Is superoptimisation a viable technique for virtual machines?

Abstract

The term “optimisation” is typically used by computer scientists to refer to the improvement of the performance of code. Superoptimisation, first proposed by Massalin in 1986, ensures true optimality through an exhaustive search of all possible programs. Superoptimisers have been developed for many machine architectures, and used for diverse purposes including automating peephole optimisation in C compilers and binary translation between instruction sets. The output of superoptimisers is frequently surprising to human experts and often takes advantage of side-effects or obscure characteristics of the targeted hardware.

Virtual machines (VMs) are an increasingly popular mechanism for providing a common platform across heterogenous hardware architectures, but no published work has examined whether superoptimisation is a viable technique for VMs.

A superoptimiser for the Java virtual machine has been developed and used to generate optimal versions of mathematical functions previously targeted by earlier research. We have demonstrated that these versions are superior to the implementations shipped with the Java SDK or generated by the Java compiler. We also have some useful observations concerning techniques for extending the tractable search space to include longer programs.

Acknowledgements

I am extremely grateful to Des Watson both for his support throughout the project, and for first introducing me to the topic of superoptimisation.

Tom Hume **MSc, Advanced Computer Science, University of Sussex** **Candidate # 93970**

Contents

1. Introduction

Computer scientists define optimisation as the improvement in efficiency of software (Aho, Lam, Sethi and Ullman, 2007). The definition of efficiency can vary: typically it might be minimal execution speed or use of resources. Optimisation is usually carried out either by a human being (in the course of system or algorithm design), automatically during the compilation of source code, or at run-time (when using Just-In-Time compilers).

Despite its etymological roots (the Latin word optimus means “best”), for many reasons optimisation rarely results in perfectly optimal software. Optimality is both difficult to achieve and hard to recognise, and perfection as measured by any single metric may not be desirable: optimising for code size might unacceptably degrade execution speed. A real-world system may make trade-offs and eschew absolute optimality for overall performance.

Superoptimisation, in contrast, concerns itself with the search for true optimality. First proposed by Massalin in 1987, it involves performing an exhaustive search of all possibilities to find optimal programs, and has been used in several contexts: notably automatic binary translation (Bansal and Aiken, 2011), peephole optimisations (Bansal and Aiken, 2006), branch code generation in the GNU C compiler (Sayle 2008) and elimination of branch instructions from compiled code (Granlund 1992). Researchers have frequently found that superoptimisers generate code which is more efficient than that hand-crafted by experts, and which often exploits unexpected or underused characteristics of the hardware it runs on.

Over the last 20 years virtual machines (VMs) have become a popular means of offering consistent run-time environments atop various hardware architectures. The Java Virtual Machine (JVM) is the most commercially successful VM (Oracle, 2012), but Smalltalk-80 and the Common Language Runtime of Microsoft’s .NET Framework provide further examples.

All superoptimisers to date have produced programs targeting specific hardware architectures. No research has yet been published into the applicability of superoptimisation to VMs; this project seeks to correct this omission with a piece of novel research.

2. Previous work

Literature review

As part of an earlier university project (Hume, 2011), the author carried out a literature review of superoptimisation. Here we re-examine the literature with specific focus on our project: considering target architectures, implementation details and experimental results.

Hardware architectures

Past superoptimisation efforts have targeted a wide range of processor instruction sets:

Researcher	Year	Architectures
Massalin	1987	68000
Granlund	1992	RS/6000, SPARC, 68000, 88000, AMD 29k, i386 (x86)
Joshi, Nelson & Randall	2002	Alpha EV6, IA-64
Bansal & Aiken	2006	x86
Sayle	2008	x86, IA-64
Crick	2009	SPARC V8, MIPS R2000
Bansal & Aiken	2011	x86 (translated from PowerPC)

Target functions

Research has tended to focus either on a few mathematical functions, or broader problems involving substitution of machine code instructions:

Researcher	Year	Target functions
Massalin	1987	`signum`, `abs`, `max`, `min`, logical tests, convert decimal to binary, multiply and divide by constants
Granlund	1992	Many, including `signum`, `min`, `max` and `abs`
Joshi, Nelson & Randall	2002	Byte swapping, packet checksum calculation
Bansal & Aiken	2006	Large numbers of automated peephole optimisations
Sayle	2008	`switch` statements in GCC output
Crick	2009	`signum`, `argredundance`, and peephole optimisations
Bansal & Aiken	2011	Automated translation of binaries between architectures

Approaches to testing

Testing has been probabilistic (Massalin, 1987 and Granlund, 1992), or eschewed in favour of generating provably optimal programs (Joshi, Nelson and Randall, 2002 or Crick, 2009):

Researcher	Year	Test approach
Massalin	1987	Probabilistic testing with known values, manual checking
Granlund	1992	Random input values, manual checking
Joshi, Nelson & Randall	2002	N/A: uses SAT solver to output provably optimal programs
Bansal & Aiken	2006	Probabilistic execution test, boolean test with SAT solver
Sayle	2008	`None`
Crick	2009	N/A: uses ASP to output provably optimal programs
Bansal & Aiken	2011	Probabilistic execution test, boolean test with SAT solver

Massalin observed that the ordering and choice of test vectors was important; tests which fail early result in better performance: "*the test vectors for the* `signum` *function included -1000, 0 and 456 as the first three vectors. This quickly eliminates programs that return the same answer regardless of argument, answers of the same sign, as well as programs that return their argument*".

Implementation details

This project is treads in the path of Massalin and Granlund, so we examined their papers more closely for lessons which might affect our implementation.

While Massalin enumerated and tested every possibility in ascending order of program length (Massalin, 1987), Granlund explored and pruned a tree structure of possibilities, to avoid ever testing obviously useless sets of instructions (Granlund, 1992).

Massalin avoided the use of pointers, used a subset of the 68020 instruction set, and filtered out sequences of instructions which were clearly not optimal (e.g. "move X, Y followed by move Y, X"). For one target function (`Abs`) he explicitly avoided the use of conditional jumps. Granlund also avoided pointer operations, restricting himself to register operations only, and created a superset of allowed operations which all his various target architectures supported. These did not include any branching operations (understandably, as he was looking to superoptimisation to remove such operations from the output of GCC). Bansal and Aiken limited themselves to a single branch per sequence in their 2006 paper. In their 2011 work on binary translation, they allowed the use of direct memory accesses, branches and flags.

Granlund pruned his tree of possibilities by limiting the set of constants used in generated code to -1, 0, +1, 2^{31} and 2^{31}-1. Even so, he noted that a search for programs 5 operations long with three inputs took more than an hour; Massalin spent several hours looking for programs 12 operations long.

Most superoptimisers have used program size as their cost function; Bansal and Aiken (2006) considered an estimate of runtime as well.

Observations regarding results

Massalin noted that "*many functions that would normally be written with conditional jumps are optimized into short programs without them*" (Massalin, 1987), and that in particular his superoptimiser produced a `signum` implementation 4 instructions long for the 68000 architecture, by taking advantage of properties of 2s complement arithmetic and using the carry bit:

```
(x in d0)
add.l d0,d0       add d0 to itself
subx.l dl,dl      subtract (dl + Carry) from dl
negx.l d0         put (0 - d0 - Carry) into d0
addx.l dl,dl      add (dl + Carry) to dl
(signum(x) in dl} (4 instructions}
```

Granlund similarly noted that the superoptimiser took advantage of aspects of the hardware which might elude experts: "*this difference in behavior of the i386 and RS/6000 was a surprise to at least one quite proficient assembler language programmer who has written compilers for both machines*" (Granlund, 1992)

These observations bring to mind the work of Adrian Thompson in evolvable hardware (Thompson 1996), deriving circuits whose components sometimes communicated by means other than cell-to-cell wires. Thompson's circuits performed their tasks well, but not using the mechanisms a human being would naturally consider.

Regarding genetic approaches

On several occasions when explaining superoptimisation to peers, the listener would associate it with the use of genetic algorithms (GAs). There is a significant difference between the two approaches: GAs reduce the search space by using a fitness function to navigate towards a goal state, whilst superoptimisation is an exhaustive search of possibilities. Thus a genetic approach might make search more tractable, but runs the dual risks of getting stuck in local maxima and reliance on a strong and well-defined fitness function to guide the evolution of programs.

In contrast, superoptimisation has to cope with exponential growth of the search space, but guarantees that it will be fully explored.

Superoptimisation and VMs

Through the review of available literature and through conversations with some researchers in this field (notably, Bansal and Aiken), it was established that no known previous work has been published examining superoptimisation in the specific context of VMs.

Previous superoptimisers have produced output which used side-effects of the hardware architecture in unexpected ways. VMs are implemented in software, and VMs implemented by many vendors must be well-specified in order to be consistent across implementations.

We cannot take for granted that such VMs will include the side-effects a superoptimiser might make good use of, nor that a well-known VM can be programmed in non-intuitive ways. Thus it is not clear that a superoptimiser would generate any useful results when targeting a VM. However, this assumption is far from certain and so targeting superoptimisers to VMs remains a worthwhile research area.

3. Method

Introduction

To investigate the viability of superoptimisation for VMs, we:

1. Developed a superoptimiser for the JVM: i.e. a program which allows a target function to be defined and searches the set of all possible Java bytecode programs for the shortest ones which fit the definition of this target function;
2. Used the superoptimiser thus developed to search for optimal versions of several mathematical functions generated by previous researchers, all of which ship as part of the Java standard libraries and thus have reference implementations;
3. Verified the generated optimal programs by hand and in a test framework;
4. Compared the generated optimal programs to both the Java standard library implementations, and our own versions compiled from Java source.

Knuth (1970, referenced in Crick 2009) and Hennessy and Patterson (2007, referenced in Crick 2009), asserted that 5-6 operations is the minimal viable length for a useful program. The longest Java standard library version of any target functions we sought was `signum`, 9 Java opcodes long. We therefore confined ourselves to searching for short programs up to 8 JVM operations in length: long enough that if a result is found, we have an improvement.

Failing to find shorter versions of any target function would also be an interesting result, as it would demonstrate the optimality (for size) of the existing implementations.

Target functions

Our target functions for superoptimisation were as follows:

1. `Identity`: returns its only argument. As the simplest possible function, and one whose optimal sequence is easily hand-calculated, this was used early in the project to test a complete run of the system;

2. `Negate`: returns the negative of its argument. A trivial program that manipulates its input;
3. `Max`: returns the greater of its two arguments. As a decision procedure, it felt likely that this function would use one of the branching operations in the JVM (e.g. `IFEQ`);
4. `Min`: returns the lesser of its two arguments; a useful function to compare to `Max`;
5. `Abs`: returns the absolute of its argument (i.e. a positive integer);
6. `Signum`: returns 1 if the argument is positive, −1 if negative, and 0 if 0.

Of these, `Signum` is the most complex and interesting and has the longest Java standard library implementation. It was the key sequence examined by many previous researchers from Massalin onwards, and interesting implementations have frequently been found.

Limitations of the superoptimiser

The superoptimiser limits itself to a subset of Java operations, in order to restrict the search space (and thus superoptimiser execution time) and simplify implementation. The limitations are as follows:

1. Whilst branching opcodes (e.g. `IFEQ`) are available for inclusion in generated programs, only branches forwards are permitted. Backwards branching introduces the possibility of loops, and thus requires that equivalence tests handle programs which do not terminate. After implementing backwards branches, we discovered that coping with such programs slowed equivalence testing unacceptably, and therefore disabled backwards branching. Unlike Bansal and Aiken (2006), we permit multiple branches per program;
2. The superoptimiser restricts itself to integer operations only; given the set of target programs we are superoptimising, this is appropriate. Bytes can be pushed onto the stack using `BIPUSH` or `ICONST_n` opcodes, and JVM arithmetic carried out on them and integers. The superoptimiser therefore need not concern itself with issues of type safety in generated sequences;
3. As well as eschewing non-integer data types, we make no use of Java objects or arrays, further restricting the types of programs which the superoptimiser might generate, at the gain of not considering opcodes specific to object or array manipulation (e.g. `INVOKEVIRTUAL`, `ARRAYLENGTH`) for inclusion into programs;

4. Generated programs make no use of the constant pool; they may therefore be suboptimal if a shorter program would return or use “magic” values;

5. No unconditional branching is allowed, i.e. the `GOTO` operation is not permitted, and conditional branches based on constants are filtered out. In a late search for a 7-long `signum`, we permitted `GOTO`;

6. `ISTORE` and `ILOAD` operations allow an arbitrary number of local variables to have values written to and read from. Java also includes shorthand operations for accessing a limited number of local variables, `ISTORE_n` and `ILOAD_n` (where $n \leq 3$). We allow these shorthand operations, and thus limit our programs to using a maximum of 4 local variables. Given that our maximum sequence length is 8, and for a variable to be minimally used it must be read from and written, this limitation does not restrict the set of viable programs the superoptimiser may produce;

7. In some of our later experiments to search for longer sequences, we further constrained the search space by insisting that programs start by reading their argument (`ILOAD_0`) and finish by returning a value (`IRETURN`).

8. We also constrained the set of valid arguments to `BIPUSH` and `IINC` to a subset of possible byte values: -127, 127, 0, positive and negative powers of 2 and these last values less 1.

The JVM opcodes used by the superoptimiser are listed in Appendix A.

Critique of method

In addition to the limitations above, our experiments could be criticised as follows:

1. Our metric for optimality is code size (measured by the number of JVM opcodes): shorter programs are better. Whilst in keeping with earlier work, this ignores the possibility that opcodes may differ in execution speed, or be translated by a Just-In-Time compiler to machine code instructions for the underlying hardware in an unpredictable fashion;

2. Our equivalence testing was relatively unsophisticated: we tried some hand-picked values to exercise generated programs. For those which pass these tests we did manual analysis, or fed large numbers of random inputs and compared output behaviour to that of the original function. Whilst we feel confident in this approach, more formal alternatives like abstract interpretation (Cousot, 1981) may have a role to play too.

3. The exponential growth of the search space prevented us from looking at programs beyond 8 opcodes in length. Greater parallelisation across many machines and further optimisation of the superoptimiser might allow us to consider slightly longer programs; but it's in the nature of an exponentially growing search space that significant linear improvements in efficiency lead to only small improvements in the length of program we can consider.

Other investigations

In addition to the superoptimisation of specific target functions, we carried out several experiments to analyse the behaviour of the superoptimiser itself:

1. We examined the efficacy of individual filters on programs of different lengths by generating a million random programs of each length, running each filter against each program, and keeping count of how many programs failed each filter;

2. In a further examination of filtering, we looked for overlaps between filters: i.e. which ones partially duplicated the actions of one another, at different sequence lengths. Again, this was done by generating large numbers of random programs of various lengths and looking at intersections between filter failure on each;

4. High-level system design

Platform and components

Clojure programming language

Our superoptimiser is implemented using the Clojure programming language. Clojure is a recent (released 2007) LISP implemented on top of the JVM. Several key features of the language made it attractive for this project:

1. Clojure programs run inside a JVM and can reference Java classes; in-process loading of Java classes and execution of methods is therefore straightforward, and the testing of code generated by the superoptimiser simple;
2. Lazy sequences, an abstraction which allows for infinite-length lists to be produced and consumed;
3. Good primitives for parallelisation, including a parallel map operation, `pmap`, which allows efficient processing of sequences on multi-core processors and a lightweight threading mechanism called `futures` (Hickey, 2012);
4. Being a functional language, functions are first-class citizens; this has proved particularly useful when working with lists of tests.

The implementation was created using the Eclipse IDE and the CounterClockwise (Petit, 2012) plug-in for Clojure development.

We used a number of standard Clojure libraries for the project: `ASM`, `lein-eclipse`, `noir`, `clj-http`, `tools.logging`, `clojure-csv`, `math.combinatorics` and `data.priority-map`.

ASM bytecode manipulation library

To generate Java bytecode we used ASM 4.0 (OW2 Consortium, 2012), an open source framework for bytecode manipulation. Two of the design goals of ASM are simplicity of use and speed of operation, which sat well with our desire to developer a superoptimiser in a short period of time and create hundreds of millions of Java classes.

ASM offers two APIs: a core API which provides an event-based representation of classes, and a tree API which is object-based. The latter was judged more suitable for use in this project, as it allows for simple construction of a Java method and class from a sequence of operations.

ASM is itself written in Java, and could thus be accessed directly from Clojure programs.

A component of ASM, the Bytecode Outline Plugin allows the Eclipse IDE to quickly decompile and analyse Java classes.

Amazon EC2 for parallelisation

Towards the end of the project we explored the larger search space of longer programs by parallelising searches between multiple machines. For this we used Amazon Elastic Computing Cluster (EC2) VMs.

EC2 instances were managed through the web-based console provided by Amazon (2012), and using a third party command-line tool (Kay, 2012) to create large numbers of VMs.

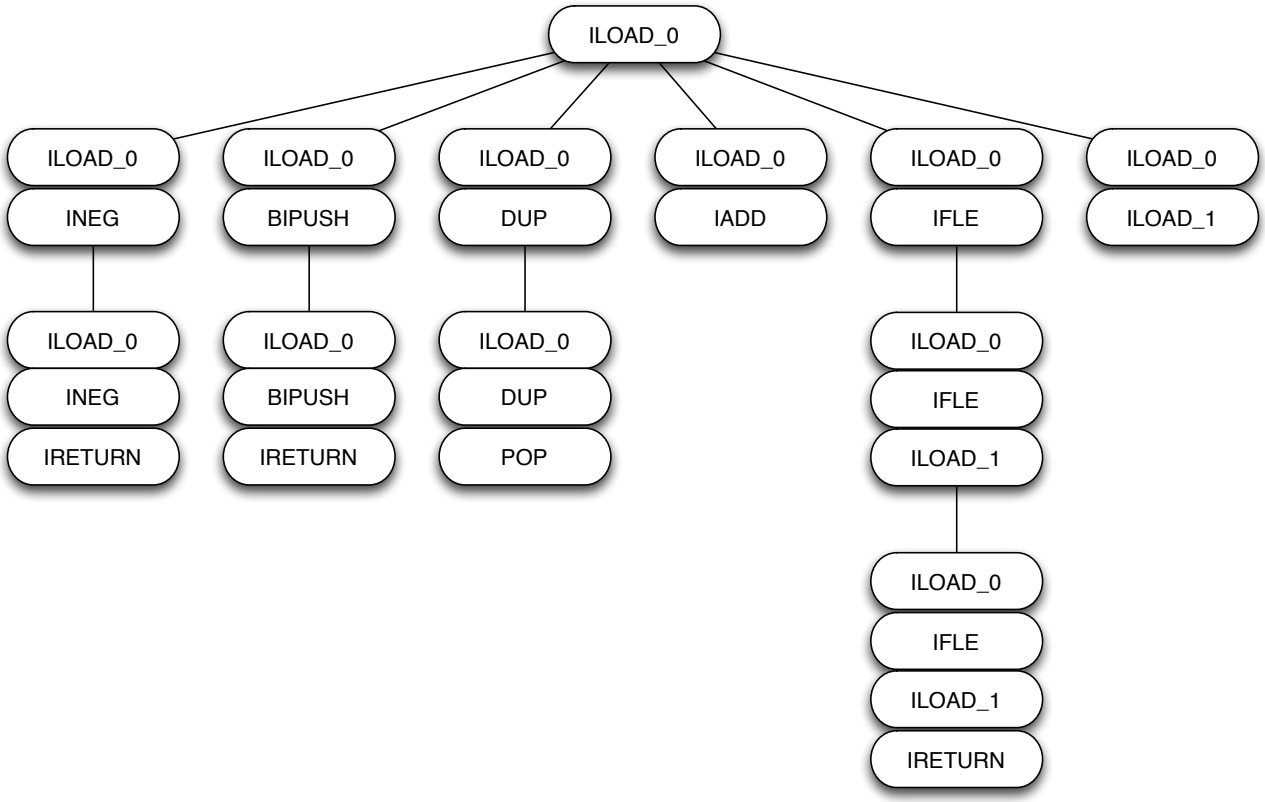
Logging between EC2 instances was facilitated using the `syslog4j` library (Yunke, 2012).

Key abstractions

Visualising the search space

An early version of the superoptimiser generated candidate sequences naively by taking the cartesian product of all permitted opcodes. It quickly became clear that exploring larger programs in this way would take too long, and that a method which considers every possibility, no matter how quickly, is inferior to a method which safely avoids considering some part of the search space at all.

We therefore consider the space of possible programs as a tree, with each node a sequence of Java opcodes. Children of node N contain sequences prefixed by the sequence at N, thus:



This tree is traversed to produce a lazy sequence of Java programs which can be consumed by the rest of the superoptimiser. Our first implementation traversed the tree breadth-first, to guarantee that the first match found is the shortest. The memory requirements of breadth-first traversal proved overwhelming, so we moved to a depth-first search, capped to a specific depth (this being the maximum sequence length). We now rely on completing the whole search before drawing conclusions from the results, as with depth-first traversal it is possible to find a longer result before a shorter one.

Fertility and validity filters

To minimise the number of programs considered (without discounting potentially optimal ones) we prune the tree of possibilities using two sorts of filter:

1. *Fertility filters* are used to determine whether the children of a given node in the tree can ever be valid. If a node is infertile, its opcodes cannot ever be the prefix of an optimal JVM program, so we can avoid considering its children;
2. *Validity filters* are used to determine whether a given node is itself a program worth considering.

All filters are effectively performing static analysis on sequences of JVM opcodes.

A given node might be fertile but not valid: e.g. one whose sequence lacks a final `IRETURN`.

A node might be valid but not fertile: e.g. one whose sequence ends in an `IRETURN`.

Given that the sequence of every valid node must end in an `IRETURN`, and no fertile node can end in an `IRETURN`, no node can be both valid and fertile.

A node which is both invalid and infertile might be one whose operations lead to a stack underflow early in execution. In this example it can never result in a valid JVM program, no matter how many subsequent operations are added to it.

Filters encode axioms

Our filters are derived from simple axioms concerning the programs we are generating:

1. Such programs must be optimal, and thus contain no redundant operations;
2. They must be valid JVM programs, e.g. pass all verification tests which a Java `ClassLoader` might apply when loading them);
3. They must adhere to more general characteristics of useful programs, e.g. their output must depend on their input(s).

Each filter is implemented in a separate source file within the `Filters` package. The available filters are:

1. `InfluenceFilter`, which tracks the influence of program inputs on individual entries in the operand stack and local variables, across each operation in the program and through to a final `IRETURN`. If the output of a program is not affected by its input, it will fail this filter. `InfluenceFilter` is a validity filter. The sequence (`ILOAD_0 BIPUSH IRETURN`) would fail the `InfluenceFilter`, because the value returned will always be the constant pushed in the second operation, no matter what argument is supplied;

2. `OperandStackFilter`, which checks for stack underflows. Each Java opcode requires a certain minimum number of entries to be available on the stack; in the event of too few stack entries being available, the program is invalid. `OperandStackFilter` is both a fertility and validity filter. (`ILOAD_0 IADD`) would fail the `OperandStackFilter` because the `IADD` operation requires that there be two entries on the stack to add, and after the initial `ILOAD_0` there is only one;

3. `RedundancyFilter`, which looks for repeated states in our program (state being the combination of contents of local variables and the operand stack). Given that the superoptimiser generates only acyclic programs which are naturally deterministic, the repetition of any state is an indication of redundancy, and therefore sub-optimality. `RedundancyFilter` is both a fertility and validity filter. (`ILOAD_0 DUP POP`) fails the `RedundancyFilter` because after the `POP`, the stack is in an identical state to that which it is in after the `ILOAD_0`, and no local variables have been affected since that operation;

4. A second function inside the `Opcodes` file, `contains-no-redundant-pairs?`, looks for specific pairs of operations which are themselves redundant, e.g. multiplying by 1 (`(ICONST_1 IMUL)`) or adding zero (`(ICONST_0 IADD)`). We consider this function part of the `RedundancyFilter` in our discussions, and it too is both a fertility and validity filter;

5. `ReturnFilter` contains two simple filters: one to identify whether a program finishes with an `IRETURN` operation, and another to test whether it contains no `IRETURN` operations. The former is a validity filter, the latter a fertility filter;

6. `StackHeightFilter`, which verifies that the size of the operand stack is equal at either of the two possible destinations of a comparison instruction. This filter is unusual in that it can only be applied to a sequence of opcodes once arguments to instructions have been filled in (as branch destinations are arguments). It is a validity filter. (`ILOAD_0 IFLE` 2 `ILOAD_1 IRETURN`) fails the `StackHeightFilter`, as the stack can have height of 1 or 0 at the final `IRETURN`, depending on whether the comparison with zero in the second opcode passes or fails;

7. `VariableUseFilter`, which checks that variables are only read from once they have been written, and are not written to twice without an intervening read. In the latter case, the first write would be redundant, and the program therefore suboptimal. `VariableUseFilter` is both a fertility and validity filter; for the latter, it additionally checks that when a sequence ends, no variable has been written to without being subsequently read (another wasted write, indicating sub-optimality). The sequence (`ILOAD_0 ILOAD_1`), in a program with only a single argument, would fail the `VariableUseFilter` as the `ILOAD_1` reads from a variable which has never been written.

Each filter was individually unit tested, unit tests appearing in its source file. In total the `Filters` package contains 109 unit tests.

Equivalence testing

Equivalence testing is carried out in three ways:

1. A set of tests is supplied as part of the problem definition Clojure file (see below). Each test is a function which executes the generated method on a given input and expects a given output. Thus we assume that all programs are deterministic, avoiding randomness or dependence on any external input beyond those explicitly passed in;

2. Sequences which pass these tests are reported and manually checked, either by hand or by writing custom Java code;

3. Later in the project, we needed to test large numbers of potential matches for `Signum`. The `Signum.clj` source file contains a method `check-sequences` which runs a large number (10,000+) of random integers into the Java `Integer.signum()` implementation and each of a list of supplied class maps, reporting on passes and failures of this test.

None of these methods, save perhaps careful manual testing, absolutely proves equivalence. However we are comfortable that for the target functions being examined, the combination of the above demonstrates it.

A sample problem definition, with the set of tests visible in `eq-tests-filter`, would be:

```
(let [class-name "Negate"
 method-name "neg"
 method-signature "(I)I"
 eq-tests-filter [
 (fn zero-untouched? [i]  (= 0 (invoke-method i method-name 0)))
 (fn one-to-minus-one? [i]  (= -1 (invoke-method i method-name 1)))
 (fn minus-one-to-one? [i]  (= 1 (invoke-method i method-name -1)))
 (fn large-positive? [i]  (= -1232 (invoke-method i method-name 1232)))
 (fn large-negative? [i]  (= 9873 (invoke-method i method-name -9873)))
]]
```

Early versions of the superoptimiser tried to rank tests by the numbers of failures they had experienced thus far in the program run, and run them in order of this ranking. Our intention was to ensure that the most likely test to fail would run first, and therefore that failures occur quickly to keep overall run-time to a minimum. Mid-project this was found to be slower than careful ordering, and dropped.

Flow of control

At a high level, the superoptimiser works as follows:

1. A lazy sequence L, where each entry in L is itself a sequence of Java opcodes, is generated by the `Opcodes/opcode-sequence` function;

2. Each entry E from L is analysed to work out the maximum number of local variables it might use, using `Opcodes/count-storage-ops`;

3. E is then expanded by `Opcodes/expand-opcodes` into a set of sequences by expanding every argument of every opcode within it. For instance, a branching operation may have multiple destinations; `IINC` may increment one of many local variables; `BIPUSH` may push many different bytes onto the operand stack;

4. Each of these sequences is then used to populate a class map which is numbered, with the whole process driven by `Opcodes/expanded-numbered-opcode-sequence`;

5. As each entry is consumed, a Java class is created from its sequence of opcodes and arguments, and the target function tests run against this class. Any class which passes all tests is recorded as a pass; those which fail are silently discarded.

Parallelisation and scaling

The superoptimiser needs to generate and test many programs. In order to do this we parallelise in two ways:

1. Using the primitives Clojure provides for processing sequences, (`map`, `reduce`, etc.) and a parallel mapper `pmap`, which spreads the processing of its input across multiple threads. We use `pmap` to parallelise the testing of candidate programs on a single machine, and experimented with applying it to different parts of the superoptimiser;

2. Spreading the searching procedure across many machines, to complete searches in a reasonable amount of time and apply greater resources of computing power than the laptop on which the superoptimiser was developed.

Spreading across many machines

Splitting the search space across many separate machines, and having them collaborate in a single search, was done in a very simple fashion.

Two sorts of machine were used: a single control node, and multiple worker nodes. Both were hosted on the Amazon EC2 infrastructure, the former a `t1.micro` size machine, the latter `c1.xlarge` (high-CPU) machine.

The control node ran a simple web server, started from the command-line thus:

```
lein run -m Cluster.Server 18 Drivers.Signum
```

In this case, the web server has been instructed to split the search space between 18 nodes, and have them run the `Drivers.Signum` search.

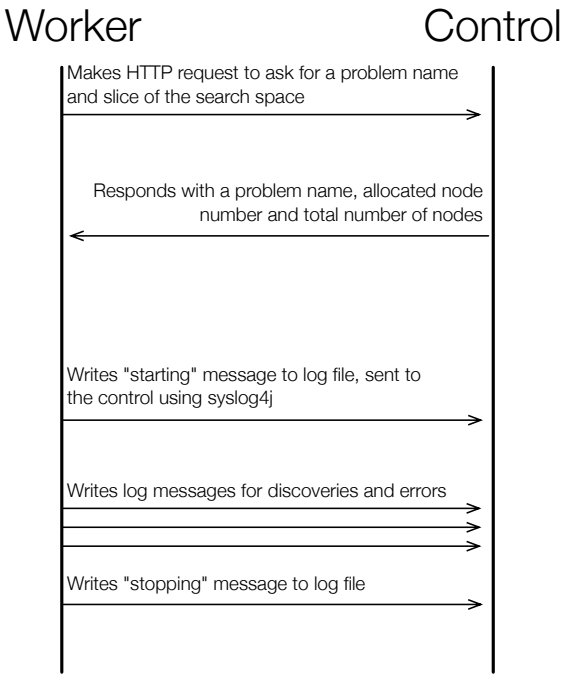
Worker nodes were started using the Amazon EC2 command-line tool, thus:

```
aws run ami-bdc06ad4 -i c1.xlarge -n 18
```

In this case, we are starting 18 instances of the machine image `ami-bdc06ad4`, each of which is to be a `c1.xlarge` VM. On start-up, each image runs the following command-line automatically (where 1.2.3.4 is the IP address of the control node):

```
lein run -m Cluster.Client http://1.2.3.4/init
```

This starts the Clojure interpreter and makes a single HTTP request to the server, which gives it a number M from 0..N-1, where N is the number of worker nodes across which the job is being split. The worker node then starts the generation of a sequence of candidate programs, beginning at M and taking every Nth candidate from then on. In this way the load is split evenly across N-1 machines.



Outline of communications between a worker node and the control node

Individual worker nodes log their discoveries using the standard Clojure tools.logging package, which is configured to use the `syslog4j` library to send all log entries back to the control node, where they are aggregated and written to disk. In this way results from all nodes are stored centrally. Worker nodes log when they have started and finished their search.

This system for parallelisation has the advantages of conceptual simplicity, reuse of pre-existing code, and speed of development (being written in less than 8 hours). It makes no provision for worker nodes experiencing issues or terminating mid-process. Whilst it would be practical to notice this from logs and manually re-start a node to look through a given slice of the search space again, in operation we never saw such a termination occur. This system can spread the load of a search more-or-less evenly. In practice we saw some nodes take a little longer than others (up to 15 minutes on searches that took more than a day to run).

5. Low-level implementation

Key data structures

Most of the work of the superoptimiser is focused around three data structures:

1. `Global/opcodes`, a map of JVM opcode names (expressed as keywords) to their characteristics. One of the more complex examples would be

```
:ifeq {
      :opcode 153
      :args [:branch-dest]
      :opstack-needs 1
      :opstack-effect -1
      :jump true
      :cjump true
}
```

   This defines the `IFEQ` opcode (number 153) as taking a single argument consisting of a branch destination. `IFEQ` needs 1 entry on the operand stack to be valid, and its net effect is to remove one entry from the operand stack. It is both a branching operation, and a conditional branch.

2. A single code sequence is expressed as a sequences of opcode-argument tuples. Initially the tuples only contain opcodes:

```
((:iload_0) (:bipush) (:ireturn))
```

   As part of the expansion of an opcode sequence in `Opcodes/expand-opcodes`, any arguments are added in, leading to structures like these:

```
((:iload_0) (:bipush 0) (:ireturn))
((:iload_0) (:bipush 1) (:ireturn))
((:iload_0) (:bipush 2) (:ireturn))
```

   etc.

3. Finally, a class map is a data structure which encapsulates a single program, together with information gathered and potentially reused during processing. An example class map would be:

```
{
      :seq-num 1981282
      :vars 2
      :length 6
      :code ((:iload_1) (:iload_0) (:dup2) (:if_icmple 2) (:swap)
(:ireturn))
      :jumps {3 5}
}
```

   This defines sequence number 1981282 as having two local variables, being 6 opcodes long, having a certain code sequence, and a single branching instruction from opcode 3 to 5 (zero-indexed, of course).

Class maps are the output of `Opcodes/expanded-numbered-opcode-sequence` and are filtered by `Superoptimise/check-passes`, to weed out those which do not pass equivalence tests for the target function.

Key algorithms

The bulk of the superoptimiser is composed of small functions which are described in their declarations, and perform highly specific tasks. These are chained together to create a sequence of class maps (see above), populate them with various fields and ultimately create and test classes based on them.

Clojure's excellent support for data structures means that the hard work of depth-first traversal of a tree of possible programs is carried out in a few lines of code, in the function `Opcodes/opcode-sequence` and `Opcodes/get-children`. These two functions traverse the tree and ensure that appropriate filters are called.

The few non-trivial and project-specific algorithms are located inside the individual filters; we explain the non-trivial filters here.

InfluenceFilter

1. Creates a map, `infl-map`, containing a map `:vars` with an entry for each local variable the program uses, and a sequence `:stack` corresponding to the stack;
2. Recurses through each operation in the program in turn:
    1. If the operation is loading a constant onto the stack (e.g. `BIPUSH`), place an entry onto `:stack` marked with `nil`, to indicate it is not influenced by any input variables;
    2. If the operation is a manipulation of two entries from `:stack` (e.g. `IADD`), take the influence values from the top two entries of the `:stack` and combine them; so if the top two entries were the first and second argument to a 2-argument function, the item at the top of `:stack` after this operation would be a single entry listing both variables;
    3. If the operation is an `ILOAD_n`, mark the top entry of `:stack` as being influenced by the value of variable number n;
    4. If it's an `ISTORE_n`, mark the value of variable number n as being the value of the top entry of `:stack`;
3. Influence of argument variables is therefore transmitted from the arguments, through the stack and intermediate local variables, until the end of the program;

4. On reaching any branching operation, we give up on determining influence; at this point we do a simple check that there is an `ILOAD_n` in the sequence for each argument - not ideal, but this catches a few cases which would otherwise fall through;

5. On reaching an `IRETURN`, look at the top entry of `:stack`. If it isn't marked as being influenced by N variables, where N is the number of arguments the program takes, return `false`; otherwise return `true`.

OperandStackFilter

The implementation of this filter is extremely simple:

1. Set a counter to zero and iterate through each operation in the input program;

2. For each operation:

    1. If the `:opstack-needs` field of the entry in `Global/opcodes` for the opcode of this operation is greater than the current value of the counter, we have an operand stack underflow, return `false`;

    2. Otherwise add the `:opstack-effect` field for the entry to the counter, and continue;

3. If we hit the end of the sequence, the input program has not suffered from stack underflow; return `true`.

RedundancyFilter

This filter maintains a sequences of past states for the program at each instruction, and each time a new instruction is reached, calculates the state of the program based on the current opcode and the previous state. State is considered to be a combination of the operand stack and local variables. The filter then compares the current state to each previous state; if there is a match, the program contains redundancy (as it has returned to a previous state) and the filter returns `false`.

Each state is a map, containing a `:stack` entry corresponding to the state of the stack (with each entry being a numbered constant, calculation or argument; `:max-var`, a maximum number of variables in play; `:max-const`, a maximum number of constants in play; `:max-calc`, a maximum number of calculations in play; and `:vars`, a map of variables to values.

So if the first operation is `ILOAD_0`, `[:arg-0]` is pushed onto the stack. If the next operation is `BIPUSH`, `[:constant 0]` is pushed onto the stack; if the next one is `IADD`, the stack becomes `[:calc-0]`; and so on.

VariableUseFilter

This filter moves through each operation in the program, tracking the last operation applied to each variable, and looking for inconsistencies. Operations are kept in a map of variable number to a keyword (either `:read` or `:write`). Absence of an entry from this map indicates that this variable hasn't been used at all.

For instance, if a variable is written to, and then written to again (with no intervening read), the first write was redundant, and the filter fails the program.

Similarly if a variable is read before it has been written to, the program cannot be valid.

`VariableUseFilter` is called with an argument indicating whether a variable which is written to, but never read, should cause the program to fail. The filter is called with this argument `false` when it's used as a fertility filter (as subsequent operations added by a child sequence might read from any given variable), and `true` when called as a validity filter (as this would indicate a pointless write, which is therefore redundant and a sign of sub-optimality).

Unit tests

Unit testing was used throughout the project to ensure that individual functions behave as expected; we found that this approach is particularly suited to a functional language like Clojure, where immutability and very limited shared state make defining inputs and outputs to individual functions straightforward. The superoptimiser is supported by around 150 such unit tests.

Unit tests are naturally executed when a source file is loaded and compiled by the Clojure runtime; failures are therefore visible whenever a superoptimisation session begins.

The standard `clojure.test` package was used for all unit testing. Unit tests in the source code are situated after the function which they test. For instance, `RedundancyFilter/constant?` Is defined and tested as follows:

```
(defn constant?
  "Is this entry from the stack a constant?"
  [e]
  (if (vector? e) (if (= :constant (first e)) true false) false))

(is (= true (constant? '[:constant 0])))
(is (= false (constant? '[:calc 0])))
```

Defining target functions

A target function is defined within a single Clojure file; see the example source code for the `Identity` function below. The structure should be fairly self-explanatory: class and method name are defined, along with the appropriately formatted method signature (Lindholm and Yellin 1999, p102). A set of functions corresponding to tests is then supplied, in the order in which they are to be run. A piece of boilerplate code at the foot of the definition drives the superoptimiser, and takes the maximum sequence length to consider as its first argument. Typically this would be set to the size of a known implementation, either hand-written or extracting from existing Java implementations of the function.

```
(ns Drivers.Identity)
(use 'Main.Superoptimise)

(let [class-name "IdentityTest"
  method-name "identity"
  method-signature "(I)I"
  eq-tests-filter [
    (fn one-is-one? [i] (= 1 (invoke-method i method-name 1)))
    (fn zero-is-zero? [i] (= 0 (invoke-method i method-name 0)))
    (fn minus-one-is-minus-one? [i] (= -1 (invoke-method i method-name
-1)))
    (fn minint-is-minint? [i] (= Integer/MIN_VALUE (invoke-method i
method-name Integer/MIN_VALUE)))
    (fn maxint-is-maxint? [i] (= Integer/MAX_VALUE (invoke-method i
method-name Integer/MAX_VALUE)))
    (fn one-is-not-zero? [i] (not (= 1 (invoke-method i method-name
0))))
    (fn one-is-not-minus-one? [i] (not (= 1 (invoke-method i method-name
-1)))) ]]
     (defn -main []
       (time (doall
           (superoptimise-pmap 2 class-name method-name method-signature
eq-tests-filter)))))
```

Source code structure

The source code for the superoptimiser is divided into the following 5 packages:

1. `Cluster`, which contains the `Client` and `Server` components used to split a large superoptimisation between many separate machines;
2. `Drivers`, within which each file encapsulates a single function to be superoptimised. Current drivers are `Abs.clj`, `Identity.clj`, `Max.clj`, `Min.clj`, `Negate.clj` and `Signum.clj`;
3. `Filters`, in which individual files contain the various fertility and validity filters, used for pruning the search space;
4. `Main`, which contains files for generating Java bytecode (`Bytecode.clj`), sequences of Java opcodes (`Opcodes.clj`), generic functions and shared data structures (`Global.clj`) and functions that handle running the above and performing equivalence testing (`Superoptimise.clj`);
5. `Tests`, which contains various temporary Clojure scripts used at various points during development to test ideas or functionality, but which are not required for operation of the superoptimiser.

All source code was stored in a local Git repository. Feature branches were used to develop significant individual features or to carry out exploratory or experimental work, with tested features being merged back into the master branch.

Pass-forward filtering

Later in the project, when working to improve the performance of the superoptimiser, we noted an inefficiency in the algorithms underlying filtering.

Each filter operates on a sequence of opcodes, such as:

```
((:iload_0) (:bipush) (:ireturn))
```

Typically a filter examines each opcode in a program in turn and evaluates it, building a data structure as it goes. For instance, the `OperandStackFilter` maintains a count

corresponding to the current height of the stack: after `ILOAD_0` it is 1, after the `BIPUSH` 2, and so on.

Given that the tree of possible opcode sequences is explored depth-first, in order to be applying the `OperandStackFilter` to

```
((:iload_0) (:bipush) (:ireturn))
```

…one must already have applied it to the sequence

```
((:iload_0) (:bipush))
```

…And indeed, to

```
((:iload_0))
```

By the time we evaluate the 3-long program, the processing of `ILOAD_0` has therefore happened 3 times and the `BIPUSH` twice. This is a clear inefficiency, and one we attempted to address by having filters save their current state into the class map. On starting, a filter would read from the state encoded into the class map by their parent, and only need to process the last operation added.

Whilst this seemed to offer an obvious theoretical advantage, in practice benchmarking at a few different sequence lengths (up to 4) showed it to be significantly slower. We believe this is because:

1. The cost savings would only be seen for the children of sequences which pass all fertility filters. Our filters remove many sequences, making these cost savings infrequent;
2. There is a cost associated with saving filter state into a map and reloading it, and this cost is in aggregate greater than that of the recalculations done in our original, naive method.

Performance of the system

Execution times for the superoptimiser increase exponentially in correspondence with the maximum length of the sequence being sought; this is, quite naturally, a property of doing an exhaustive search over an exponentially growing search space.

Given that we are testing Java programs by executing them against known inputs, some component of the superoptimiser must be written in Java. The superoptimiser is developed in

Clojure, a LISP which runs atop the JVM. Whilst Clojure affords many advantages, one might expect an implementation written for a virtual machine to perform worse than one written, say, in C.

Running a Clojure program from the command-line incurs the start-up penalty of loading and initialising a JVM. Whilst for very simple searches (e.g. for the `Identity` function) this is a significant percentage of overall run-time, for larger sequences the start-up penalty quickly becomes negligible in the context of overall execution time.

We have demonstrated that the length of time the superoptimiser takes to enumerate and count all possibilities in the search space of a given length sequence is dwarfed by the length of time taken to test and execute candidate programs. The bulk of execution time is therefore being spent doing the unavoidable work of loading and running Java classes, where the performance impact of using Clojure is minimal. We do not believe that the choice of Clojure has negatively impacted the overall performance of the system.

Underlying both the superoptimiser and the testing process is a JVM, on which all code runs. The performance of all aspects of this project therefore depends on the performance characteristics of the JVM implementation.

Profiling and optimisation

At a couple of points throughout the project, effort was put into improving the performance of the system. This was driven either by prior experience of where inefficiencies might lie (backed up with simple performance measurements like timing of functions), or by using a JVM profiler like YourKIt (YourKit, 2012) to examine the superoptimiser during a run and note which functions were taking up most run-time, and would thus benefit from attention.

In particular we looked at:

1. Loading of classes; the superoptimiser creates, loads and executes many classes - billions, in some cases. In order to avoid overloading a single Java `ClassLoader` with this many classes, or risking loaded classes affecting one another, an early implementation of the system instantiated a new `ClassLoader` for each generated class, and discarded this `ClassLoader` once the class had been loaded and tested. Analyses of performance showed that this led to a significant slowdown in operation. Subsequently

we began to use a single `ClassLoader` (`Bytecode/classloader`) which is re-instantiated every 50,000 times a class is loaded (in `Bytecode/get-class`). Early experiments suggested a 65,536 limit on the total number of classes loaded into the JVM, leading us to choose to re-instantiate the `ClassLoader` every 50,000: the remaining 15,536 classes being useful for Clojure or Java;

2. Class generation; the ASM library is used to create Java classes, and we investigated reusing a single ASM `ClassWriter` many times, to create many different classes. Benchmarking suggested that this made no measurable difference to overall performance, confirming the claims of efficiency made by the authors of ASM;

3. At a few places in the source code, we were testing whether an opcode was a branching instruction using an `is-jump?` function, which compared it to those opcodes which are jumps. We replaced this with a test for a `:jump` flag in the `Global/opcodes` data structure, which led to a significant 18% saving in run-time of the `Opcodes/expanded-numbered-opcode-sequence` function;

4. Both in equivalence testing and in the application of filters, a sequence of tests are run against an input and the first failure returns `false`. We therefore found that in both cases ordering the tests/filters made a difference to overall performance. In the case of equivalence testing, having early tests which return values unlikely to be generated by programs (i.e. not 0, 1, –1, etc.) results in earlier tests failing sooner;

5. In a late attempt to narrow down the likely search space for a 6 or greater long `Signum` implementation, we required that programs start by reading their argument (with an `ILOAD_0`) and finish with an `IRETURN`. Whilst recognising that this would mean that search was no longer exhaustive, it seemed likely that a program might be found with these characteristics;

6. Finally, we attempted to reduce the search space still further to allow a search for an 8-long `Signum` implementation by limiting the set of possible arguments for a signed and unsigned byte, to powers of 2, powers of 2 less 1, 0, maximum, and minimum values.

Cyclic code, threading and futures

The programs produced by the superoptimiser are acyclic: whilst forward branches are possible, backward branches are not permitted. At one point during the project, we considered allowing backward branches to permit loops. This would then introduce the possibility of infinite loops, which would need to be detected and terminated. Given the NP-hardness of the halting problem, running candidate sequences containing loops in a separate thread with a timeout seemed like the best strategy.

Clojure provides a primitive for such work, the `future` function (Hickey, 2012). In testing it became apparent that whilst a `future`-based approach would allow timeouts to be detected, cancelling a `future` would not, counter to expectations, stop an executing thread. This led to vast numbers of executing threads piling up and eventually causing the superoptimiser to hang.

We experimented with Java `ThreadGroups` to explicitly cancel the threads associated with a `future`, and found that a 2000ms timeout would be necessary to avoid false positives in detecting hanging programs. It became apparent that the additional time penalty of adding this timeout on a significant percentage of programs being tested rendered a run of the superoptimiser unacceptably slow. We therefore regretfully abandoned cyclic code and restricted the superoptimiser to forward branches only.

6. Results

Introduction

To recap, our target functions for superoptimisation were:

1. `Identity`: returns its only argument;
2. `Negate`: returns the negative of its argument;
3. `Max`: returns the greater of its two arguments;
4. `Min`: returns the lesser of its two arguments;
5. `Abs`: returns the absolute of its argument;
6. `Signum`: returns 1 if the argument is positive, –1 if negative, and 0 if 0.

Note that in the sections below that the trivial (first two) target functions were timed on a Macbook Air laptop, and the remainder times on one or more EC2 virtual machines.

Superoptimised versions of target functions

Identity

`Drivers.Identity` returned the following result in 0.521s when run on a Macbook Air:

```
{:seq-num 0, :vars 1, :length 2, :code ((:iload_0) (:ireturn)), :jumps
{}}
```

This matched a trivially obvious hand-written version of the identity function.

Negate

`Drivers.Negate` returned the following result in 2.173s when run on a Macbook Air:

```
{:seq-num 258, :vars 1, :length 3, :code ((:iload_0) (:ineg)
(:ireturn)), :jumps {}}
```

This matched a trivially obvious hand-written version of the negation function.

Max

`Drivers.Max` returned the following results in 23,214.592s when run on a single `c1.xlarge` EC2 instance, or 9,074s when run across 15 `c1.xlarge` EC2 instances:

```
{:seq-num 1980759, :vars 2, :length 6, :code ((:iload_1) (:iload_0)
(:dup2) (:if_icmplt 2) (:swap) (:ireturn)), :jumps {3 5}}

{:seq-num 1981282, :vars 2, :length 6, :code ((:iload_1) (:iload_0)
(:dup2) (:if_icmple 2) (:swap) (:ireturn)), :jumps {3 5}}

{:seq-num 37053343, :vars 2, :length 6, :code ((:iload_0) (:iload_1)
(:dup2) (:if_icmplt 2) (:swap) (:ireturn)), :jumps {3 5}}

{:seq-num 37053866, :vars 2, :length 6, :code ((:iload_0) (:iload_1)
(:dup2) (:if_icmple 2) (:swap) (:ireturn)), :jumps {3 5}}
```

For the `Max` function a less-than and less-than-or-equals comparison are equivalent, and the order of the arguments is unimportant - explaining these four results.

It's also worth noting that the second time this program was generated, it was across 15 instances - yet the time to generate was just under half. Parallelisation didn't cut execution time of the superoptimiser proportionally to the number of worker nodes.

Min

`Drivers.Min` returned the following results in 8,933s when run across 15 `c1.xlarge` EC2 instances:

```
{:seq-num 1990018, :vars 2, :length 6, :code ((:iload_1) (:iload_0)
(:dup2) (:if_icmpgt 2) (:swap) (:ireturn)), :jumps {3 5}}

{:seq-num 1980247, :vars 2, :length 6, :code ((:iload_1) (:iload_0)
(:dup2) (:if_icmpge 2) (:swap) (:ireturn)), :jumps {3 5}}

{:seq-num 37052831, :vars 2, :length 6, :code ((:iload_0) (:iload_1)
(:dup2) (:if_icmpge 2) (:swap) (:ireturn)), :jumps {3 5}}

{:seq-num 37062602, :vars 2, :length 6, :code ((:iload_0) (:iload_1)
(:dup2) (:if_icmpgt 2) (:swap) (:ireturn)), :jumps {3 5}}
```

Abs

`Drivers.Abs` returned the following results in 5,287s when run across 10 `c1.xlarge` EC2 instances:

```
{:seq-num 77665, :vars 1, :length 5, :code ((:iload_0) (:dup) (:ifge 2)
(:ineg) (:ireturn)), :jumps {2 4}}

{:seq-num 156453, :vars 1, :length 5, :code ((:iload_0) (:iload_0)
(:ifge 2) (:ineg) (:ireturn)), :jumps {2 4}}

{:seq-num 157276, :vars 1, :length 5, :code ((:iload_0) (:iload_0)
(:ifgt 2) (:ineg) (:ireturn)), :jumps {2 4}}

{:seq-num 78488, :vars 1, :length 5, :code ((:iload_0) (:dup) (:ifgt 2)
(:ineg) (:ireturn)), :jumps {2 4}}
```

Signum

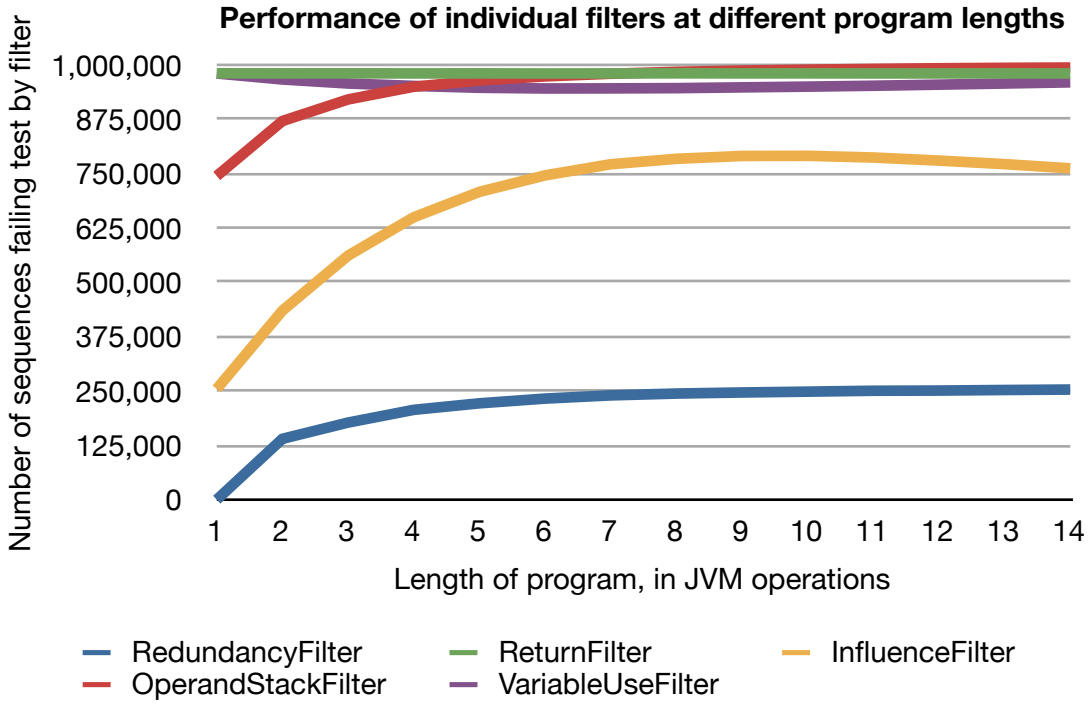
`Drivers.Signum` failed to find any results for sequences of up to length 6 (when run using the "`ILOAD_0`/`IRETURN`" optimisation) or length 5 (where our search was truly exhaustive).

A search across sequences up to 8 long, using a reduced set of possible byte arguments to opcodes (e.g. `BIPUSH`) and the `ILOAD_0`/`IRETURN` optimisation above led to 64 results, which appear in Appendix E. This search took 96,475s using 18 worker nodes.

Efficacy of filters in pruning the search space

The graph below summarises the efficacy of each of our filters when run against a million randomly generated programs of a given length. Programs were generated completely randomly, and need not have been valid Java programs: for instance, the program `(IRETURN)` would be an invalid Java program, but would be an example of a 1-long program which passes `ReturnFilter`.

Programs far longer than those we tested are included to demonstrate the likely performance of the superoptimiser given more resources than we could amass:



Overlap between filters

We compared the overlap between filters for different lengths of program. These tables summarise overlap between filters for a million random programs of different lengths. The "Any" column demonstrates overlap with any, or no other, filter.

Length 3

	Redundancy	Return	Influence	Operand Stack	Variable Use	Any
Redundancy	N/A	173774	63456	171299	172467	177370
Return	173774	N/A	542577	902633	938487	980630
Influence	63456	542577	N/A	538335	560291	561322
Operand Stack	171299	902633	538335	N/A	885669	920525
Variable Use	172467	938487	560291	885669	N/A	957188
Any	177370	980630	561322	920525	957188	N/A

Length 7

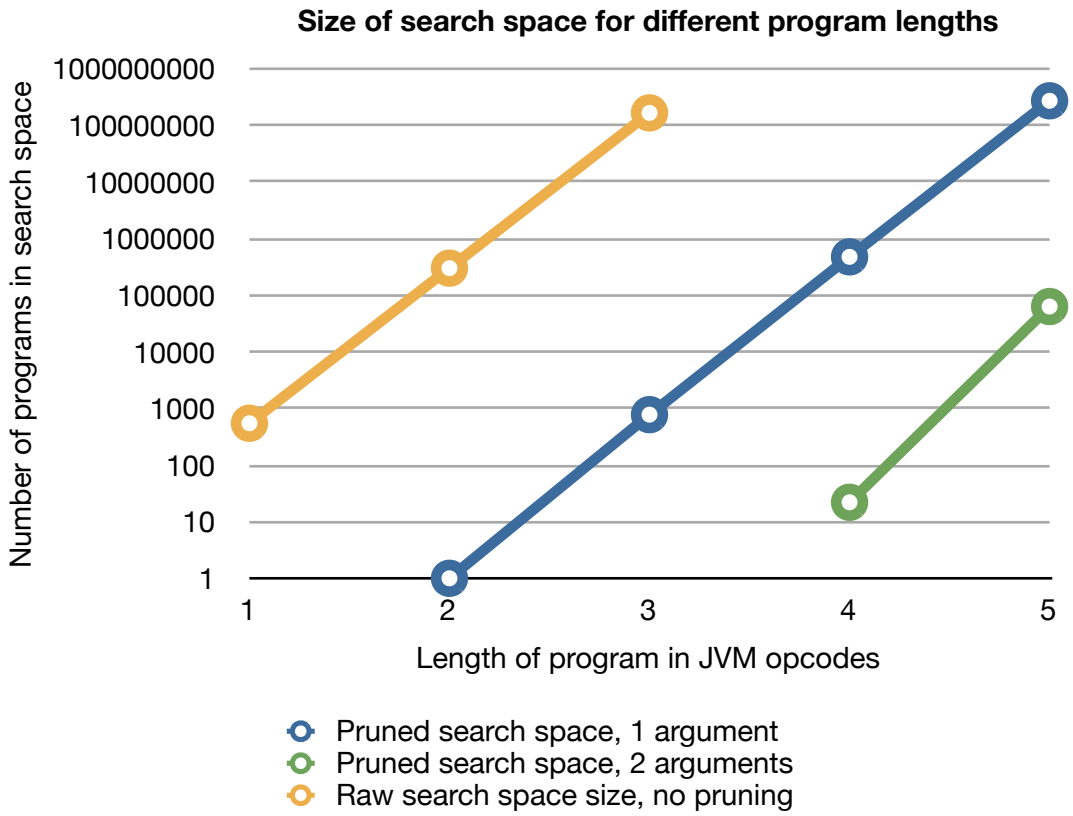
	Redundancy	Return	Influence	Operand Stack	Variable Use	Any
Redundancy	N/A	235083	166162	235955	229402	239863
Return	235083	N/A	753705	960466	927399	980379
Influence	166162	753705	N/A	761537	768753	771497
Operand Stack	235955	960466	761537	N/A	927329	979653
Variable Use	229402	927399	768753	927329	N/A	945959
Any	239863	980379	771497	979653	945959	N/A

Length 14

	Redundancy	Return	Influence	Operand Stack	Variable Use	Any
Redundancy	N/A	248282	197937	251731	244702	253309
Return	248282	N/A	746419	974372	941062	980315
Influence	197937	746419	N/A	758563	759521	762024
Operand Stack	251731	974372	758563	N/A	954069	993921
Variable Use	244702	941062	759521	954069	N/A	959884
Any	253309	980315	762024	993921	959884	N/A

Growth of search space with sequence length

During this project we generated programs with either one argument (`Identity`, `Negate`, `Abs`, `Signum`) or two (`Min`, `Max`). The graph below shows how the raw size of the search space is reduced by the cumulative effect of all our filters:

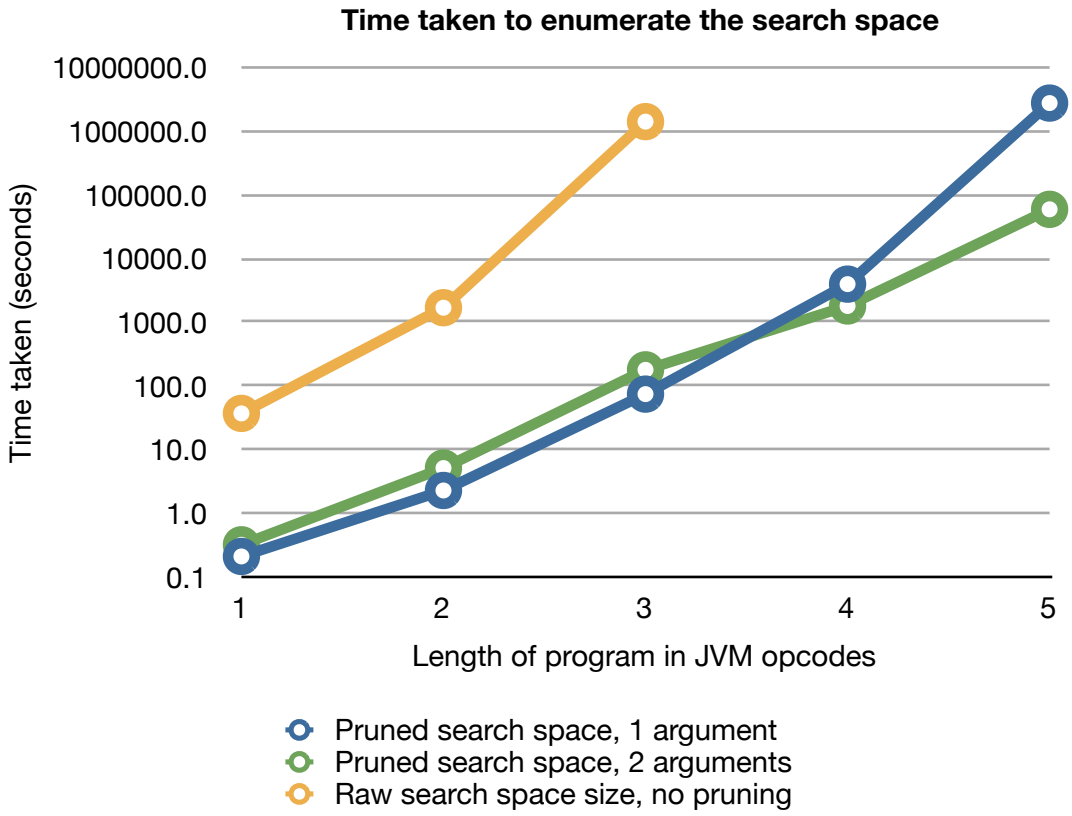


Even counting the size of the unpruned search space took an increasingly intractable amount of time, which is why we have no data for raw search space size at lengths 4 and 5.

The search space for programs with more than one argument is naturally smaller, as each argument must be read at least once, and both arguments used to calculate the output (as mandated by the `InfluenceFilter`).

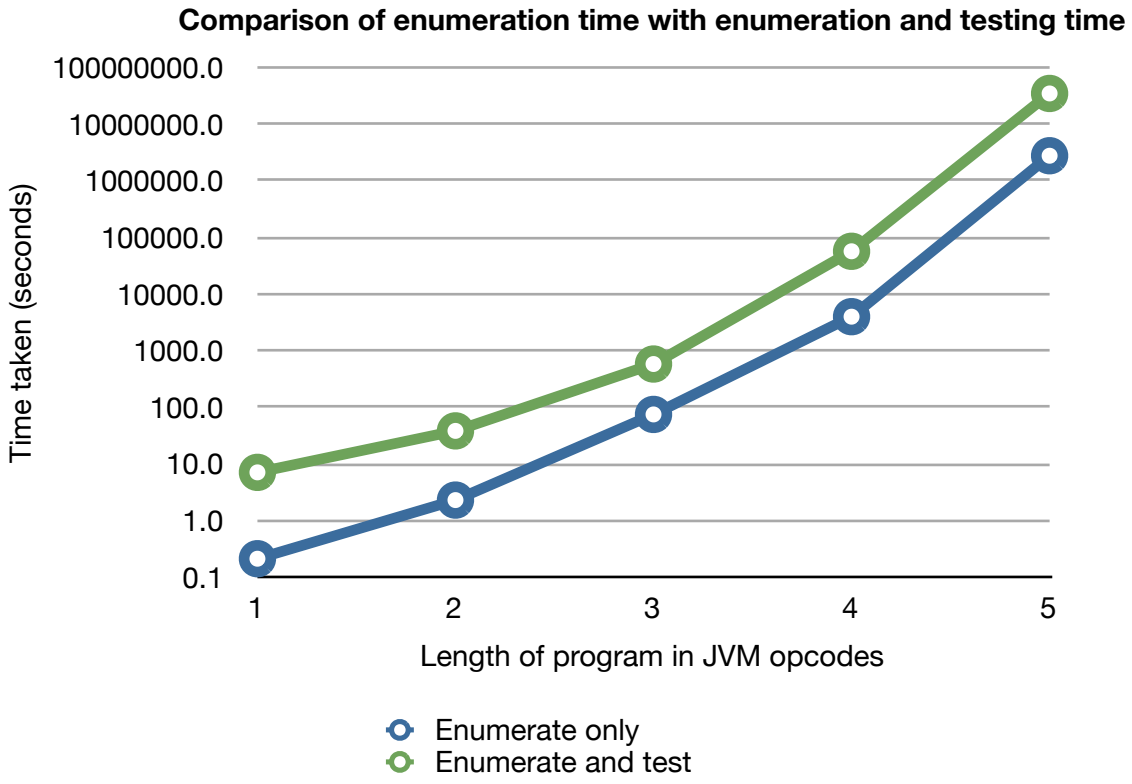
Time taken to enumerate the search space

The time taken to enumerate the search space (i.e. to generate every sequence, but not including time taken to generate classes and run tests) on a Macbook Air laptop shows similar characteristics of exponential growth:



Enumeration time vs testing time

The graph below compares the time taken to enumerate the search space to time taken to enumerate the search space, generate and test classes. We can see that generation and testing of code adds a near-fixed proportion to the overall run-time, as would be expected:



Our measurements indicated that on a single `c1.xlarge` EC2 instance, we were able to generate and test roughly 8000 Java classes/second. On the same machine we could enumerate roughly 120,000 programs a second.

Observations regarding test data

Massalin concluded that "*test vectors are chosen (manually) to maximize the probability that a random program will fail on the first or second test*" (Massalin 1986). We saw differences in overall performance of the superoptimiser with different ordering of equivalence tests. Specifically, having an early test input value of zero when generating an `Abs` program took the time-to-run from 135s up to 212s. This was partly because many programs might return zero and therefore pass the test, and partly because such an input would lead to `ArithmeticExceptions` being thrown more frequently, which seemed to slow down overall running of tests.

In general, those tests which expected unusual output values (i.e. not 0, 1 or an argument) were more likely to fail, and therefore made better candidates to be run earlier.

In an early run of the system we were excited to note some extremely short solutions for `Signum`, which it transpired was a result of poor test data, for instance:

```
{:seq-num 129377958, :vars 1, :class SignumTest, :length 4, :code
(:iload 0 :bipush 2 :irem :return)}
```

This program performs correctly when given inputs of 1001, -1001 and 0; but fails when given positive or negative even numbers (which our initial test data did not include). Another clear demonstration of the need to provide a variety of test data, in order to avoid inadvertently produce an optimal program which performs the wrong function.

7. Analysis

Comparison of superoptimised functions to Java SDK versions

Tables below show comparisons between the bytecode for the function as shipping in the JDK (Java version 1.6.0_33, running on MacOS X), the bytecode as generated by the Java compiler, and the superoptimised version of each function.

Source code for the manually compiled versions of these functions can be found in Appendix D, below.

Note that bytecode for the first two of these includes some extraneous labels, line numbers and additional data around stack size etc. A full bytecode dump of the compiled `Identity` function, for instance, looks like this:

```
L0
 LINENUMBER 6 L0
 ILOAD 0: i
 IRETURN
L1
 LOCALVARIABLE i int L0 L1 0
 MAXSTACK = 1
 MAXLOCALS = 1
```

In considering program length throughout this project, we do not include extraneous, debugging-related, or non-essential operations; and have therefore removed them from our comparison below.

In cases where the superoptimiser produced many optimal programs of the same length, we have chosen to present only one in the comparison here.

Identity

Java SDK original	Manually compiled	Superoptimised version
N/A - the identity function isn't part of the Java SDK	`ILOAD 0` `IRETURN`	`ILOAD_0` `IRETURN`

The only difference between the compiled and superoptimised version of this function is the use of the `ILOAD` operator with a 0 argument, versus the `ILOAD_0` short-hand operator.

Negate

Java SDK original	Manually compiled	Superoptimised version

N/A - the negation function isn't part of the Java SDK	`ILOAD 0`
	`INEG`
	`IRETURN`
		`ILOAD_0`
		`INEG`
		`IRETURN`

As with `Identity`, the only difference is the use of the `ILOAD_0` shorthand.

Max

Java SDK original	Manually compiled	Superoptimised version

`ILOAD 0`
`ILOAD 1`
`IF_ICMPLT L1`
`ILOAD 0`
`GOTO L2`
`L1 ILOAD 1`
`L2 IRETURN`
	`ILOAD 0`
	`ILOAD 1`
	`IF_ICMPLE L1`
	`ILOAD 0`
	`IRETURN`
	`L1 ILOAD 1`
	`IRETURN`
		`ILOAD_1`
		`ILOAD_0`
		`DUP2`
		`IF_ICMPLT L1`
		`SWAP`
		`L1 IRETURN`

The Java SDK and manually compiled versions differ slightly, the former containing two branching instructions and the latter a duplicated `IRETURN`; but the superoptimised version uses a `DUP2` opcode and a `SWAP` to save itself a single opcode.

This use of opcodes which directly manipulate the stack, beyond pushing or popping it, recurs throughout the superoptimised programs.

Min

Java SDK original	Manually compiled	Superoptimised version
ILOAD 0 ILOAD 1 IF_ICMPGT L1 ILOAD 0 GOTO L2 L1 ILOAD 1 L2 IRETURN	ILOAD 0 ILOAD 1 IF_ICMPGE L1 ILOAD 0 IRETURN L1 ILOAD 1 IRETURN	ILOAD_1 ILOAD_0 DUP2 IF_ICMPGT L1 SWAP L1 IRETURN

As one might expect, this function is in all versions a mirror of `Max`.

Abs

Java SDK original	Manually compiled	Superoptimised version
ILOAD 0 IFGE L1 ILOAD 0 INEG GOTO L2 L1 ILOAD 0 L2 IRETURN	ILOAD 0 IFLE L1 ILOAD 0 IRETURN L1 ILOAD 0 INEG IRETURN	ILOAD_0 DUP IFGE L1 INEG L1 IRETURN

Whilst the Java SDK and manually compiled version are effectively equivalent (each loading their argument onto the stack at least twice - once for comparison and once for return), the superoptimised function duplicates its argument and, having used one copy for the comparison, returns the other after perhaps negating it - thus saving two operations.

Signum

Java SDK original	Manually compiled	Superoptimised version
ILOAD 0	ILOAD 0	ILOAD_0
BIPUSH 31	IFLE L1	DUP
ISHR	ICONST_1	IFEQ L1
ILOAD 0	IRETURN	ICONST_1
INEG	L1 ILOAD 0	DUP_X1
BIPUSH 31	IFGE L2	IF_ICMPGE L2
IUSHR	ICONST_M1	L1 INEG
IOR	IRETURN	L2 IRETURN
IRETURN	L2 ICONST_0	
	IRETURN	

The Java SDK implementation of `Signum` has an elegance that other functions lack, and the feeling of having been hand-optimised by someone familiar with bitwise arithmetic. We idly wonder whether post-Massalin, implementers felt the need to deliver a strongly optimised implementation. It uses neither comparisons nor jumps and returns a result in 9 operations. My comparatively naive compiled version is slightly longer, using a more traditional couple of comparisons to do its work.

The superoptimised version is an operation shorter than even the SDK-supplied version, and uses two branches and stack manipulation (`DUP` and `DUP_X1`) to achieve the same effect. It's worth noting that the Java language does not contain primitives which would map directly to these instructions.

Other superoptimised versions of the same length demonstrated different strategies, in general substituting one of the two jumps for some bitwise arithmetic:

```
        ILOAD_0
        ICONST_M1
        ISHR
        ILOAD_0
        IFLE L1
        ICONST_M1
        ISUB
    L1  IRETURN
```

With its mix of arithmetical operations (the `ISHR`) and a single comparison, this version brings to mind a comment from Massalin regarding his findings: "*one of the most interesting results is not the programs themselves, but a better understanding of the interrelations between arithmetic and logical instructions*" (Massalin, 1987).

Overlap in filters at different program lengths

We didn't observe any patterns of interest in the data for filter overlap at different program lengths. Whilst filters vary in efficacy at different lengths, their tendency to overlap appears consistent for each pair of filters.

Projections from growth of search space

Both the pruned and unpruned search space for JVM programs grow exponentially with instruction count. The maximum length our searches reached was shorter than those recorded by Massalin, Granlund, or Bansal and Aiken.

However, our generated programs were longer: 8 opcodes for `Signum` on the JVM compared to 4 on the 68020 by Massalin (though the former includes two instructions to load the argument and return a value, which the latter lacks). It's worth noting that the 68020 is to some extent a CISC processor and the JVM RISC.

Whilst a stack-based VM like the JVM offers a lower number of possibilities per operation (compared to a machine which may have direct access to gigabytes of memory), it appears that the need to chain larger number of higher-level operations together mitigates this advantage to some extent: programs are naturally longer.

Performance of filters as sequence length grows

Techniques to prune the search space allow slightly longer sequences to become tractable for searching, but do not solve the underlying issue of exponential growth. The sum total of all our pruning of the search space was to enable searches across programs approximately 4 opcodes longer in the same time (for programs with a single argument).

The most promising filters were those which could enforce the rules of program correctness: `ReturnFilter` showed a near-constant performance whatever the length of the program being searched for; the `OperandStackFilter` tended towards slightly higher performance

for programs 7 operations and greater in length; and the `VariableUseFilter` tended downwards slightly to near-constant performance.

The `InfluenceFilter` was interesting in that it appeared to peak in performance at program of 9-10 operations, and then wane slowly. One possible explanation for this is the diffusion of influence within a program over the course of its instructions, or the simplistic way in which the `InfluenceFilter` handled comparison instructions (proceeding forwards, i.e as though passing through the comparison, and make no check of the alternative case).

Our early expectation was that the `RedundancyFilter` might be more useful than measurements suggested it was. It would appear that only a quarter of our randomly generated Java programs more than 7 operations long contain any redundancy.

Efficiency of the implementation

Given the relative times taken to enumerate the search space vs generating and testing all programs, we conclude that the latter is the most time consuming aspect of a superoptimiser run. The loading and testing of a class needs to be done inside a JVM, so we see little opportunity for improving the performance of this aspect of the superoptimiser by using a language other than Java.

It is possible to conceive of alternative routes for implementation: e.g. generating Java programs, translating these programs into machine code, and executing this code natively.

Further investigation into the costs of loading classes would seem to be prudent. In 1987 Massalin was able to superoptimise programs 12 instructions long in several hours, using a superoptimiser written in Assembler. He used a subset of the 68020 instruction set to do this, but the exact subset is not recorded in his paper - making it hard to compare the size of the search space he explored to that explored in our experiments.

The graph of “Time taken to enumerate the search space” suggests that performance worsens with sequence length. This may be an argument to revisit the pass-forward implementation of filters discussed earlier.

Cost of superoptimisation

At standard charges for Amazon EC2 instances ($0.60/hour for a `c1.xlarge` instance, $0.02/hour for a `t1.micro` instance) the costs of superoptimising our more complex target functions were as follows:

Function	Cost
`Abs`	$12.04
`Max`	$27.06
`Min`	$27.06
`Signum`	$292.14

Again, the impact of exponential growth can be seen, though a commercial case for superoptimisation might amortise the one-time cost of superoptimising these functions across the 930 million Java Runtime Environment downloads per year, or the 3 billion mobile phones that include a Java runtime (Oracle, 2012).

8. Conclusions

Our conclusions are as follows:

1. Pruning methods based on ensuring the validity of generated programs (`OperandStackFilter`, `VariableUseFilter`, `ReturnFilter`) delivered significant and useful results, but their efficacy tends towards a constant factor over time, whilst search space grows exponentially. To look for longer programs, it seems that we will require either further filters of similar or better efficacy (which do not overlap with existing filters) and/or the ability to run the superoptimiser with increased computing resources. Neither strategy offers a solution for generating arbitrary length programs;

2. As Massalin noted in his original paper, "*One of the most interesting results is not the programs themselves, but a better understanding of the interrelations between arithmetic and logical instructions*". In many of the generated programs we saw JVM stack operations used in an extremely elegant fashion to produce shorter programs (as with `Max`) or combined with arithmetic operations to produce code which is shorter than versions apparently hand-optimised by experts (as with `Signum`);

3. However, we didn't generate any programs which used unusual features of the VM. We suggest that this because a VM which conforms to a software standard and has multiple consistent implementations has fewer opportunities for obscure side-properties than a single-vendor hardware platform;

4. We discovered shorter implementations of mathematical functions than those produced by a compiler or bundled with the Java SDK, for every non-trivial function tested including `Signum`. The Massalin experiments from 1987 have been replicated for the first time in the context of a VM, and their positive results repeated.

   We therefore conclude that superoptimisation is indeed a viable approach for discovering optimal programs for virtual machines.

9. Further work

Our results and analysis from this project suggest a number of fruitful directions for future research:

1. Implementing more interesting cost functions than program length:
    1. Execution time; whilst micro-benchmarks of smaller Java programs are notoriously difficult to gather accurately (Boyer, 2008), it ought to be possible, using Aspect-Oriented Programming (Kiczales et al., 1997) or other methods, to instrument classes and compare running times;
    2. Energy efficiency; Lafond and Lilius (2007) measured and reported the energy usage of individual opcodes on two implementations of the JVM. This might be used in producing a cost function to generate programs which minimise their use of power, and be a particularly important optimisation in battery-powered devices like mobile phones;
2. Considering the run-time performance of generated programs, either as a cost function or by deriving better benchmarking capabilities for extremely short programs run inside the JVM. In particular such benchmarks might wish to take into account the underlying use of a JIT compiler in most modern JVMs, which will convert Java opcodes to machine code instructions;
3. More parallelisation; our efforts at parallelisation were very limited given the scale of computing power available today, through IAAS providers like Google and Amazon, or distributed computing efforts like Seti@Home. It would be interesting to spend more time improving on this aspect of the project and running at larger scale to look for longer programs. Bik and Gannon (1998) proposed a tool for automatic parallelisation of Java programs. One useful metric for the optimality of a program might be its suitability for use with such a tool;

4. Implement backwards loops to generate cyclic programs; this is likely to be fairly straightforward from the current code-base. It would necessitate testing cyclic programs in a time-limited separate thread, and revisiting assumptions in several filters. For instance, a sequences of opcodes may be redundant in themselves, but a later branch instruction whose destination lands in their midst may mean they nevertheless have a part to play in an optimal program;

5. Add further JVM opcodes to permit the use of arrays. Cyclic programs in conjunction with array support would allow for the generation of simple selection and sorting programs. The author suffers from an unrealistic fantasy of generating new optimal algorithms automatically in such a manner.

References

AHO, A., LAM, S., SETHI, R., and ULLMAN, J. D. (2007) *Compilers: Principles, Techniques and Tools, 2nd Edition.* Pearson Education

AMAZON (2012) *Amazon Web Services*. Available from: https://console.aws.amazon.com/console/home (accessed 23 August 2012)

BANSAL, S. and AIKEN, A. (2006) Automatic generation of peephole superoptimizers. *ACM SIGPLAN Notices*

BANSAL, S. and AIKEN, A. (2008) Binary translation using peephole superoptimizers. In: *OSDI '08: Proceedings of the 8th conference on Symposium on Operating Systems Design & Implementation* pp 177-192

BIK, A.J.C. and GANNON, D.B. (1998) A Prototype Bytecode Parallelization Tool. *Concurrency, Practice and Experience 10, 11-13,* pp. 879-886

BOYER, B. (2008) *Robust Java benchmarking, Part 1: Issues.* Available from: http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html (accessed 23 August 2012)

COUSOT, P. (1981) "Semantic Foundations of Program Analysis", in Muchnik, S.S. and Jones, N.D. *Program Flow Analysis: Theory and Applications* Prentice Hall pp. 304-346

GRANLUND, T. (1992) Eliminating branches using a superoptimizer and the GNU C compiler. *ACM SIGPLAN Notices*

HENNESSY, J.L. and PATTERSON, D.A. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann

HICKEY, R. (2012) *future.* Available from: http://clojuredocs.org/clojure_core/clojure.core/future (accessed 23 August 2012)

HUME, T.W. (2011) Literature review: Superoptimizers. Available from: https://docs.google.com/document/d/1DiyUrinBPLHfiiVGtgSig17RYy98R5_S6OuteA8qMa4/edit (accessed 7 December 2011)

JOSHI, R., Nelson, G., and Randall, K. (2002) Denali: a goal-directed superoptimizer. In: *PLDI '02 Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* New York: ACM

KAY, T. (2012) *aws - simple access to Amazon EC2 and S3 and SQS and SDB and ELB.* Available from: http://www.timkay.com/aws/ (accessed 23 August 2012)

KICZALES, G. et al. (1997) Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming*, vol.1241. pp. 220–242

KNUTH, D. E. (1970) An Empirical Study of FORTRAN Programs. *Software: Practice and Experience* 1 (2): pp. 105-133

LAFOND, S. and LILIUS, J. (2007) Energy consumption analysis for two embedded Java virtual machines. *Journal of Systems Architecture: the EUROMICRO Journal Volume 53 Issue 5-6* pp. 328-337

LINDHOLM, T., and YELLIN, F. (1999) *The Java Virtual Machine Specification 2nd edition.* California: Addison-Wesley

MASSALIN, H. (1987) Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, pp. 122-126

ORACLE (2012) *Learn About Java Technology.* Available from: http://www.java.com/en/about/ (accessed 23 August 2012)

OWS Consortium (2012) *ASM.* Available from: http://asm.ow2.org/ (accessed 23 August 2012)

SAYLE, R.A. (2008) A Superoptimizer Analysis of Multiway Branch Code Generation. In: *GCC Developers' Summit, 2008* Ottowa pp 103-116

PETIT, L. (2012) *Counterclockwise is an Eclipse plugin helping developers write Clojure code*. Available from: http://code.google.com/p/counterclockwise/ (accessed 28 August 2012)

THOMPSON, A. (1996) “An evolved circuit, intrinsic in silicon, entwined with physics”, in Higuchi, T. and Iwata, M. *Proceedings of the 1st International Conference on Evolvable Systems (ICES'96)* Springer-Verlag

TIWARI, V., MALIK, S. and WOLFE, A. (1994) Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 2, No. 4* pp. 437–445

YOURKIT (2012) *YourKit Java Profiler.* Available from: http://www.yourkit.com/overview/index.jsp (accessed 23 August 2012)

YUNKE, J. (2012) *SYslog4j: Complete Syslog Implementation for Java.* Available from: http://syslog4j.org/ (accessed 23 August 2012)

Appendix A: Supported opcodes

The following JVM opcodes are used by our superoptimiser to compose programs:

Mnemonic	Opcode #	Mnemonic	Opcode #
BIPUSH	16	IFEQ	153
DUP	89	IFNE	154
DUP_X1	90	IFLT	155
DUP_X2	91	IFGE	156
DUP2	92	IFGT	157
DUP2_X1	93	IFLE	158
DUP2_X2	94	IINC	132
POP	87	ILOAD_0	26
POP2	88	ILOAD_1	27
SWAP	95	ILOAD_2	28
IADD	96	ILOAD_3	29
IAND	126	IMUL	104
ICONST_M1	2	INEG	116
ICONST_0	3	IOR	128
ICONST_1	4	IREM	112
ICONST_2	5	IRETURN	172
ICONST_3	6	ISHL	120
ICONST_4	7	ISHR	122
ICONST_5	8	ISTORE_0	59
IDIV	108	ISTORE_1	60
IF_ICMPEQ	159	ISTORE_2	61
IF_ICMPNE	160	ISTORE_3	62
IF_ICMPLT	161	ISUB	100
IF_ICMPGE	162	IUSHR	124
IF_ICMPGT	163	IXOR	130
IF_ICMPLE	164		

Appendix B: Downloading source code

Source code for the project is available from https://github.com/twhume/superoptimiser

In this repository sit three projects:

1. `ASMExperiments`: tests and experimental Java code to exercise the ASM library;
2. `Scratchpad`: a test project for playing with ideas;
3. `Superoptimiser`: the superoptimiser, proper.

Only `Superoptimiser` is required to run the system. `ASMExperiments` may be interesting from the point of view of determining how the class file generation works.

Appendix C: Getting started

To get started with the superoptimiser:

1. Install Leinengen as per the instructions at https://github.com/technomancy/leiningen
2. Make a working directory and from the shell, do (what you type in black, sample responses in grey):

```
$ git clone https://github.com/twhume/superoptimiser.git
Cloning into 'superoptimiser'...
remote: Counting objects: 1927, done.
remote: Compressing objects: 100% (477/477), done.
remote: Total 1927 (delta 1175), reused 1927 (delta 1175)
Receiving objects: 100% (1927/1927), 6.50 MiB | 48 KiB/s, done.
Resolving deltas: 100% (1175/1175), done.
$ cd superoptimiser/SuperOptimiser/
$ lein deps
Copying 40 files to /private/tmp/superoptimiser/SuperOptimiser/lib
$ lein run -m Drivers.Identity
Aug 22, 2012 8:15:02 AM sun.reflect.NativeMethodAccessorImpl invoke0
INFO: PASS IdentityTest.identity {:seq-num 0, :vars 1, :length
2, :code ((:iload_0) (:ireturn)), :jumps {}}
"Elapsed time: 292.711 msecs"
(nil)
```

3. That's it; you've superoptimised the identity function and been shown a matching result. Now try other functions (though some may take a long time to run; edit the source code to define what length of sequence to search up to): `Drivers.Abs`, `Drivers.Negate`, `Drivers.Min`, `Drivers.Max`, or `Drivers.Signum;`
4. It should then be easy to add code to superoptimise other target functions.

Appendix D: Hand-written target functions

This appendix details source code for the hand-written versions of certain target functions, used to compare performance of the Java compiler to the superoptimiser in the Analysis section above.

Identity

```
public class Identity {

	public static int identity(int i) {
		return i;
	}

}
```

Negate

```
public class Negate {

	public static int negate(int x) {
		return -x;
	}

}
```

Min

```
public class Min {

	public static int min(int x, int y) {
		if (x<y) return x;
		return y;
	}

}
```

Max

```
public class Max {

	public static int max(int x, int y) {
		if (x>y) return x;
		return y;
	}

}
```

Abs

```
public class Abs {

	public static int abs(int x) {
		if (x>0) return x;
		return -x;
	}

}
```

Signum

```
public class Signum {

	public static int signum(int x, int y) {
		if (x>0) return 1;
		if (x<0) return -1;
		return 0;
	}

}
```

Appendix E: Results for Signum

The following are the 64 results which a search for `Signum` in sequences of up to 8 opcodes in length returned. All have been verified by probabilistic testing, passing 100,000 random integers into each and verifying that their behaviour matched that of `Integer.signum()`:

```
{:seq-num 8342841, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq
5) (:iconst_m1) (:dup_x1) (:if_icmple 2) (:ineg) (:ireturn)), :jumps {2
7, 5 7}}
```

```
{:seq-num 8342840, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq
4) (:iconst_m1) (:dup_x1) (:if_icmple 2) (:ineg) (:ireturn)), :jumps {2
6, 5 7}}
```

```
{:seq-num 8600148, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq
4) (:iconst_1) (:dup_x1) (:if_icmpge 2) (:ineg) (:ireturn)), :jumps {2
6, 5 7}}
```

```
{:seq-num 8343214, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq
5) (:iconst_m1) (:swap) (:iflt 2) (:ineg) (:ireturn)), :jumps {2 7, 5
7}}
```

```
{:seq-num 8343227, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq
4) (:iconst_m1) (:swap) (:ifle 2) (:ineg) (:ireturn)), :jumps {2 6, 5
7}}
```

```
{:seq-num 8343213, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq
4) (:iconst_m1) (:swap) (:iflt 2) (:ineg) (:ireturn)), :jumps {2 6, 5
7}}
```

```
{:seq-num 8576787, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq
3) (:iconst_5) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 5}}
```

{:seq-num 8600673, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 4) (:iconst_1) (:swap) (:ifge 2) (:ineg) (:ireturn)), :jumps {2 6, 5 7}}

{:seq-num 8601454, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 5) (:iconst_1) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 7}}

{:seq-num 8346871, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 5) (:iconst_m1) (:ishr) (:iconst_1) (:ior) (:ireturn)), :jumps {2 7}}

{:seq-num 8664699, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 3) (:iconst_3) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 5}}

{:seq-num 8600757, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 5) (:iconst_1) (:swap) (:ifgt 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 8600756, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 4) (:iconst_1) (:swap) (:ifgt 2) (:ineg) (:ireturn)), :jumps {2 6, 5 7}}

{:seq-num 8601453, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 3) (:iconst_1) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 5}}

{:seq-num 8600149, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 5) (:iconst_1) (:dup_x1) (:if_icmpge 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 8343228, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 5) (:iconst_m1) (:swap) (:ifle 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 8576788, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 5) (:iconst_5) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 7}}

{:seq-num 8600674, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 5) (:iconst_1) (:swap) (:ifge 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 8664700, :vars 1, :length 8, :code ((:iload_0) (:dup) (:ifeq 5) (:iconst_3) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 7}}

{:seq-num 28678719, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 4) (:iconst_m1) (:swap) (:iflt 2) (:ineg) (:ireturn)), :jumps {2 6, 5 7}}

{:seq-num 28936263, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_1) (:swap) (:ifgt 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 28678734, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_m1) (:swap) (:ifle 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 28678733, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 4) (:iconst_m1) (:swap) (:ifle 2) (:ineg) (:ireturn)), :jumps {2 6, 5 7}}

{:seq-num 28936262, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 4) (:iconst_1) (:swap) (:ifgt 2) (:ineg) (:ireturn)), :jumps {2 6, 5 7}}

{:seq-num 28935654, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 4) (:iconst_1) (:dup_x1) (:if_icmpge 2) (:ineg) (:ireturn)), :jumps {2 6, 5 7}}

{:seq-num 28678346, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 4) (:iconst_m1) (:dup_x1) (:if_icmple 2) (:ineg) (:ireturn)), :jumps {2 6, 5 7}}

{:seq-num 28912294, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_5) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 7}}

{:seq-num 28936960, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_1) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 7}}

{:seq-num 28936180, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_1) (:swap) (:ifge 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 28936959, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 3) (:iconst_1) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 5}}

{:seq-num 28678347, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_m1) (:dup_x1) (:if_icmple 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 29000206, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_3) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 7}}

{:seq-num 28912293, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 3) (:iconst_5) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 5}}

{:seq-num 28936179, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 4) (:iconst_1) (:swap) (:ifge 2) (:ineg) (:ireturn)), :jumps {2 6, 5 7}}

{:seq-num 29000205, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 3) (:iconst_3) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {2 5}}

{:seq-num 28678720, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_m1) (:swap) (:iflt 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 28682377, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_m1) (:ishr) (:iconst_1) (:ior) (:ireturn)), :jumps {2 7}}

{:seq-num 28935655, :vars 1, :length 8, :code ((:iload_0) (:iload_0) (:ifeq 5) (:iconst_1) (:dup_x1) (:if_icmpge 2) (:ineg) (:ireturn)), :jumps {2 7, 5 7}}

{:seq-num 43704108, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:iload_0) (:ifeq 2) (:swap) (:ifle 2) (:ineg) (:ireturn)), :jumps {3 5, 5 7}}

{:seq-num 43704101, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:iload_0) (:ifeq 2) (:swap) (:iflt 2) (:ineg) (:ireturn)), :jumps {3 5, 5 7}}

{:seq-num 44212700, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:iload_0) (:ifgt 3) (:iushr) (:dup) (:ineg) (:ireturn)), :jumps {3 6}}

{:seq-num 44213368, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:iload_0) (:ifgt 3) (:dup_x1) (:iushr) (:ineg) (:ireturn)), :jumps {3 6}}

{:seq-num 50308056, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:ishr) (:iload_0) (:ifle 3) (:iconst_1) (:ior) (:ireturn)), :jumps {4 7}}

{:seq-num 50307842, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:ishr) (:iload_0) (:ifle 3) (:iconst_m1) (:isub) (:ireturn)), :jumps {4 7}}

{:seq-num 50306874, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:ishr) (:iload_0) (:ifeq 3) (:iconst_1) (:ior) (:ireturn)), :jumps {4 7}}

{:seq-num 50308052, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:ishr) (:iload_0) (:ifle 3) (:iconst_1) (:iadd) (:ireturn)), :jumps {4 7}}

{:seq-num 50401472, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:ishr) (:iconst_1) (:iload_0) (:ifgt 2) (:swap) (:ireturn)), :jumps {5 7}}

{:seq-num 50308060, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:ishr) (:iload_0) (:ifle 3) (:iconst_1) (:ixor) (:ireturn)), :jumps {4 7}}

{:seq-num 50308009, :vars 1, :length 8, :code ((:iload_0) (:iconst_m1) (:ishr) (:iload_0) (:ifle 3) (:pop) (:iconst_1) (:ireturn)), :jumps {4 7}}

{:seq-num 125116391, :vars 1, :length 8, :code ((:iload_0) (:ineg) (:iconst_m1) (:iload_0) (:iflt 3) (:dup_x1) (:iushr) (:ireturn)), :jumps {4 7}}

{:seq-num 125116377, :vars 1, :length 8, :code ((:iload_0) (:ineg) (:iconst_m1) (:iload_0) (:iflt 3) (:iushr) (:dup) (:ireturn)), :jumps {4 7}}

{:seq-num 131298137, :vars 1, :length 8, :code ((:iload_0) (:iconst_5) (:iload_0) (:ifeq 3) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {3 6}}

{:seq-num 151251094, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:iload_0) (:ifgt 4) (:ineg) (:dup_x1) (:ishr) (:ireturn)), :jumps {3 7}}

{:seq-num 151235582, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:iload_0) (:ifgt 4) (:dup_x1) (:ineg) (:ishr) (:ireturn)), :jumps {3 7}}

{:seq-num 151251399, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:iload_0) (:ifgt 4) (:ineg) (:ishr) (:dup) (:ireturn)), :jumps {3 7}}

{:seq-num 150722749, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:iload_0) (:ifeq 2) (:swap) (:ifgt 2) (:ineg) (:ireturn)), :jumps {3 5, 5 7}}

{:seq-num 151236145, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:iload_0) (:ifgt 4) (:swap) (:iconst_m1) (:ishr) (:ireturn)), :jumps {3 7}}

{:seq-num 154533908, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:dup2) (:if_icmpge 4) (:swap) (:iconst_m1) (:ishr) (:ireturn)), :jumps {3 7}}

{:seq-num 154549162, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:dup2) (:if_icmpge 4) (:ineg) (:ishr) (:dup) (:ireturn)), :jumps {3 7}}

{:seq-num 154548857, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:dup2) (:if_icmpge 4) (:ineg) (:dup_x1) (:ishr) (:ireturn)), :jumps {3 7}}

{:seq-num 154533345, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:dup2) (:if_icmpge 4) (:dup_x1) (:ineg) (:ishr) (:ireturn)), :jumps {3 7}}

{:seq-num 150722464, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:iload_0) (:ifeq 2) (:swap) (:ifge 2) (:ineg) (:ireturn)), :jumps {3 5, 5 7}}

{:seq-num 150723336, :vars 1, :length 8, :code ((:iload_0) (:iconst_1) (:iload_0) (:ifeq 3) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {3 6}}

{:seq-num 205485111, :vars 1, :length 8, :code ((:iload_0) (:iconst_3) (:iload_0) (:ifeq 3) (:ior) (:iconst_2) (:irem) (:ireturn)), :jumps {3 6}}