From Specification to Optimisation: An Architecture for Optimisation of Java Bytecode

Richard Warburton[1] and Sara Kalvala[2]

[1] University of Warwick, Coventry, UK
R.L.M.Warburton@warwick.ac.uk
[2] University of Warwick, Coventry, UK
Sara.Kalvala@warwick.ac.uk

Abstract. We present the architecture of the Rosser toolkit that allows optimisations to be specified in a domain specific language, then compiled and deployed towards optimising object programs. The optimisers generated by Rosser exploit model checking to apply dataflow analysis to programs to find optimising opportunities. The transformational language is derived from a formal basis and consequently can be proved sound. We validate the technique by comparing the application of optimisers generated by our system against hand-written optimisations using the Java based Scimark 2.0 benchmark.

1 Introduction

An optimisation phase is an integral part of most real-world compilers, and significant effort in compiler development is spent in obtaining fast-running code. This effort must be balanced with the need to ensure that optimisations do not introduce errors into programs, and the desire to not worsen compilation time significantly. Several publications, such as [8] have described the use of domain specific languages, based on temporal logic, in order to describe optimisations. Rosser allows the application of specifications of compiler optimisations to Java Bytecode. Optimisations are matched against programs using model-checking, and graph rewriting is used to actually modify the programs.

The contributions from the design of Rosser to the design of compilers include:

- An implementation that automatically generates optimisations from specifications and can be practically used against a real world programming language.
- A novel intermediate representation of Java programs, that uses BDDs to aid in symbolic model checking.
- A method of interactively and visually rewriting the control flow graph (CFG) of Java programs using the Rosser system.
- A case-study backed analysis of the performance ramifications of using model checking for dataflow analysis compared with hand-written analysers.

O. de Moor and M. Schwartzbach (Eds.): CC 2009, LNCS 5501, pp. 17–31, 2009.

The remainder of this paper is structured as follows: we summarize the TRANS language, discuss the design of Rosser, provide a brief explanation of the verification of soundness, discuss some experimental results, compare with related work, and discuss ongoing work.

2 Background and Specification Language

2.1 Specification Language Overview

The design of a specification language for optimisations needs to satisfy many constraints: the optimisations should be expressed in such a way that they are easy to understand and their correctness verified, and it should be possible to clearly express the conditions in which the optimisations apply.

In the TRANS language [7], the optimisations are represented through two components: a rewrite rule and a side condition which indicates the situations in which the rewrite can be applied safely. The specification of dead code elimination is shown in Fig. 1. Other optimisations specified in TRANS include lazy code motion, constant propagation, strength reduction, branch elimination, skip elimination, loop fusion, and lazy strength reduction; further details of these can be found in [4].

```
    n : x := e   ⇒   skip
if  ¬ EX (E ( ¬ def(x) U use(x) ∧ ¬ node(n) )) @ n
```

Fig. 1. Dead Code Elimination in TRANS

In order to use this language towards optimisation of Java programs, we replaced the original syntax for program fragments with code in `Jimple`, one of the intermediate representations used in `Soot` [18]. As such, the core of the rewrite rules is based on standard programming syntax (assignment statements, go-to and if statements, etc) and will therefore not be explained here. The syntax is expanded with a few constructs to support meta-variables, representing either syntactic fragments of the program or nodes of the CFG.

The side condition language is an extension of first order CTL, where formulae are built up from basic predicates that describe properties of states. There are two types of these basic predicates used to obtain information about a node in the control flow graph; these are the *node* and *stmt* predicates. The formula $node(x)$ will hold at a node n in a valuation that maps n to x. The formula $stmt(s)$ will hold at a node n where the valuation makes the pattern s match the statement at node n. As well as judgements about states the language can make "global" judgements. For example, the formula ϕ @ $n \ \wedge conlit(c)$ states that ϕ holds at n and c is a constant literal, throughout the program.

A logical judgement of the form: ϕ @ n states that the formula ϕ is *satisfied* at node n of the control flow graph. We base our language for expressing conditions on CTL [3], a path-based logic which can express many optimisations while still being efficient to model-check. However, we modify the logic slightly to make

it easier to express properties of programs: we include past temporal operators ($\overleftarrow{E}$ and $\overleftarrow{A}$) and extend the next state operators (EX and AX) so that one can specify what kind of edge they operate over. For example, the operators EX_{seq} and AX_{branch} stand for "there exists a next state via a *seq* edge" and "for all next states reached via a *branch* edge" respectively.

It is also possible to make use of user defined predicates via a simple macro system. These can be used in the same way as core language predicates such as *use*. They are defined by an equality between a named binding and the temporal logic condition that the predicate should be 'expanded' into.

Actions. A simple rewrite merely replaces the code at one node with new code; however, most optimisations must actually change the structure of CFGs. These structural changes are supported by four types of action: the *replace* action which replaces a node with some sequence of nodes, the *remove_edge* and *add_edge* actions which add and remove edges respectively and the *split_edge* action which inserts a node between two other nodes joined by an edge. All the actions maintain the invariant that if the *Dimple* representation can generate `Jimple` before the action has been performed, then it must do afterwards. This motivates the choice of several specific actions, rather than unrestricted graph rewriting.

Strategies. The TRANS language contains three strategies, that offer operators for combining different transformation. The MATCH ϕ IN T strategy restricts the domain of information in the transformation T by the condition ϕ. The T_1 THEN T_2 strategy applies the sequential composition of T_1 and T_2. When actions are applied normally, ambiguity with respect to what node actions and rewrites are applied to are automatically resolved. In other words, if there are several bindings that have the same value for a Node attribute that is being used in a rewrite rule then only one of them is non-deterministically selected. The APPLY_ALL T strategy uses all of the valuations within transformation T, without this restriction.

2.2 Implementation Background

Since side conditions in TRANS specifications use temporal logic, a model checking based approach is used to obtain the results of the analyses. The use of Binary Decision Diagrams (BDDs), in both model-checking and data-flow analysis applications, has significantly reduced memory consumption [12], and improved runtime performance.

To facilitate the use of a BDD representation of object programs, Rosser generates code in `Jedd`, an extension to `Java` that allows a high level representation of BDDs [11]. Relations are introduced as a primitive type within `Jedd`, and several operations, such as union, intersection, difference and comparison are defined over them. BDDs can be directly coded as relations. `Jedd` has been used as the basis for implementing inter-procedural data flow analysis [2]. The operators of `Jedd` are summarised in Table 1.

Table 1. `Jedd` operations

Operation	Comment
$x = 0B$	Assigns the empty set to relation x
$x = 1B$	Assigns the set of all possible elements to relation x
$(x =>)\ r$	projects attribute x away from relation r
$(x => y)\ r$	renames attribute x, from relation r to y
$(x => x\ y)\ r$	copies attribute x, from relation r to y
$r1\ \&\ r2$	Intersection of relations $r1$ and $r2$
$r1 \mid r2$	Union of relations $r1$ and $r2$
$r1 - r2$	Set Difference of relations $r1$ and $r2$
$r1\{x\} >< r2\{y\}$	Joins relations $r1$ and $r2$ where x equals y, projecting y
$r1\{x\} <> r2\{y\}$	Joins relations $r1$ and $r2$ where x equals y, projecting x and y

3 Architecture of Rosser

3.1 Architectural Overview

The Rosser compiler framework comprises three components. A meta-compiler, RosserC, translates TRANS specifications to produce the code for the optimising phase. Every optimisation specification is compiled into the general form of finding satisfying valuations for its side condition, by application of its side condition to the intermediate representation. The program generated (referred to as RosserS) is loaded into the runtime framework and applied to a program via the `Soot` framework.

`Soot` provides the program already translated into `Jimple`, where expressions are represented as trees, at a `Java`-like level, and control flow at a lower level utilising basic conditionals and goto statements [17]. We introduce *Dimple*—a representation equivalent to `Jimple` in overall structure, but using BDDs instead of Plain Old `Java` Objects (POJOs) in order to implement the optimisations. *Dimple* represents the relations between parents and children as `Jimple` expression trees. The translation between `Jimple` and *Dimple* is done through the RosserF framework. Only parts of the program that are relevant to the optimisations are translated, since only some components of the program need to be pattern-matched. For example since we specify no inter-procedural optimisations there is no representation of the class hierachy in our implementation.

These interactions are illustrated in Fig. 2.

3.2 Representation of Programs in *Dimple*

The *Dimple* representation introduced in this paper offers a novel approach to the intermediate representation of programs. Whilst BDDs have been used as the basis of representing sets of data during dataflow analysis [2], they haven't been used to represent entire programs before.

The type system of *Dimple* is described through several domains:

OP consists of all operators represented in `Jimple`, for example addition, and negation.

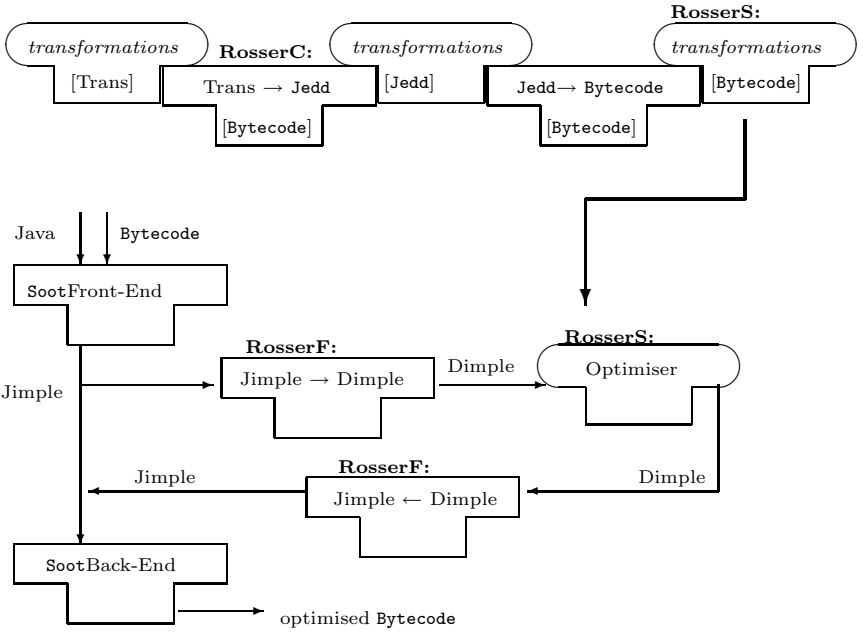

Fig. 2. Architecture of the Rosser framework

ET is the edge type domain and contains three possible values: sequential, branch and exception, and is used in pattern matching different edge.
Node lists every node in the control flow graph.
Call references every method invocation.
Value contains an entry for every possible value in the program, for example the expression $x + 1$, contains and entry for x, 1 and $x + 1$.

Table 2 summarises the translation of the syntactic components of programs in `Jimple`. Recall that `use` and `def` are predicates that hold true if their argument (a variable) is read from or written into at a given node. This fragment of the translation presents the `MustDef` relation, while similar treatment applies to `MayDef`, `MayUse` and `MustUse` relations, which all store information about different use/def chains in the same format. The next section describes these relations in more detail.

3.3 Use/def Analysis

The first phase in translation is the refinement of `use` and `def` predicates. These predicates hold true at statements where the variable that they refer to is either read or written to. In `Java`, as with other programming language, it is possible for several variables to alias the same heap object. This affects `use` and `def` predicates because it requires a refinement of the semantics to *variables that alias heap objects* that are either read or written to. Since assignment may differ

Table 2. Representing programs in *Dimple*

Name	Type	Comment/Example
`Nodes`	⟨`Node`⟩	All nodes in CFG
`Skips`	⟨`Node`⟩	All `Noop` instructions
`Edges`	⟨`Node,Node,ET`⟩	`x = 1; y = x` →⟨`x = 1, y = x, SEQ`⟩
`ReturnValues`	⟨`Node,Value`⟩	`return x;` →⟨`return x, x`⟩
`Assign`	⟨`Node,Value,Value`⟩	`y = z + 1` →⟨`y = z + 1,y,z + 1`⟩
`IfStmt`	⟨`Node,Value`⟩	`if (x==3)` →⟨`if (x==3), x==3`⟩
`Expr`	⟨`Value,Value,Value,OP`⟩	`x + y` →⟨`x + y, x, y, +`⟩
`UExpr`	⟨`Value,Value,OP`⟩	`!x` →⟨`!x, x, !`⟩
`Conlit`	⟨`Value`⟩	All Constants, eg ⟨`1`⟩
`Varlit`	⟨`Value`⟩	All Variable literals, eg ⟨`x`⟩
`CallSites`	⟨`Value,Call`⟩	Relation between call sites and values
`MustDef`	⟨`Node,Value`⟩	At `x = 3;` ⟨`x = 3,x`⟩

depending on what path through the program's control flow graph was taken, the aliasing relationship depends on this path.

As with traditional dataflow analyses [1], the approach taken in Rosser is to divide relationships into 'must' and 'may' forms. For example, `MustUse`(x) @ n, indicates that for all execution paths at node n, x is used in the computation that occurs at node n. If `MayUse`(x) @ n, then there exists an execution path such that, at node n, x is used in a computation. The situation is symmetric for `MayDef` and `MustDef`. TRANS specifications, however, do not use these conditional variants of the predicates and must be refined accordingly by RosserC.

In order to be a sound refinement of the predicates in TRANS it is necessary for the may/must variants to conservatively approximate their behaviour. That is to say, they must never enable an optimisation that would otherwise be disabled. In order to determine whether the predicate would enable incorrect optimisation we use the concept of *polarity*. The polarity of a predicate is positive if there is an even number of negations preceeding the predicate, and negative otherwise. If a predicate has a positive polarity then nodes where it holds true are being added to the possible points of optimisation. The converse also holds true. Since the `must` variant of a predicate holds true at a subset of nodes where the predicate holds true we refine predicates with a positive polarity to their `must` variants. The `may` variant of a predicate holds true at a superset of nodes where its predicate holds true, so we refine predicates with a negative polarity to the `may` variant.

This approach to refining use/def predicates means that specifications written in the TRANS language are portable over both languages that allow and disallow aliasing. This has the added advantage of facilitating prototyping of optimisations in simple contexts (for example against Local primitives, which are pass by value) and then be able to apply them in more complicated situations. This underlies one of the design principles of Rosser: to move the burden of compiler development away from the optimisation specification and into the framework.

Table 3. Temporal Logic refinements

Before	After
`AG` p	¬ `E` (true `U` ¬ p)
`A` (p `U` q)	¬ (`E` (¬ q `U` (¬ p ∧ ¬ q)] ∨ `EG` ¬ q))
`AX` p	¬ `EX` ¬ p

3.4 Refinement and Type-Checking

The CTL formulae of a specification are initially refined to a smaller set of connectives in order to simplify the output phases.

Rewrite rules are refined to a pattern matching component, which becomes part of the side condition, and a TRANS action. In the case of dead code elimination, this is the *replace* action, which swaps an existing node bound to a meta-variable, inserting an IR element generated from variable bindings in its place.

RosserC also performs type checking. The goal is to statically identify the types of all the meta variables within the TRANS specification. This is beneficial for two reasons. Firstly the output code is statically typed, and so type checking TRANS formulae helps generate object code. Secondly it is helpful in order to reduce the number of accidental or transcription errors within TRANS formulae. If a meta-variable has to bind to a structure of one type in a certain place within the specification and a different type in another part, then it is clearly not a well-formed TRANS specification. Consider the hypothetical specification:

$n : x := e \Rightarrow$ skip *if* conlit(n)

This specification fails type checking because the metavariable n has to be a node in its use on the left hand side, and a constant literal if it is an argument to `conlit`.

Fig. 3 shows the effect of refinement on the specification of dead code elimination shown in Fig. 1. Here the pattern matching has become part of the side condition and the use/def predicates have been refined.

```
   replace n with skip
if
   stmt(x := e) @ n ∧ ¬ EX (E ( ¬ maydef (x) U mustuse (x) ∧ ¬ node(n) )) @ n
```

Fig. 3. Refined specification of dead code elimination

3.5 Code Generation

The RosserC compiler outputs `Jedd` code, where for each optimisation a corresponding class is generated. The side condition is compiled into a method called `condition`, whose return type is a relation, with an attribute for each metavariable within the specification, its only parameter being the method to be optimised. A transformation is applied through method `transformation`, which in

```
1   <e,n,x,x1:N6> x2 = 1B;
2   x2 = x2{x1} >< meth.Nodes{n};
3   <e,n,x> x3 = meth.Assign;
4   x2 = x2{e,n,x} >< x3{e,n,x};
5   x2 &= (x1 => x1,x1 => n)((n => )(x2));
6   <e,n,x> x4 = (x1 => )(x2);
```

Fig. 4. Compilation of `stmt(x := e) @ n`

```
1   <e,n,x,x6:N7> x7 = 1B;
2   x7 = x7{x6,x} >< meth.MustUse{n,x};
3   <e,n,x,x6:N8> x8 = 1B;
4   x8 &= (x6 => x6,x6 => n)((n => )(x8));
5   x8 =  1B  - x8;
6   x8 = x7 & x8;
```

Fig. 5. Compilation of `mustuse(x)` $\wedge \neg$ `node(n)`

turn calls the `condition` method and then iterates over all the values within the resulting valuation set. Generating the `condition` method body proceeds by recursion of the structure of the now refined TRANS side conditions.

Fig. 4 shows the compiled pattern matching for `stmt(x := e) @ n` from the specification of dead code elimination. First, a temporary attribute `x1` is introduced into the valuation to designate the current node. This can be seen in the type of the variable `x2` on line 1. Line 2 restricts this attribute to nodes. In lines 3 and 4 the variables e, n and x are restricted to the right hand side, result variable and node of assignments, respectively. Lines 5 and 6 show the temporary node being equated to n and then projected away.

The `Jedd` code shown in Fig. 5 illustrates predicates being compiled. Line 2 shows the restriction of `mustuse` to a local finite domain. In line 4 the temporary attribute `x6`, that fulfils the same purpose as `x1` in the previous example is unified with the attribute `n`. Lines 3 and 4 calculate the set of valuations where the current node is n. Line 5 implements the $\neg$ operator, calculating valuations where the current node isn't n. Finally we take the intersection of the subcomponents, in order to satisfy the $\wedge$ in the example. Note that literals after colons, for example N6, refer to physical domains that are used by the BDD implementation. The first letter is the same as the corresponding logical domain, for example N refers to Node. Since there may be multiple physical domains for each logical domain they are numbered.

The code generation algorithm used in Rosser generates standard imperative code, using `Jedd` as its object language. The return type of the `condition` method is a relation, containing one attribute that corresponds to a TRANS metavariable in the original specification. At every stage, intermediate variables that are generated are typed as the same type as the return type. When generating conditions for node conditions, a temporal part of the condition, there is additionally an attribute that represents the current node of the specification.

```
cs true        = [res = 1B]
cs False       = [res = 0B]
cs conlit(v)   = [t1 = 1B, res = t1{v} >< Conlit{c}]
cs varlit(v)   = [t1 = 1B, res = t1{v} >< Varlit{v}]
cs ¬ φ          = cs φ @ [res = 1B - pred]
cs φ @ n       = cs φ @ [res = (at =>) pred{n,at}
                 <> pred{at,n}]
cs φ ∧ ψ       = cs φ @ cs ψ @ [res = pred1 & pred2]
cs φ ∨ ψ       = cs φ @ cs ψ @ [res = pred1 | pred2]

ct true        = [res = 1B]
ct False       = [res = 0B]
ct node(n)     = [t1=1B, res=t1{n,at} >< t1{at,n}]
ct stmt(p)     = cp p at
ct ¬ φ          = ct φ @ [res = 1B - pred]
ct EX[e] φ     = ct φ @ [t1 = Edges{et}
                 >< new{et => e}{et}, res =
                 (to=>at) pred{at} <> t1{from} ]
ct EX φ        = ct φ @ [t1 = (et=>)Edges,
                 res = (to=>at) pred{at} <> t1{from} ]
ct E[ φ U ψ ]  = ct φ @ ct ψ @ until pred1 pred2
ct φ ∧ ψ       = ct φ @ ct ψ @ [res = pred1 & pred2]
ct φ ∨ ψ       = ct φ @ ct ψ @ [res = pred1 | pred2]
```

where the `until` function is defined as:

```
until pred1 pred2 = [ t1 = (et=>) Edges,
                      acc = pred2,
                      do { prev = acc;
                           t2 = (from=>) pred1{at} <> t1{to};
                         acc |= pred2 & t2
                       } while(prev != acc),
                      res = acc  ]
```

Fig. 6. Side Condition compilation

Fig. 6 describes how side conditions are compiled. The function *cs* compiles side conditions, whilst *ct* compiles the sub-expressions within side conditions that have some temporal aspect. There are a few common attributes about the way different components within the side condition introduce new temporary variables. Basic predicates, such as `node(n)`, create new temporaries. TRANS unary operators, such as ¬, depend on the result of their inner expression, stored in a single temporary, referred to in the definition as `pred`, while binary operators, such as ∧ depend on two temporaries, `pred1` and `pred2`. All expressions store the result of their component of the model checking in a variable, referred to as `res` in the definition. Variables called `t1`, `t2` etc. refer to temporary variables within the object code of inner components. In the generated code, all these variables have disjoint names to each other, however, this is abstracted from the following section for reasons of readability. The function `cp` emits code to pattern match an expression with a sequence of nodes. Its first parameter is the pattern

```
1 while ( _it.hasNext() ) {
2   Object[] _val = (Object[]) _it.next();
3   try {
4     Unit _x = _f.SkipPattern();
5     units.swapWith(((Unit) _val[1]), _x);
6     ObjNumberer.patch(((Unit) _val[1]), _x);
7   } catch(Throwable t) {
8     System.err.println(t);
9   }
10 }
```

Fig. 7. `Jedd` code for the *replace* action

to match, and the second is the node to match it at. This is also ommitted from the presentation for reasons of brevity. The definition provides a mapping from TRANS IR to a list of `Jedd` instructions.

3.6 Action Code Generation

Fig. 7 gives the example action for code elimination, `x := e => skip`. In the code `_it` is the name of the iterator for the results set of the analysis. Line 1 shows the loop condition over this set. Line 2 shows that each element of this set is represented by an array of elements. Line 4 constructs the replacement `skip` instruction. Line 5 replaces the old instruction with the `skip` inside of `Jimple`. Line 6 replaces it within *Dimple* by renumbering the elements. Note that `_f` is a factory class for new Pattern instances.

The replacement becomes inherently simple due to the way pattern matching is refined into the side condition. Additionally to rewriting, Rosser supports the insertion and deletion of new nodes and edges. These are all implemented similarly, iterating over the elements of the finite set of valuations and replacing each element. By renumbering elements of the underlying domain that are being rewritten in simple cases such as this, rewrite rules don't need to alter the structure of the CFG at all, and thus the CFG doesn't need to be recalculated.

3.7 Interactive and Batch Mode

The RosserF runtime framework can be operated in one of two main modes: interactive or batch. The interactive mode is designed to allow the user to develop new optimisation specifications, while batch mode is a traditional compiler process that applies a list of optimisations sequentially. The interactive mode has been developed on the principle that the development of new ideas is informed by experiment. Building on this principle, interactive mode allows one to develop a specific method to apply to the program being optimised.

The interactive mode provides a *conditional* sub-view and a *transformational* sub-view. The *conditional* view provides the user with a view of the control flow graph of the selected method. The user can then enter a side condition, with which to model check the program. This then generates a set of valuations

for the given program, and a visual representation of the valuations on the control flow graph. The *transformational* view allows the user to apply complete TRANS transformations to the selected method and visually see the results, in the form of before and after control flow graphs.

The ability to allow the user to test out the effectiveness of different compiler optimisations improves the productivity of developing an effective optimisation strategy. The approach of specifying optimisations by way of a domain specific language enables the user of the system to more easily apply an optimisation, than one could with a hand-written optimisation.

4 Performance Analysis

Since this approach to generating compiler optimisations involves generating compiler code indirectly from a specification, it raises questions about its practical applicability. We compare Rosser with hand-written optimisations in the mature `Soot` framework [17], which is arguably a very high standard against which the performance of generated optimisations can be measured.

We use the Scimark scientific computing benchmark [14] to compare the performance of optimisation phases. The performances are compared in terms of *effectiveness* (the extent to which the performance of the program being optimised is improved) and *efficiency* (how long it takes to apply a transformation to a program). The benchmarking was all performed on a 2Ghz Core 2 Duo with 2GB of RAM.

The performance of three optimisations is compared: lazy code motion, common subexpression elimination and dead code elimination. In both frameworks these optimisations are applied in this order. We chose only to compare these three optimisations since they are all commonly known compiler optimisations, and are used extensively in most compilers and therefore have a large effect on performance of compilers. We have experimented with more complex optimisations as well.

4.1 Effectiveness

The two sides of Fig. 8 show the running times of the Scimark 2.0 benchmark on two different virtual machines. The three columns for each program show runtimes without any static optimisation, optimised by `Soot`, and optimised by Rosser, respectively. Since the SUN JVM already incorporates many of the optimisations that are being applied, the speedup generated by Rosser is 13.5%, comparable to `Soot` which improves performance by 14.5%. Since the implementation demonstrates that this approach to optimisation works, rather than comparing the relative merits of ahead of time and runtime optimisation, this isn't a convincing argument against our approach to optimiser generation, as the performance of Rosser is comparable to the hand-written `Soot` optimisations for this benchmark. In both cases the program with best improvement is the SOR benchmark, and in both cases it is the lazy code motion optimisation that makes the impact, since the other two optimisations are performed by the SUN JVM

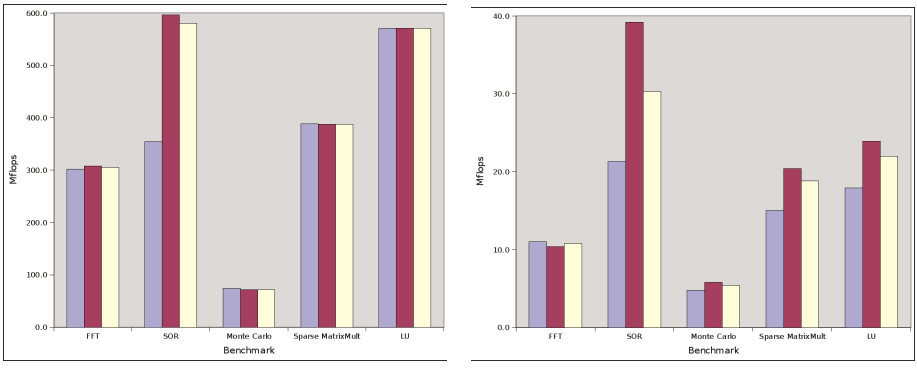

Fig. 8. Scimark 2.0 on SUN "Hotspot" JVM 1.6 and SableVM 1.13

anyway. The numbers on SableVM are more flattering to both `Soot` and Rosser, due to the more simplistic optimisations performed by the SableVM. Here Rosser improves performance by an average of 25%, while `Soot` achieves 42%. Again our generated optimisations perform slightly worse than hand-written optimisations, but offer a similar overall level of effectiveness. The `Soot` implementation of lazy code motion performs critical edge splitting before applying its optimisation, while Rosser doesn't. This might explain the difference in effectiveness.

4.2 Efficiency

The `Soot` system applied its optimisations to Scimark in 15 seconds, while Rosser took approximately 270 seconds. This does not seem very encouraging, but more detailed analysis revealed that two methods in Scimark were responsible for large amount of time used by Rosser. For the other 131 methods, the Rosser system only took 30 seconds, and the corresponding `Soot` time was 14 seconds. The two problematic methods weren't the largest within Scimark, and the nature of their pathology is currently unknown, and is being investigated. In most experiments the Rosser optimiser was about 2× slower than the hand-written `Soot` optimiser. Overall, a slowdown of a factor of two seems a reasonable price to pay considering the other benefits of the approach.

5 Related Work

Dataflow analysis has long been employed within the compiler optimisation community to iteratively compute the nodes within a program at which optimisations can be soundly applied [1,13]. Model checking is a technique in which a decision is made as to whether a given model satisfies some specification. David Schmidt and Bernhard Steffen recognised that there is a strong link between these two research areas. Equations for dataflow analyses have been shown to be expressible in Modal-Mu Calculus [15], and dataflow analysis algorithms have been generated from modal logics [16]. This approach is implemented in DFA & OPT-Metaframe [6], a toolkit designed to aid compiler construction by

generating analyses and transformations from specifications. Transformations within this system are implemented imperatively, rather than using declarative style rewrite rules, however, the temporal logic specification is converted into a model checker and then optimised. In our case, we found CTL to be sufficient to model the side conditions of transformations.

Rewrite rules with temporal conditions have also been used in the Cobalt system [9] which focuses on automated provability and also provides executable specifications, achieved through temporal conditions common to many dataflow analysis approaches. This allows the basic inductive form of the correctness theorem to be proved once and for all, given sufficient optimisation specific conditions are met. The optimisation specific proof obligations can be discharged automatically, using an automatic theorem prover. The specific nature of Cobalt's temporal conditions, while facilitating automatic discharging of proof obligations, is limited compared to the flexibility provided in TRANS from supporting CTL side conditions, even if this may require more expensive model checking.

This is the main motivation for developing Rhodium [10], another domain specific language for developing compiler optimisations. Rhodium consists of local rules that manipulate dataflow facts. This is a significant departure in approach from TRANS , since it uses more traditional, data flow analysis based specifications rather than temporal side conditions.

The Temporal Transformation Logic (TTL) [5] also uses CTL, but emphasizes verification of the soundness of the transformations themselves. Accordingly, instead of approaching optimisation as rewriting, TTL has a set of transformational primitives, each representing a common element used within compiler optimisations, for example replacing an expression with a variable. Each of the transformational primitives has an associated soundness condition that, if satisfied, implies the soundness of the transformation. TTL is presented as a specification language for other compiler implementations; on the other hand, TRANS can be refined and executed as the optimisation stage of a compiler.

6 Conclusions

While programming language theorists and compiler design scholars have often proposed methods for improving trust in the optimisations applied during compilation, there is typically a gap in putting such methodologies in practice: the practitioner may devise more adventurous optimisations, which rely on a more subtle understanding of control flow for the justification of its correctness. The semantics of TRANS have been formalized within the Isabelle/HOL theorem prover, and a proof system is currently being developed to allow most of the proof obligations of verifying the soundness of new TRANS specifications to be discharged automatically.

The Rosser system described in this paper applies compiler optimisations specified formally to Java programs within standard program development environments: the optimisations are mechanically translated into running code, and applied to given object programs within the `Soot` environment using a simple model checker for matching side conditions of optimisations to object code.

There is of course a performance price to be paid by not programing the optimiser directly. We believe this cost is minimised by actually compiling the optimisations into `Jedd` rather than interpreting them, and the benefits of a declarative approach outweigh the performance cost, as sophisticated optimisations are often applied only when the code is ready for release—which is usually not a good time to find that the optimiser has introduced new bugs. The use of a formal notation has other benefits: it aids the interactive development of new optimisations and the explanation of the optimisations to third parties.

TRANS as described doesn't allow one to specify inter-procedural optimisations. Currently we are experimenting with using the inter-procedural control flow graph, with a slightly modified TRANS that matches against blocks in addition to nodes. We are also expanding the repertoire of intra-procedural optimisations, and also deal with the vexing issue of exception handling. The formal treatment of `Java` exceptions is an ongoing research exercise.

We have extended the work of David Lacey in several ways:

- an implementation that uses a widely used, real world, programming language by way of `Java`, rather than a small research prototype language.
- an algorithm that compiles, rather than interprets, TRANS specifications.
- a novel intermediate representation for programs, using BDDs.
- use of a domain specific language for output, showing how to minimise implementation effort.

A criticism that can be made to our approach is that it relies on the correctness of the translators for the domain specific languages. For example if either of the translations from Jedd to Java source or Java source to Java bytecode are incorrect, then the entire program translation/optimisation may introduce bugs, even if our specific tool doesn't introduce bugs.

Progress made by the language semantics community must be used in solving a very practical issue, namely the development of optimisation tools which do not introduce new errors into object code. The methodology presented here makes use of model-checking to enable the deployment of complex but potentially effective optimisations in a safe manner.

Acknowledgements

Richard Warburton is funded by the EPSRC under grant EP/DO32466/1 "Verification of the optimising phase of a compiler". We are grateful to the anonymous reviewers for their detailed and helpful suggestions.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Pearson Education, London (2007)
2. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 103–114. ACM Press, New York (2003)

3. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
4. Kalvala, S., Warburton, R., Lacey, D.: Program transformations using temporal logic side conditions. Technical Report 439, Department of Computer Science, University of Warwick (2008)
5. Kanade, A., Sanyal, A., Khedker, U.: A PVS based framework for validating compiler optimizations. In: SEFM 2006: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, Washington, DC, USA, pp. 108–117. IEEE Computer Society Press, Los Alamitos (2006)
6. Klein, M., Knoop, D., Koschutzki, D., Steffen, B.: DFA & OPT-METAFrame: A toolkit for program analysis and optimization. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 422–426. Springer, Heidelberg (1996)
7. Lacey, D.: Program Transformation using Temporal Logic Specifications. PhD thesis, Oxford University Computing Laboratory (2003)
8. Lacey, D., Jones, N.D., Wyk, E.V., Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic. ACM SIGPLAN Notices 37(1), 283–294 (2002)
9. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation. ACM Press, New York (2003)
10. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 364–377. ACM Press, New York (2005)
11. Lhoták, O., Hendren, L.: Jedd: A BDD-based relational extension of Java. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation. ACM Press, New York (2004)
12. Lhoták, O., Hendren, L.: Context-sensitive points-to analysis: is it worth it? In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 47–64. Springer, Heidelberg (2006)
13. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
14. Pozo, R., Miller, B.: Java Scimark 2.0. National Institute of Standard and Technology, `http://math.nist.gov/scimark2/`
15. Schmidt, D.A., Steffen, B.: Data-flow analysis as model checking of abstract interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503. Springer, Heidelberg (1998)
16. Steffen, B.: Generating data flow analysis algorithms from modal specifications. Science of Computer Programming 21, 115–139 (1993)
17. Vallée-Rai, R., Gagnon, E., Hendren, L.J., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java bytecode using the Soot framework: Is it feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
18. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proceedings of CASCON 1999, pp. 125–135 (1999)