Automatic Validation of Code-Improving Transformations[⋆]

Robert van Engelen, David Whalley, and Xin Yuan

Dept. of Computer Science, Florida State University, Tallahassee, FL 32306-4530
{engelen,whalley,xyuan}@cs.fsu.edu

Abstract. Programmers of embedded systems often develop software in assembly code due to critical speed and/or space constraints and inadequate support from compilers. Many embedded applications are being used as a component of an increasing number of critical systems. While achieving high performance for these systems is important, ensuring that these systems execute correctly is vital. One portion of this process is to ensure that code-improving transformations on a program will not change the program's semantic behavior. This paper describes a general approach for validation of many low-level code-improving transformations made either by a compiler or specified by hand.

1 Introduction

Software is being used as a component of an increasing number of critical systems. Ensuring that these systems execute correctly is vital. One portion of this process is to ensure that the compiler produces machine code that accurately represents the algorithms specified at the source code level. This is a formidable task since an optimizing compiler not only translates the source code to machine code, it may apply hundreds or thousands of compiler optimizations to even a relatively small program. However, it is crucial to try to get software correct for many systems. This problem is exacerbated for embedded systems development, where applications are often either developed in assembly code manually or compiler generated assembly is modified by hand to meet speed and/or space constraints. Code improving transformations accomplished manually are much more suspect than code generated automatically by a compiler.

There has been much work in the area of attempting to prove the correctness of compilers [3, 4, 6]. More success has been made in the area of validating compilations rather than the compiler itself [2]. Likewise, there has been progress in proving type, memory safeness, and other related properties of a compilation [7, 8, 10]. Horwitz attempted to identify semantic differences between source programs in a simple high-level language containing a limited number of constructs [5]. In our approach we show the equivalence of the program representation before and after each improving transformation. While many code-improving transformations may be applied to a program representation, each

⋆ Supported by NSF grant CCR-9904943.

```
0. r[16]=0;
   r[17]=HI[_s];
   r[19]=r[17]+LO[_s];   r[17]:
   r[17]=r[16]+r[19];    r[16]:

1. r[17]=HI[_s]; r[16]=0;
   r[19]=r[17]+LO[_s];   r[17]:
   r[17]=r[16]+r[19];    r[16]:

2. r[19]=HI[_s]+LO[_s]; r[16]=0;
   r[17]=r[16]+r[19];    r[16]:

3. r[17]=0+HI[_s]+LO[_s]; r[19]=HI[_s]+LO[_s];
```

(a) Merging Effects in Old Region

```
0. r[16]=0;               r[16]:
   r[17]=HI[_s];
   r[19]=r[17]+LO[_s];    r[17]:
   r[17]=r[19];

1. r[17]=HI[_s];
   r[19]=r[17]+LO[_s];    r[17]:
   r[17]=r[19];

2. r[19]=HI[_s]+LO[_s];
   r[17]=r[19];

3. r[17]=HI[_s]+LO[_s]; r[19]=HI[_s]+LO[_s];
```

(b) Merging Effects in New Region

Fig. 1. Example Merging Effects within a Single Block

individual transformation typically consists of only a few changes. Also, if there is an error, then the compiler writer or assembly programmer would find it desirable for a system to identify the transformation that introduced the error. For each code-improving transformation we only attempt to show the equivalence of the region of the program associated with the changes rather than showing the equivalence of the entire program representation. We show equivalence of the region before and after the transformation by demonstrating that the effects the region will have on the rest of the program will remain the same.

2 Implementation

We validate code-improving transformations in the *vpo* compiler [1], which uses RTLs (register transfer lists) to represent machine instructions. Each register transfer is an assignment that represents a single effect on a register or memory cell of the machine. Thus, the RTL representation served as a good starting point for calculating the semantic effects of a region. Merging the RTLs in a region obtains an order-independent representation of effects. Merging also eliminates the use of temporaries within the region. Fig. 1 displays an example of merging effects. Each RTL is merged into the effects one at a time. When the destination of an effect is no longer live, then the effect is deleted (the point where a register dies is depicted to the right of that RTL). For instance, step 2 in Fig. 1(a) deletes the effect that updates `r[17]` since the register is no longer live. The final effects of the old and new regions in Fig. 1 are identical after normalization.

We automatically detect changes associated with a transformation by making a copy of the program representation before each code-improving transformation and comparing the program representation after the transformation with the copy. Each region consists of a single entry point and one or more exit points. We find the closest block in the control-flow graph that dominates all of the modified blocks. This dominating block contains the entry point of the region. A separate set of effects is calculated for each exit point from the region. The old and new regions are considered equivalent only if for each exit point they have the same effects.

Loops can be processed innermost first. The effects of each node at the same loop level are calculated only after all of its non-back edge predecessors have been processed. Merging the effects across loop nesting levels requires calculating the effects of an entire loop. One issue that we address is the represenation of a recurrence, which involves the use of a variable or register that is set on a previous loop iteration. An induction variable is one example of a simple recurrence.

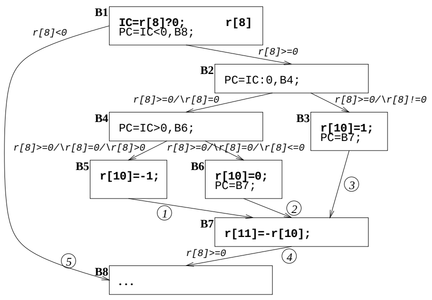
We modified the *vpo* compiler with calls to CTADEL to normalize effects. The CTADEL system [11] is an extensible rule-based symbolic manipulation program. The merging and subsequent normalization of effects results in canonical representations that enable a structural comparison to show that effects are semantically identical. The canonical representations of the effects corresponding to the exit points of old and new regions are compared by *vpo* to determine that the semantic effect of the transformed region of code is unchanged. The equivalence of the modified region is a sufficient condition for the correctness of a transformation, but it not a necessary condition.

The symbolic normalization of effects is illustrated by an example depicted in Fig. 2. The calculation of effects proceeds from the dominating block of the region (**B1**). The guarding conditions that result from conditional control flow are propagated down and used to guard RTLs. The final effects accurately describe the semantics of the region of code. Clearly, block **B5** is unreachable code. The *vpo* compiler applies dead-code elimination to remove block **B5**. The effect of the code after the transformation has been applied is unchanged (not shown). Hence, the dead-code elimination optimization is validated.

3 Results and Conclusions

A variety of types of transformations in the *vpo* compiler have been validated using our approach, including *algebraic simplification of expressions*, *basic block reordering*, *branch chaining*, *common subexpression elimination*, *constant folding*, *constant propagation*, *unreachable code elimination*, *dead store elimination*, *evaluation order determination*, *filling delay slots*, *induction variable removal*, *instruction selection*, *jump minimization*, *register allocation*, *strength reduction*, and *useless jump elimination*.

Table 1 shows some test programs that we have compiled while validating code-improving transformations. The third column indicates the number of improving transformations that were applied during the compilation of each program. The fourth column represents the percentage of transformations that we are able to validate. The only transformations that we cannot validate are those with regions that span basic blocks at different loop nesting levels since the ability to represent effects containing entire loops is in a development stage. The fifth column represents the average static number of instructions for each region associated with all code-improving transformations during the compilation. The final column denotes the ratio of compilation times when validating programs versus a normal compilation. The overhead is due to the application of sometimes tens of thousands of rewrite rules to normalize effects after a single transformation.



1. At (1) from block **B5** after merging: $\mathtt{r[10]} = \{-1\ \mathit{if}\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} > 0\}$
2. At (2) from block **B6** after merging: $\mathtt{r[10]} = \{0\ \mathit{if}\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} \leq 0\}$
3. At (3) from block **B3** after merging: $\mathtt{r[10]} = \{1\ \mathit{if}\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} \neq 0\}$
4. After combining the effects of the blocks **B5**, **B6**, and **B3** we obtain
$$\mathtt{r[10]} = \left\{\begin{array}{ll} 1 & \mathit{if}\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} \neq 0 \\ 0 & \mathit{if}\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} \leq 0 \\ -1 & \mathit{if}\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} > 0 \end{array}\right\} \overset{\text{simplify}}{=} \left\{\begin{array}{l} 1\ \mathit{if}\ \mathtt{r[8]} > 0 \\ 0\ \mathit{if}\ \mathtt{r[8]} = 0 \end{array}\right\}$$
5. Merging the above effects with the effects of block **B7** yields
$$\mathtt{r[10]} = \left\{\begin{array}{l} 1\ \mathit{if}\ \mathtt{r[8]} > 0 \\ 0\ \mathit{if}\ \mathtt{r[8]} = 0 \end{array}\right\};\ \mathtt{r[11]} = -\left\{\begin{array}{l} 1\ \mathit{if}\ \mathtt{r[8]} > 0 \\ 0\ \mathit{if}\ \mathtt{r[8]} = 0 \end{array}\right\} \overset{\text{simplify}}{=} \left\{\begin{array}{ll} -1 & \mathit{if}\ \mathtt{r[8]} > 0 \\ 0 & \mathit{if}\ \mathtt{r[8]} = 0 \end{array}\right\}$$
6. Merging the (empty) effects at transition (5) with the effects at transition (4) we obtain the effects of the region of code
$$\mathtt{r[10]} = \left\{\begin{array}{ll} \left\{\begin{array}{l} 1\ \mathit{if}\ \mathtt{r[8]} > 0 \\ 0\ \mathit{if}\ \mathtt{r[8]} = 0 \end{array}\right\} & \mathit{if}\ \mathtt{r[8]} \geq 0 \\ \mathtt{r[10]} & \mathit{if}\ \mathtt{r[8]} < 0 \end{array}\right\} \overset{\text{simplify}}{=} \left\{\begin{array}{ll} 1 & \mathit{if}\ \mathtt{r[8]} > 0 \\ 0 & \mathit{if}\ \mathtt{r[8]} = 0 \\ \mathtt{r[10]} & \mathit{if}\ r[8] < 0 \end{array}\right\}$$
$$\mathtt{r[11]} = \left\{\begin{array}{ll} \left\{\begin{array}{ll} -1 & \mathit{if}\ \mathtt{r[8]} > 0 \\ 0 & \mathit{if}\ \mathtt{r[8]} = 0 \end{array}\right\} & \mathit{if}\ \mathtt{r[8]} \geq 0 \\ \mathtt{r[11]} & \mathit{if}\ \mathtt{r[8]} < 0 \end{array}\right\} \overset{\text{simplify}}{=} \left\{\begin{array}{ll} -1 & \mathit{if}\ \mathtt{r[8]} > 0 \\ 0 & \mathit{if}\ \mathtt{r[8]} = 0 \\ \mathtt{r[11]} & \mathit{if}\ r[8] < 0 \end{array}\right\}$$

where the guard condition $\mathtt{r[8]} \geq 0$ is derived by forming the disjunction of the guard conditions on the incoming edges to block **B7**, which is the simplified form of
$(\mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} \neq 0) \vee (\mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} \leq 0) \vee (\mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} > 0)$

Fig. 2. Example Normalization of Effects

However, this overhead would probably be acceptable, as compared to the cost of not detecting potential errors. Validation can also be selectively applied on a subset of transformations to reduce the overall compilation time.

To summarize our conclusions, we have demonstrated that it is feasible to use our approach to validate many conventional code-improving transformations. Unlike an approach that requires the compiler writer to provide invariants for each different type of code-improving transformation [9], our general approach was applied to all of these transformations without requiring any special information. We believe that our approach could be used to validate many hand-specified transformations on assembly code by programmers of embedded systems.

Table 1. Benchmarks

Program	Description	#Trans	#Validated	Region	Overhead
`ackerman`	Ackerman's numbers	89	100.0%	3.18	13.64
`arraymerge`	merge two sorted arrays	483	89.2%	4.23	63.89
`banner`	poster generator	385	90.6%	5.42	34.13
`bubblesort`	bubblesort on an array	342	85.4%	6.10	34.37
`cal`	calendar generator	790	91.1%	5.16	105.64
`head`	displays the first few lines of files	302	89.4%	8.42	152.64
`matmult`	multiplies 2 square matrices	312	89.7%	5.55	28.97
`puzzle`	benchmark that solves a puzzle	1928	78.5%	5.85	128.98
`queens`	eight queens problem	296	85.8%	6.79	73.65
`sieve`	finds prime numbers	217	80.6%	6.85	21.90
`sum`	checksum and block count of a file	235	91.9%	8.62	195.19
`uniq`	filter out repeated lines in a file	519	91.1%	4.21	163.26
Average		492	88.6%	5.87	84.64

References

[1] M. E. Benitez and J. W. Davidson. A Portable Global Optimizer and Linker. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 329–338, June 1988.

[2] A. Cimatti and et. al. A Provably Correct Embedded Verifier for the Certification of Safety Critical Software. In *International Conference on Computer Aided Verification*, pages 202–213, June 1997.

[3] P. Dybjer. Using Domain Algebras to Prove the Correctness of a Compiler. *Lecture Notes in Computer Science*, 182:329–338, 1986.

[4] J. Guttman, J. Ramsdell, and M. Wand. VLISP: a Verified Implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.

[5] S. Horwitz. Identifying the Semantic and Textual Differences between Two Versions of a Program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[6] F. Morris. Advice on Structuring Compilers and Proving Them Correct. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 144–152, 1973.

[7] G. Necula. Proof-Carrying Code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[8] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[9] M. Rinard and D. Marinov. Credible Compilation with Pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verfication*, 1999.

[10] D. Tarditi, J. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[11] R. van Engelen, L. Wolters, and G. Cats. Ctadel: A generator of multi-platform high performance codes for pde-based scientific applications. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 86–93, May 1996.