METHODS:

            /* get a list of methods in this class */
            eStatus = getClassMethods(class);
            if(eStatus==JR_NO_CLASS_METHODS)
            {
                nextEvalStatus=JR_EVAL_GET_METHODS;
                evalStatus = JR_EVAL_NEXT_CLASS;
                return JR_EVAL_OK;
            }
            else if(eStatus!=JR_EVAL_OK)
            {
                logError("failed to get class method list","");
                return eStatus;
            }

            logInfo("evaluating branch targets for class: ", getClassName(class));

            /* find out there the branch targets are */
            eStatus=getBranchTargets(class);
            if(eStatus!=JR_OK)
            {
                logError("failed to get class branch targets", "");
                return eStatus;
            }

            logInfo("evaluating methods for class: ", getClassName(class));

            ecm = classMethods;
            nextEvalStatus=JR_EVAL_METHOD;
            evalStatus=JR_EVAL_CONTINUE;
        break;
        case JR_EVAL_METHOD:

            /* move to next class method */
            ecm = classMethods;
            classMethods=classMethods->nextEcm;
            free(ecm);

            if(classMethods==NULL)
            {
                /* finished with this class */
                nextEvalStatus=JR_EVAL_GET_METHODS;
                evalStatus = JR_EVAL_NEXT_CLASS;
                return JR_EVAL_OK;
            }

            ecm=classMethods;
            nextEvalStatus=JR_EVAL_METHOD;
            evalStatus = JR_EVAL_CONTINUE;

        break;
        case JR_EVAL_CONTINUE:
            ecm=classMethods;
        break;
        case JR_EVAL_FINISHED:
            /* evaluator has been told it is finished */
            return JR_EVAL_FINISHED;
        break;
        default:
            logError2("unexpected eval status flag: ", evalStatus);
            return JR_INVALID_EVAL_STATUS;
        break;
    }
/*
    logDebug("this class: ", classMethods->className);
    logDebug("this method: ", classMethods->methodName);
*/

    /* now continue processing */
    if(evalStatus==JR_EVAL_CONTINUE)
    {
        /* free instr PROPERLY! */
```

```
free(instr);
instr = newInstr();
instr->bytecode = convertToUnsignedByte((ecm->stream +
                                         ecm->streamOffset));
instr->eData.streamOffset=ecm->streamOffset;
instr->eData.className=ecm->className;
instr->eData.methodName=ecm->methodName;

/* check whether this instruction is a branch target */
instr->eData.isBranchTarget=isBranchTarget(ecm->branchTargets,
                                           ecm->streamOffset);
incrementStream(ecm, JR_BYTEBUF);

eStatus = createInstrStruct(instr);
if(eStatus!=JR_OK)
{
    logError("failed to create internal instruction structure","");
    return JR_CONSTRUCT_FAIL;
}

logDebug("", instr->name);
indent();

/* need to examine forms to see what matches our operand stack */
eStatus = matchForm(class, instr, ecm->stream, ecm->streamOffset);
if(eStatus!=JR_OK)
{
    logError("failed to match form to instruction","");
    return eStatus;
}

form = getForm(instr->forms, instr->eData.formId);
instrList = instr; /* save the head of the list */

/* form will be NULL if no forms specified */
if(form!=NULL)
{
    /* check whether we are allowed expand instructions */
    if(applyExpansion==TRUE)
    {
        /* check for alternatives ---------------------------------- */
        if(form->alts!=NULL)
        {
            alt = form->alts;
            while(alt)
            {
                /* create new instruction for each alt */
                instr->eData.nextInstr = newInstr();
                /* move to the new instr */
                instr=instr->eData.nextInstr;

                /* set the bytecode */
                instr->bytecode = alt->bytecode;

                /* set eData */
                instr->eData.streamOffset=ecm->streamOffset-JR_BYTEBUF;
                instr->eData.className=ecm->className;
                instr->eData.methodName=ecm->methodName;

                /* create internal instruction structure */
                eStatus = createInstrStruct(instr);
                if(eStatus!=JR_OK)
                {
                    logError("failed to create alternate internal instruction structure","");
                    return JR_CONSTRUCT_FAIL;
                }

                logDebug("expanding to: ", instr->name);

                /* setup pointers from this alternate instruction to alternative
                   values (which are located in the original instruction) */
                instr->eData.operandvals = alt->operandvals;
                instr->eData.prestackvals = alt->prestackvals;
                instr->eData.poststackvals = alt->poststackvals;

                /* match form - assume non-recursive alternates (that would be silly) */
                eStatus = matchForm(class, instr, ecm->stream, ecm->streamOffset);
                if(eStatus!=JR_EVAL_OK)
                {
                    logError("failed to match form to alternate instruction","");
                    return eStatus;
                }

                /* move to next alt */
                alt=alt->nextAlt;
            }

            /* move to the head of the list+1 (to skip original) */
            instr = instrList->eData.nextInstr;

            /* copy forward the branch target flag to the first instr */
            instr->eData.isBranchTarget=instrList->eData.isBranchTarget;

        }
        /* end check for alternatives ------------------------------ */
    }
```

```
/* now for each instruction in the list (if there is one) */
while(instr)
{
    /* extract this form from the formlist */
    form = getForm(instr->forms, instr->eData.formId);

    /* get operands */
    operand = form->operands;
    count = countOperands(operand);

    for(i = 0; i<count; i++)
    {

        operand = getOperand(form->operands, i);

        /* should never get these - they should be removed by matchForm */
        if(operand->type==nxubyte)
        {
            logError("found nxubyte in instruction: ",instr->name);
            return JR_EVAL_FAIL;
        }
        else
        {
            if(instr->eData.operandvals==NULL)
            {
                operand->operandVal = newValue();
                operand->operandVal->length = JR_BYTEBUF;
                strncpy(operand->operandVal->value,
                        (ecm->stream + ecm->streamOffset),
                        JR_BYTEBUF);
                incrementStream(ecm, JR_BYTEBUF);
            }
            else
            {
                operandval = getOperandval(instr->eData.operandvals,
                                           operand->order);
                operand->operandVal = newValue();

                /* check if the value is inherited */
                if(operandval->inherit==TRUE)
                {
                    operand->operandVal->length=JR_BYTEBUF;
                    strncpy(operand->operandVal->value,
                           (ecm->stream + ecm->streamOffset),
                            JR_BYTEBUF);
                    incrementStream(ecm, JR_BYTEBUF);
                }
                else
                {
                    operand->operandVal->length = operandval->altVal->length;
                    strncpy(operand->operandVal->value,
                            operandval->altVal->value,
                            operandval->altVal->length);
                }
            }
        }

        /* hack for specialised aload_4 and aload_5 to discard the
            operand */
        if(instr->bytecode==203 || instr->bytecode==216)
        {
            incrementStream(ecm, JR_BYTEBUF);
        }

        instr = instr->eData.nextInstr;
    }

} /* if form!=NULL */

if(instrList->eData.nextInstr)
{
    /* alternate instructions found, discard original and return
       list of alternates */
    instr = instrList->eData.nextInstr;
    free(instrList);
}
else
{
    /* move to head of list, ready to return */
    instr = instrList;
}

unindent();
return JR_EVAL_OK;
}
else
{
    logError2("expected lexer continuation, found: ", evalStatus);
    return JR_INVALID_EVAL_STATUS;
}
}
```

D.3 jrCapture.c

```
#include "jrJava.h"
#include "jrUtils.h"
#include "jrEval.h"
#include "jrCapture.h"
#include <time.h>

char                outputPath[JR_MAX_PATH];

char                codeOutputPath[JR_MAX_PATH+4];
char                freqOutputPath[JR_MAX_PATH+8];
char                successorOutputPath[JR_MAX_PATH+13];
char                predecessorOutputPath[JR_MAX_PATH+15];


FILE               *codeFile; /* file that holds disassembled code */
FILE               *freqFile; /* instructions by frequency */
FILE               *successorFile;
FILE               *predecessorFile;

char                *names[JR_BYTECODE_COUNT];
int                  sizes[JR_BYTECODE_COUNT];
int                  freqs[JR_BYTECODE_COUNT];
jr_captured_operand *capturedOperands[JR_BYTECODE_COUNT];
jr_cessor           *successors[JR_BYTECODE_COUNT];
jr_cessor           *predecessors[JR_BYTECODE_COUNT];

int                 lastBytecode=-1; /* last bytecode captured */
int                 lastInstrBytes=0;
int                 lastBranches=FALSE; /* flag indicating the last bytecode
                                           captured was a branch instruction */

long totalInstrBytes=0;
int  thisInstrBytes=0;

int cpStatus = JR_OK;


jr_captured_operand *newCapturedOperand()
{
    jr_captured_operand *co=malloc(sizeof(jr_captured_operand));
    if(co!=NULL)
    {
        co->value=0;
        co->freq=0;
        co->nextCapturedOperand=NULL;
    }
    return co;
}

jr_cessor *newCessor()
{
    jr_cessor *c=malloc(sizeof(jr_cessor));
    if(c!=NULL)
    {
        c->bytecode=0;
        c->bytes=0;
        c->freq=0;
        c->nextCessor=NULL;
    }
    return c;
}

jr_captured_operand *getCo(unsigned char bytecode, int value)
{
    jr_captured_operand *co=capturedOperands[bytecode];
    while(co)
    {
        if(co->value==value)
            break;

        co=co->nextCapturedOperand;
    }
    return co;
}


int setupCapture(char *jarPath)

{

    time_t timer=time(NULL);
    int i;



    logInfo("Enter path to record statistics:","");

    scanf("%s", &outputPath);



    /* debug dir codes */

    if(outputPath[0]=='u')
```

```
{

    strcpy(outputPath,

 "/u/cs/2/cs2ajss/final/semester2/cm30082/code/jarReader/");

}

else if(outputPath[0]=='h')

{

    strcpy(outputPath,

      "/home/alan/final/semester2/cm30082/code/jarReader/");

}

strcpy(codeOutputPath, outputPath);


codeFile = fopen(strcat(codeOutputPath,"code"), "wb");

if(codeFile==NULL || ferror(codeFile)>0)

{

    logError(codeOutputPath, " : no such file or directory");

    return JR_FILE_NOT_FOUND;

}


fprintf(codeFile, "%s\n%s%s\n%s\n",
"--------------------------------------------------------------------------------",
asctime(localtime(&timer)), jarPath,
"--------------------------------------------------------------------------------");

/* clean up */
fclose(codeFile);

strcpy(freqOutputPath, outputPath);

freqFile = fopen(strcat(freqOutputPath,"freq.csv"), "wb");

if(freqFile==NULL || ferror(freqFile)>0)

{

    logError(freqOutputPath, " : no such file or directory");

    return JR_FILE_NOT_FOUND;

}

fprintf(freqFile,
"\"bytecode\",\"name\",\"instruction frequency\",\"instruction size frequency (bytes)\",\"total bytes\",\"operand\",\"o

fclose(freqFile);

strcpy(successorOutputPath, outputPath);

successorFile = fopen(strcat(successorOutputPath,"successor.csv"), "wb");

if(successorFile==NULL || ferror(successorFile)>0)

{

    logError(successorOutputPath, " : no such file or directory");

    return JR_FILE_NOT_FOUND;

}

fprintf(successorFile,
"\"bytecode\",\"name\",\"successor bytecode\",\"successor name\",\"successor frequency\",\"pair size frequency (bytes)\

fclose(successorFile);

strcpy(predecessorOutputPath, outputPath);

predecessorFile = fopen(strcat(predecessorOutputPath,"predecessor.csv"), "wb");

if(predecessorFile==NULL || ferror(predecessorFile)>0)

{

    logError(predecessorOutputPath, " : no such file or directory");

    return JR_FILE_NOT_FOUND;

}

fprintf(predecessorFile,
```

```
    "\"bytecode\",\"name\",\"predecessor bytecode\",\"predecessor name\",\"predecessor frequency\",\"pair size frequency (bytes)\"\n");

    fclose(predecessorFile);

    for(i=0;i<JR_BYTECODE_COUNT;i++)
    {
        /* init the arrays */
        names[i]=NULL;
        sizes[i]=0;
        freqs[i]=0;
        capturedOperands[i]=NULL;
        successors[i]=NULL;
        predecessors[i]=NULL;
    }

    lastBytecode=-1;
    lastInstrBytes=0;
    lastBranches=FALSE;

    totalInstrBytes=0;
    thisInstrBytes=0;



    return JR_OK;

}

int captureCessor(unsigned char baseBytecode,
                  unsigned char cessorBytecode,
                  int pairBytes,
                  int cessorType)
{
    jr_cessor *cessors;

    if(cessorType==JR_SUCCESSOR)
    {
        cessors = successors[baseBytecode];
    }
    else if(cessorType==JR_PREDECESSOR)
    {
        cessors = predecessors[baseBytecode];
    }
    else
    {
        logError2("invalid cessor type: ",cessorType);
        return JR_INVALID_CESSOR;
    }
    while(cessors)
    {
        if(cessors->bytecode==cessorBytecode)
            break;

        cessors=cessors->nextCessor;
    }

    if(cessors!=NULL)
    {
        /* found cessor */
        cessors->freq+=1;
        cessors->bytes+=pairBytes;
    }
    else
    {
        cessors=newCessor();
        if(cessors==NULL)
        {
            logError("out of memory (new cessor)","");
            return JR_OUT_OF_MEMORY;
        }

        cessors->bytecode=cessorBytecode;
        cessors->bytes+=pairBytes;
        cessors->freq+=1;


        if(cessorType==JR_SUCCESSOR)
        {
            cessors->nextCessor=successors[baseBytecode];
            successors[baseBytecode]=cessors;
        }
        else
        {
            cessors->nextCessor=predecessors[baseBytecode];
            predecessors[baseBytecode]=cessors;
        }
    }

    return JR_OK;

}



int capture(jr_instr *instr)
```

```
{

    int                  i;

    jr_form             *form;
    jr_captured_operand *co;
    int                  coValue;
    int                  operandCount;



    codeFile = fopen(codeOutputPath, "ab");

    if(codeFile==NULL || ferror(codeFile)>0)

    {

        logError(outputPath, " : no such file or directory");

        return JR_FILE_NOT_FOUND;

    }



    while(instr)
    {
        thisInstrBytes=1; /* instruction bytecode */

        form = instr->forms;

        while(form)
        {
            if(form->id==instr->eData.formId)
                break;

            form=form->nextForm;
        }

        if(form!=NULL)
        {

            if(form->operands)
            {
                operandCount=countOperands(form->operands);
                thisInstrBytes+=operandCount;

                if(instr->bytecode==170 || instr->bytecode==171)
                {
                    /* there are so few of these that it doesn't matter if
                       we fail to record these stats (for now) */
                    coValue = concatinateOperands(form->operands, operandCount);
                    /*fprintf(codeFile, " %d", co->value);*/
                }
                else
                {
                    /* fix invokeinterface */
                    if(instr->bytecode==185)
                    {
                        /* hack to get real offset and ignore count and the zero:
                           invokeinterface[offsetByte][offsetByte][count][0] */
                        free(form->operands->nextOperand->nextOperand->nextOperand);
                        free(form->operands->nextOperand->nextOperand);
                        form->operands->nextOperand->nextOperand=NULL;
                        /* get new coValue */
                        coValue=concatinateOperands(form->operands, countOperands(form->operands));
                    }
                    /* invokeinterface_n */
                    else if(instr->bytecode==220)
                    {
                        /* one less operand */
                        free(form->operands->nextOperand->nextOperand);
                        free(form->operands->nextOperand);
                        form->operands->nextOperand=NULL;
                        /* get new coValue */
                        coValue=concatinateOperands(form->operands, countOperands(form->operands));
                    }
                    else
                    {
                        coValue = concatinateOperands(form->operands, operandCount);
                    }


                    /* add to captured operands list */
                    co=getCo(instr->bytecode, coValue);

                    if(co==NULL)
                    {
                        co=newCapturedOperand();
                        if(co==NULL)
                        {
                            logError("out of memory (new captured operand)", "");
                        }

                        co->value=coValue;
                        co->freq=1;

                        if(capturedOperands[instr->bytecode]==NULL)
```

```
                {
                    capturedOperands[instr->bytecode]=co;
                }
                else
                {
                    co->nextCapturedOperand=capturedOperands[instr->bytecode];
                    capturedOperands[instr->bytecode]=co;
                }
            }
            else
            {
                co->freq+=1;
            }
        }

    }

}

/* update the size for this bytecode */
sizes[instr->bytecode]+=thisInstrBytes;

/* extract the information from the instruction */
if(instr->eData.className!=NULL)

    fprintf(codeFile, "%s.", instr->eData.className);

else

    fprintf(codeFile, "NULL.");

if(instr->eData.methodName!=NULL)

    fprintf(codeFile, "%s ", instr->eData.methodName);

else

    fprintf(codeFile, "NULL ");

fprintf(codeFile, "%d:", instr->eData.streamOffset);

if(instr->name!=NULL)
{
    /* write to codeFile */

    fprintf(codeFile, " %s", instr->name);

    /* store name */
    if(names[instr->bytecode]==NULL)
    {
        names[instr->bytecode]=newStringBuffer(JR_BYTECODE_NAMEBUF);
        strcpy(names[instr->bytecode], instr->name);
        freqs[instr->bytecode]=1;
    }
    else
    {
        freqs[instr->bytecode]+=1;
    }
}

else
{
    logWarn("capturing a bytecode with no name!", "");

    fprintf(codeFile, " NULL");
}

if(form!=NULL)
{
    if(form->operands)
        fprintf(codeFile, " %d", coValue);
}

if(instr->branch!=never)

    fprintf(codeFile, " (branch=%s)", codeToBranchTypeName(instr->branch));

fprintf(codeFile, "\n");

/* now deal with the cessors */
if(lastBytecode!=-1)
{
    /* a cessor should not be logged if a branch operation forbids it */
    if(lastBranches==FALSE & instr->eData.isBranchTarget==FALSE)
    {
        cpStatus = captureCessor(lastBytecode,
                                 instr->bytecode,
                                 lastInstrBytes+thisInstrBytes,
                                 JR_SUCCESSOR);
        if(cpStatus!=JR_OK)
        {
            logError2("failed to capture successor for bytecode: ", lastBytecode);
            return cpStatus;
        }


        cpStatus = captureCessor(instr->bytecode,
                                 lastBytecode,
```

```
                                    lastInstrBytes+thisInstrBytes,
                                    JR_PREDECESSOR);
                if(cpStatus!=JR_OK)
                {
                    logError2("failed to capture predecessor for bytecode: ", instr->bytecode);
                    return cpStatus;
                }
            }
        }

        if(instr->branch==never)
            lastBranches=FALSE;
        else
            lastBranches=TRUE;

        lastBytecode=instr->bytecode;
        lastInstrBytes=thisInstrBytes;
        totalInstrBytes+=thisInstrBytes;

        instr=instr->eData.nextInstr;

    }



    /* clean up */
    fclose(codeFile);

    return JR_OK;
}

int concatinateOperands(jr_operand *operands, int count)

{

    int i;

    char buf[count];

    /* deal with negative operands */
    if(operands->operandVal->value[0]=='-')
        return -1;

    for (i=0; i<count; i++)
    {
        buf[i]=operands->operandVal->value[0];
        operands=operands->nextOperand;
    }

    switch(count)
    {
        case 1:
            return convertToUnsignedByte(buf);
        break;
        case 2:
            return convertToShort(buf);
        break;
        case 4:
            return convertToInt(buf);
        break;
        default:
            logError2("Error concatinating operands: ",count);
            return 0;
        break;
    }
}



int terminateCapture()

{
    int i;
    jr_captured_operand *co;
    jr_cessor           *cessor;

    freqFile = fopen(freqOutputPath, "ab");

    if(freqFile==NULL || ferror(freqFile)>0)

    {

        logError(freqOutputPath, " : no such file or directory");

        return JR_FILE_NOT_FOUND;

    }

    for(i=0;i<JR_BYTECODE_COUNT;i++)
    {
        if(i!=170 & i!=171)
        {
            co=capturedOperands[i];
            if(co!=NULL)
            {
                while(co)
                {
                    if(names[i]==NULL)
```

```
                {
                    fprintf(freqFile, "\"%d\",\"UNUSED\",", i);
                }
                else
                {
                    fprintf(freqFile, "\"%d\",\"%s\",", i, names[i]);
                }

                fprintf(freqFile, "\"%d\",\"%d\",\"%d\",", freqs[i],
                                                           sizes[i],
                                                           totalInstrBytes);

                fprintf(freqFile, "\"%d\",\"%d\"\n", co->value, co->freq);
                co=co->nextCapturedOperand;
            }
        }
        else
        {
            if(names[i]==NULL)
            {
                fprintf(freqFile, "\"%d\",\"UNUSED\",", i);
            }
            else
            {
                fprintf(freqFile, "\"%d\",\"%s\",", i, names[i]);
            }

            fprintf(freqFile, "\"%d\",\"%d\",\"%d\",", freqs[i],
                                                       sizes[i],
                                                       totalInstrBytes);

            fprintf(freqFile, "\"N/A\",\"N/A\"\n");
        }


    }
}

/* clean up */
fclose(freqFile);

/* now the cessors */
successorFile = fopen(successorOutputPath, "ab");

if(successorFile==NULL || ferror(successorFile)>0)
{
    logError(successorOutputPath, " : no such file or directory");
    return JR_FILE_NOT_FOUND;
}

for(i=0;i<JR_BYTECODE_COUNT;i++)
{
    cessor=successors[i];
    while(cessor)
    {
        fprintf(successorFile,
                "\"%d\",\"%s\",\"%d\",\"%s\",\"%d\",\"%d\"\n", i,
                                                               names[i],
                                                               cessor->bytecode,
                                                               names[cessor->bytecode],
                                                               cessor->freq,
                                                               cessor->bytes);
        cessor=cessor->nextCessor;
    }
}

/* clean up */
fclose(successorFile);

predecessorFile = fopen(predecessorOutputPath, "ab");

if(predecessorFile==NULL || ferror(predecessorFile)>0)
{
    logError(predecessorOutputPath, " : no such file or directory");
    return JR_FILE_NOT_FOUND;

}

for(i=0;i<JR_BYTECODE_COUNT;i++)
{
    cessor=predecessors[i];
    while(cessor)
    {

        fprintf(predecessorFile,
                "\"%d\",\"%s\",\"%d\",\"%s\",\"%d\",\"%d\"\n", i,
                                                               names[i],
                                                               cessor->bytecode,
                                                               names[cessor->bytecode],
                                                               cessor->freq,
                                                               cessor->bytes);
        cessor=cessor->nextCessor;
    }
}

/* clean up */
fclose(predecessorFile);
}
```