SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–9
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Is superoptimization viable for VM instruction sets?

Tom Hume[1]* and Des Watson[2]

[1]*School of Engineering and Informatics, University of Sussex, Falmer, Brighton, BN1 9QJ, UK*
[2]*School of Engineering and Informatics, University of Sussex, Falmer, Brighton, BN1 9QJ, UK*

SUMMARY

The technique of superoptimization attempts to ensure true optimality of code (according to predefined criteria) through an exhaustive search of all potentially viable programs. Implementations have demonstrated that superoptimizers are capable of finding shorter programs than those hand-optimized for size by experts or produced by conventional compilers. Superoptimizers have been developed for many machine architectures, and used for diverse purposes including automating peephole optimization and binary translation of instruction sets. The output of superoptimizers is frequently surprising to human experts and often takes advantage of side-effects or obscure characteristics of the targeted hardware.
Virtual machines (VMs) are increasingly popular in implementations of programming languages, since they can provide a common platform across heterogeneous hardware architectures. This paper examines whether superoptimization could be a viable technique for VMs.
A superoptimizer for the Java Virtual Machine (JVM) has been developed and used to generate demonstrably size-optimized versions of some simple mathematical functions, implemented in Java byte code. We have shown that these versions are shorter than both implementations shipped with the Java SDK, and those generated by the Java compiler. We also have some useful observations concerning techniques to allow larger programs to be optimized in reasonable time by this approach.

Received . . .

KEY WORDS: superoptimizer; JVM; virtual machine; Java; search; code optimization

1. PREVIOUS WORK

Massalin [1] first proposed the superoptimizer, showing the generation of code for the mathematical functions (amongst others) `signum`, `max`, `min` and `abs` with target code shorter than his hand-crafted or compiler-generated versions. Five years later Granlund presented the GNU Superoptimizer [2], also generating code for specific functions but for a wider range of machine architectures. Later researchers variously addressed the DEC Alpha instruction set [3], the x86 [4, 5, 6] and SPARC V8 and MIPS R2000 [7].

Superoptimization has been used in a wide range of applications where particularly high quality code is required, such as packet checksum calculation [3], peephole optimization [4], optimizing switch statements in C compilers [6] and automated binary translation [5].

One fundamental task of a superoptimizer is to ensure that the generated programs behave correctly, producing identical results to the original non-optimized implementations. Researchers have tended to test probabilistically [1, 2, 4, 5] or use algebras and solvers; the latter to either produce provably optimal programs in the first place [3, 7] or demonstrate the equivalence of a generated solution [4, 5].

*Correspondence to: Tom Hume, c/o Google Inc, 1600 Amphitheatre Parkway, Mountain View, California 94043. Email: twhume@gmail.com

2. METHOD

To investigate the viability of superoptimization for VMs, we:

1. Developed a superoptimizer for the Java Virtual Machine.
2. Used this superoptimizer to search for optimized versions of these mathematical functions, all of which ship as part of the Java standard libraries and thus have reference implementations:
   (a) `Math.max`, which returns the greater of its two integer arguments,
   (b) `Math.min`, which returns the lesser of its two integer arguments,
   (c) `Math.abs`, which returns the absolute value of its integer argument,
   (d) `Integer.signum`, which returns -1 if its argument is negative, 1 if positive, and 0 if 0.
3. Verified the generated programs both by hand and by using probabilistic testing.
4. Compared the bytecode of generated programs to those from the Java standard library implementations, and to the bytecode generated by the Java compiler from our own Java language implementations of these functions.

For the purposes of this project we considered code size our metric for optimization: i.e. an optimal program is the shortest possible one. This permitted clear comparison with the work of Massalin (who used the same metric), and simplified our measurements of performance. Harder-to-measure metrics like run-time can be dependent on many factors outside our control (address of the program in memory, the physical environment, the way in which the JVM is implemented, etc.), though we have included some performance measurements in our results based on the metrics suggested by Maierhofer and Ertl [8].

We recognised that in an environment where bytecode is an intermediate representation (e.g. executing Java using JIT compilation), bytecode size may be a poor proxy for target code size and that longer bytecode programs could make the work of a JIT compiler easier. Nevertheless in other environments such as those where bytecode is interpreted, reductions in the size of bytecode reduce overall code size requirements.

We hoped to demonstrate the potential for superoptimization in VMs by generating shorter versions of the target functions than exist in standard libraries or are generated by a compiler. A failure to find shorter versions of any function also promised to be an interesting result, as it would demonstrate the quality of existing implementations.

3. IMPLEMENTATION DETAILS

Our superoptimizer was implemented using the Clojure programming language [9], a LISP implemented on the JVM which allows for in-process loading and execution of Java classes and enjoys useful abstractions for lazy sequences and parallelisation. We used the ASM library for generation of valid Java classes. This approach meant we could generate and test programs in-process, speeding overall execution time.

The superoptimizer was run either on a Macbook Air laptop (for shorter searches) or on up to 18 Amazon EC2 high-CPU virtual machines (for longer searches). For the latter, we divided the search space between n machines such that each machine m (for values of m from 0 to $n-1$) started at the mth entry in the search space and took every nth entry thereafter. We used the syslog4j package [10] to log and aggregate results from the worker machines.

3.1. Limiting the search space

To constrain the search space and allow easier comparison between our efforts and those of earlier researchers, we imposed these limitations on our generated programs:

- We permitted branching instructions like `IFEQ`, but only for forward branches, thus avoiding loops and possibly non-terminating programs. Multiple branches per program were allowed.

DOI: 10.1002/spe

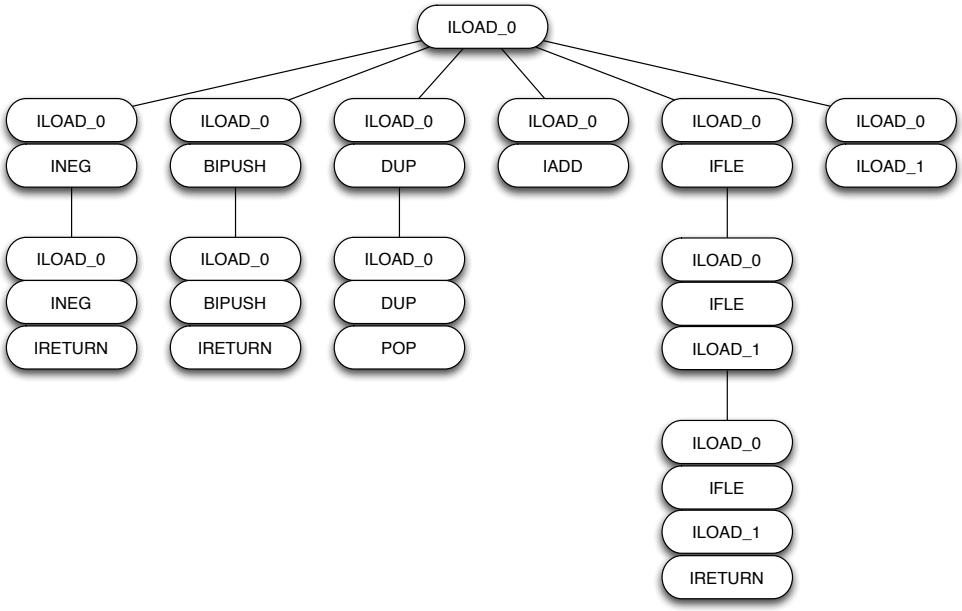

Figure 1. Search space of possible programs

- Only integer data types and operations were used (with the exception of `BIPUSH`, used to place arbitrary byte constants onto the stack). Arrays and Java objects were not permitted.
- Generated programs could make no use of the constant pool.
- No unconditional branching was allowed, either by the use of `GOTO` or by conditional branches following the push of a constant onto the operand stack.
- Programs were a maximum of 8 JVM instructions in length. This is one instruction less than the `Integer.signum()` implementation bundled with the Java SDK, the longest implementation of any of our target functions.
- Programs used a maximum of 4 local variables each. This allowed us to use the `ILOAD_n` and `ISTORE_n` shortcut instructions and avoid `ILOAD` and `ISTORE`, which reference an arbitrary number of variables. Given that we limited programs to 8 instructions, and in order to be useful a variable must be referenced twice (one read and one write), this limitation has no practical impact on our experiment.
- In searches for longer programs, we insisted that programs start by reading their argument (`ILOAD_0`) and finish by returning a value (`IRETURN`).
- For these longer searches 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.

3.2. Pruning possibilities

An early version of our superoptimizer generated candidate sequences naively by taking the cartesian product of all permitted instructions, much as Massalin had described in his original paper [1]. It quickly became clear that exploring larger programs in this way would take far too long, and that a method which considered every possibility, no matter how quickly, was inferior to a method that was capable of pruning the search space.

To allow us to safely discard portions of the search space of possible programs, we modelled it as a tree, with each node being a sequence of Java instructions. Children of node N contained sequences prefixed by the sequence at N, thus a subset of the overall space might be as shown in Figure 1.

DOI: 10.1002/spe

This tree was traversed to produce a lazy sequence of Java programs. Initially we tried traversing the tree breadth-first, to guarantee that the first match found was the shortest. The memory requirements of this approach proved overwhelming, so we switched to a depth-first search capped to a depth equal to the maximum sequence length. Whilst this was a more efficient use of memory, it allowed us to find a longer result before a shorter one, meaning our search had to complete before we could draw conclusions from the results.

To minimise the number of programs considered, without removing potential best results, we pruned the tree of possibilities using two types of filter. *Fertility filters* determined whether the children of a given node in the tree can ever be valid. If a node was infertile, its instructions could never be the prefix of an optimal JVM program, so we could avoid considering its children. *Validity filters* determined whether a node was itself a program worth considering.

All filters were effectively performing static analysis on sequences of JVM instructions. Our filters were derived from simple axioms concerning the programs we were generating in that they must be optimized, valid sequences of Java instructions, and they must adhere to general characteristics we identified of useful programs. Some filters (e.g. `StackHeightFilter`) duplicated analysis done by the Java class verifier; others (e.g. `InfluenceFilter`) were effectively stricter than the verifier. We established through measurement that it was more effective to filter programs at this stage than to have the Java `ClassLoader` reject them later by throwing a `VerifierError`.

The filters were:

1. `InfluenceFilter` ensured the output of a program depended on its input.
2. `OperandStackFilter` ensured a program did not underflow its operand stack.
3. `RedundancyFilter` ensured an acyclical program did not repeat a state during its execution.
4. `ReturnFilter` ensured a program returned a value.
5. `StackHeightFilter` ensured that both destinations of a comparative branch had an equal stack height (as mandated by the JVM).
6. `VariableUseFilter` checked that variables were neither wastefully written to (i.e. without a subsequent read, or twice in succession without an intervening read), nor read before being written to.

3.3. Checking for equivalence

We tested for equivalence in three ways:

1. Each target function was specified as a set of tests. Each test was itself a function which executed the generated program, defining an input and expected output.
2. Sequences which passed these tests were manually checked either by hand or by writing custom Java code to exercise the generated class.
3. In later experiments to find an optimized signum we submitted a large number (10,000+) of random integers to the JDK `Integer.signum` implementation and to each of our many generated classes and compared results. This would hopefully indicate any lack of equivalence.

3.4. Critique of method

It is possible that some of the pruning techniques in our filters may have overly constrained the search space. For instance, perhaps unconditional branches are more useful than we anticipated.

Our metric for optimization was code size, measured by the number of JVM instructions. In this respect we were following previous researchers, but this metric is clearly not universally appropriate. Furthermore, in programs containing branches, multiple paths may be possible, each with a different number of instructions. We do not consider optimizing for the shortest paths, but rather for shortest overall programs.

Our equivalence testing was relatively unsophisticated. None of our testing methods, save perhaps careful manual examination, absolutely guaranteed equivalence. However we were comfortable that

DOI: 10.1002/spe

for the target functions being examined, the combination of testing methods demonstrates it. More formal alternatives like abstract interpretation [11] may have a role to play here.

4. RESULTS

4.1. Generated programs

For each of `Math.max` and `Math.min` we found 4 programs, each 6 bytecode instructions long in just over 6.5 hours (on an EC2 instance), of the following form (represented as a Clojure sequence of opcode-argument pairs):

```
((:iload_1)(:iload_0)(:dup2)(:if_icmplt 2)(:swap)(:ireturn))
```

Variations read their arguments in reverse order and substituted `IF_ICMPLTE` for `IF_ICMPLT`.

For `Math.abs`, we found 4 results of length 5 in just over 1.5 hours (when run across 10 EC2 instances), of the form:

```
((:iload_0)(:dup)(:ifge 2)(:ineg)(:ireturn))
```

For `Integer.signum`, we found 64 results of length 8 in just under 27 hours, using 18 EC2 instances. These results differed in form and are discussed later.

4.2. Efficacy of filters

We pruned the search space with the set of filters outlined above and measured the effectiveness of each of these filters. Figure 2 summarises the efficacy of each filter when run against a million randomly generated programs of a given length. These need not have been valid Java programs: for instance, the program (`IRETURN`) would be an invalid Java program, and nonetheless passed by `ReturnFilter`.

The run-time cost of each of these filters was individually negligible; growth in overall run-time for the superoptimizer was proportional to the growth in the size of the search space, so we felt that detailed experiments to measure the performance of individual filters were not appropriate. Programs far longer than those we tested are included in this filter analysis to demonstrate the likely performance of the superoptimizer given more resources than we could amass.

5. ANALYSIS

5.1. Comparison of superoptimized functions to Java SDK versions

Tables I, II and III compare bytecode for the method as shipped in the JDK (Java version 1.6.0_33, running on MacOS X), bytecode as generated by the Java compiler, and our superoptimized version of each program. Program length does not include extraneous, debugging-related, or non-essential operations. In cases where the superoptimizer produced many optimized programs of the same length, we present only one here.

The Java SDK and manually compiled versions of `Math.max` differ slightly, the former containing two branching JVM instructions and the latter a duplicated `IRETURN`; but the superoptimized version uses a `DUP2` instruction and a `SWAP` to save one instruction. This use of JVM instructions which directly manipulate the stack with more sophisticated operations than `PUSH` or `POP` recurs throughout the superoptimized programs.

`Math.min` proved so similar to `Math.max` as to be unworthy of further examination.

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

Prepared using speauth.cls
DOI: 10.1002/spe

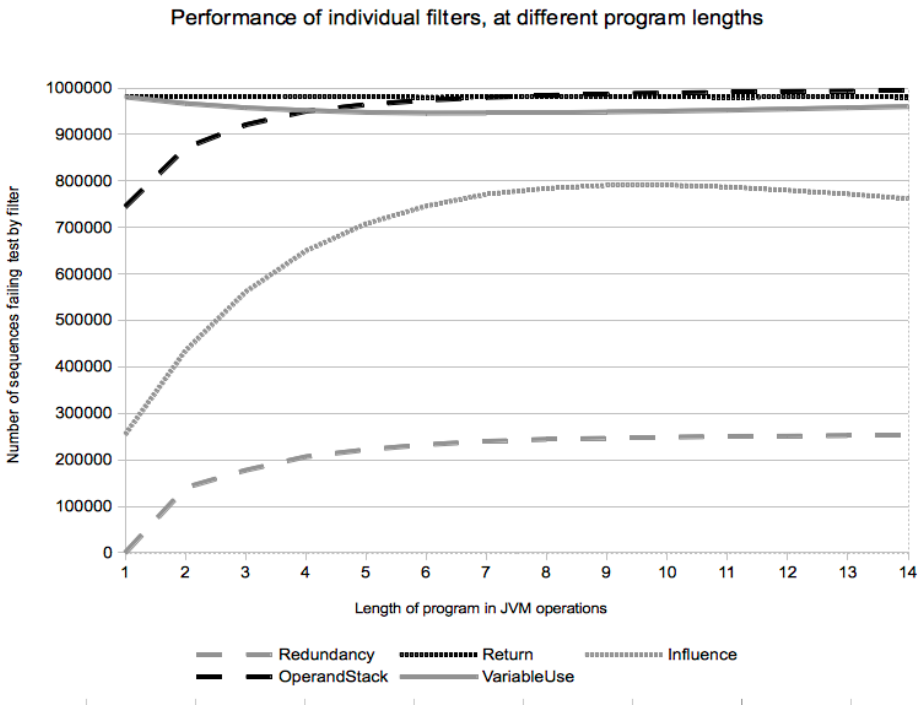

Figure 2. Performance of filters

Java SDK original	Manually compiled	Superoptimized

`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`

Table I. `Math.max`

Java SDK original	Manually compiled	Superoptimized

`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`

Table II. `Math.abs`

DOI: 10.1002/spe

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

Table III. `Integer.signum`

The Java SDK implementation of `Integer.signum` has an elegance and the feeling of having been hand-optimized by someone familiar with bitwise arithmetic. We idly wonder whether post-Massalin, implementers felt the need to deliver a strongly optimized implementation. It uses neither comparisons nor jumps and returns a result in 9 operations. The manually compiled version is slightly longer, using a more traditional pair of comparisons. The superoptimized 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. Note that the Java language does not contain primitives which would map directly to these instructions.

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

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

Using the performance measurements suggested by Maierhofer and Ertl [8] the SDK version and compiled versions of `Math.max` would each complete in 11 or 12 cycles (depending on input), and the superoptimized version in 9 or 10. For `Math.abs` the SDK version would complete in 8 or 10 cycles (depending on input), the compiled version in 9 or 8 and the superoptimized version in 7. For `Integer.signum` the SDK version would complete in 13 cycles, the compiled version in 6, 10 or 10 (depending on input) and the superoptimized version in 9 or 10.

5.2. Projections from the growth of the search space

Both the pruned and unpruned search space sizes for JVM programs grow exponentially with instruction count. Our generated programs were longer than those found by previous researchers: 8 instructions for `Integer.signum` on the JVM compared to 4 on the 68020 by Massalin (though the former includes two instructions to load the argument and one to return a value, which the latter lacks). However comparison of instruction counts between different architectures gives little indication of relative performance.

A RISC machine like the JVM offers a smaller number of possibilities for each operation, and its programs tend to involve longer sequences of operations, making superoptimization searches harder.

DOI: 10.1002/spe

Program length in JVM instructions	Search space size (unpruned)	Search space size (pruned)	Time to search in seconds (pruned)
1	548	0	0.21
2	300852	1	2.27
3	166564052	777	74.31
4	N/A	476696	3961.71
5	N/A	272190286	2760453.88

Table IV. Search space for programs of 1–5 instructions

In its favour, the JVM performs most of its operations via the stack, which results in a drastically reduced search space compared to a VM which uses direct access to memory.

5.3. 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, as Table IV (for programs with 1 argument) demonstrates.

Figure 2 shows that 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 of seven or more operations 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. Two possible explanations for this are the diffusion of influence within a program over the course of its instructions, and the simplistic way in which the `InfluenceFilter` handled comparison instructions (proceeding forwards and thus not checking the failed-comparison case).

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

6. CONCLUSIONS

Pruning methods based on ensuring the validity of generated programs (`OperandStackFilter`, `VariableUseFilter`, `ReturnFilter`) delivered significant and useful benefits, but their efficacy tended towards a constant factor over time, whilst the 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 significantly more computing resources on which to run the superoptimizer. Neither strategy scales to find arbitrary length optimized programs.

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 `Math.max`). In one example, `Integer.signum`, we saw stack and arithmetic operations combined to produce non-intuitive code which is shorter than the version shipped with the Java SDK, which appears to have been hand-optimized by experts.

We discovered shorter implementations of mathematical functions than those produced by a compiler or bundled with the Java SDK, for each of the non-trivial functions tested including `Integer.signum`. Given that we examined every viable program shorter than the implementations we have discovered, we can be sure that these implementations are optimal with respect to size.

DOI: 10.1002/spe

The Massalin experiments [1] and their positive results have been replicated, we believe for the first time, in the context of a VM. We therefore conclude that superoptimization is indeed a viable approach for discovering optimal programs for virtual machines.

7. FURTHER WORK

It would be interesting to look at other cost functions besides length, such as execution time. This would be hard to measure accurately, particularly for small programs; perhaps approaches involving Aspect-Oriented Programming [12] or instrumentation of a customised JVM may prove suitable.

We suggest parallelisation, in two senses. Firstly, it may be a route for exploring a larger search space of longer programs and secondly, the degree to which a program might be automatically parallelised [13] would be a useful metric.

Finally, we would like to see the range of supported JVM instructions extended to allow for cyclic code and operations on arrays, to allow superoptimization and discovery of simple search and sort routines.

REFERENCES

1. Massalin H. Superoptimizer – A look at the smallest program. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, Palo Alto, California, 1987; 122–126. Published as ACM SIGPLAN Notices 22:10.
2. Granlund T, Kenner R. Eliminating branches using a superoptimizer and the GNU C compiler. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, California, 1992; 341–352.
3. Joshi R, Nelson G, Randall K. Denali: a goal-directed superoptimizer. *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ACM Press, 2002; 304–314.
4. Bansal S, Aiken A. Automatic generation of peephole superoptimizers. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2006; 394–403.
5. Bansal S, Aiken A. Binary translation using peephole superoptimizers. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008; 177–192.
6. Sayle RA. A superoptimizer analysis of multiway branch code generation. *Proceedings of the GCC Developers Summit*, Ottawa, Canada, 2008; 103–116.
7. Crick T. Superoptimization: Provably optimal code generation using answer set programming. PhD thesis, University of Bath 2009.
8. Maierhofer M, Ertl MA. Local stack allocation. *Proceedings of the 7th International Conference on Compiler Construction*, CC '98, Springer-Verlag, 1998; 189–203.
9. Hickey R. Clojure 2012. URL `http://clojure.org/`, accessed 20 January 2013.
10. Yunke J. Syslog4j: Complete syslog implementation for Java 2012. URL `http://www.syslog4j.org/`, accessed 23 August 2012.
11. Cousot P. Semantic foundations of program analysis. *Program Flow Analysis: Theory and Applications*, Muchnick SS, Jones ND (eds.). chap. 10, Prentice-Hall, 1981; 303–342.
12. Kiczales G, Lamping J, Mendhekar A, Maeda C, Videira Lopes C, Loingtier JM, Irwin J. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming*, 1997; 220–242.
13. Bik AJC, Gannon DB. A prototype bytecode parallelization tool. *Concurrency: Practice and Experience* 1998; **10**(11–13):879–885.

DOI: 10.1002/spe