# Validation of Optimizing Compilers*

L. Zuck[2], A. Pnueli[1,2], and R. Leviathan[1]

[1] Weizmann Institute of Science, Rehovot, Israel, amir@wisdom.weizmann.ac.il,
[2] New York University, New York, zuck@cs.nyu.edu

**Abstract.** There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of safety-critical components of systems. Most formal verification methods verify the correctness of a high-level representation of the system against a given specification. However, if one wishes to infer from such a verification the correctness of the code which runs on the actual target architecture, it is essential to prove that the high-level representation is correctly implemented at the lower level. That is, it is essential to verify the the correctness of the translation from the high-level source-code representation to the object code, a translation which is typically performed by a compiler (or a code generator in case the source is a specification rather than a programming language).
Formally verifying a full-fledged optimizing compiler, as one would verify any other large program, is not feasible due to its size, ongoing evolution and modification, and, possibly, proprietary considerations. The *translation validation* method used in this paper is a novel approach that offers an alternative to the verification of translators in general and compilers in particular. According to the translation validation approach, rather than verifying the compiler itself, one constructs a validation tool which, after every run of the compiler, formally confirms that the target code produced on that run is a correct translation of the source program.
The paper presents a methodology for translation validation of optimizing compilers. We distinguish between *structure preserving* optimizations, for which we establish simulation relation between source and target based on computational induction, and *structure modifying* optimizations, for which we develop specialized "meta-rules". We present some examples that illustrate the use of the methodology, including a "real-life" validation of an EPIC compiler which uncovered a bug in the compiler.

* This research was supported in part by the Minerva Center for Verification of Reactive Systems, a gift from Intel, a grant from the German - Israel Foundation for Scientific Research and Development, and ONR grant N00014-99-1-0131.

# 1 Introduction

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of safety-critical portions of systems. Most verification methods deal with the high-level specification of the system. However, if one is to prove that the high-level specification is correctly implemented at the lower level, one needs to verify the compiler which performs the translations. Verifying the correctness of modern optimizing compilers is challenging due to the complexity and reconfigurability of the target architectures and the sophisticated analysis and optimization algorithms used in the compilers.

Formally verifying a full-fledged optimizing compiler, as one would verify any other large program, is not feasible, due to its size, evolution over time, and, possibly, proprietary considerations. Translation validation is a novel approach that offers an alternative to the verification of translators in general and of compilers in particular. According to the *translation validation* approach, rather than verifying the compiler itself, one constructs a *validating tool* which, after every run of the compiler, formally confirms that the target code produced is a correct translation of the source program.

The introduction of new classes of microprocessor architectures, such as the EPIC class exemplified by the Intel IA-64 architecture, places an even heavier responsibility on optimizing compilers. This is due to the expectation that static compile-time dependence analysis and instruction scheduling could lead to instruction-level parallelism that could compete favorably with other architectures, such as those of the Pentium family, where dependences are determined and instructions are reordered at run time by the hardware. As a result, a new family of sophisticated optimizations are currently being developed and incorporated into compilers targeted at arhchitectures as the Trimaran and the SGI Pro-64 compilers.

Prior work ([PSS98a]) developed a tool for translation validation, $CVT$, which managed to automatically verify translations involving about 10,000 lines of source code in about 10 minutes. The success of this tool critically depended on some simplifying assumptions which restricted the source and target to programs with a single external loop and assumed a very limited set of optimizations.

Other approaches [Nec00,RM00] considered translation validation of less restricted languages allowing, for example, nested loops. They also considered a more extensive set of optimizations. However, the methods proposed there were restricted to *structure preserving* optimizations, and could not directly deal with more aggressive optimizations such as *loop distribution* and *loop tiling* which are used in more advanced optimizing compilers.

Our ultimate goal is to develop a methodology for the translation validation of advanced optimizing compilers, with an emphasis on EPIC-targeted compilers, and the aggressive optimizations characteristic to such compilers. Our methods will handle an extensive set of optimizations and can be used to implement fully automatic certifiers for a wide range of compilers, ensuring an extremely high level of confidence in the compiler in areas, such as safety-critical systems and compilation into silicon, where correctness is of paramount concern.

In this paper we develop the theory of a correct translation. This theory provides a precise definition of the notion of a target program being a correct translation of a source program, and the methods by which such a relation can be formally established. We distinguish between *structure preserving* optimizations which admit a clear mapping of control points in the target program to corresponding control points in the source program. Most high-level optimizations belong to this class. For such transformation, we apply the well known method of *simulation* which relates the execution between two target control points to the corresponding source execution. A more challenging class of optimizations does not guarantee such correspondence and, for this class, we have developed specialized approaches to which we refer as *meta-rules*. Typical optimizations belonging to this class are *loop distribution* and *fusion*, *loop tiling*, and *loop interchange*.

One of the side-products we anticipate from this work is the formulation of a validation-oriented instrumentation, which will instruct the writer of future compilers how to incorporate into the optimization modules appropriate additional outputs which will make validation straightforward. This will lead to a theory of construction of *self-certifying* compilers.

## 1.1 Related Work

The work here is an extension of the work in [PSS98a], and we intend to use the tools developed there in our implementation of the validating tool studied here. The work in [Nec00] covers some important aspects

of our work. For one, it extends the source programs considered from a single-loop program to C programs with arbitrarily nested loop structure. An additional important feature is that the method requires no compiler instrumentation at all, and applies various heuristics to recover and identify the optimizations performed and the associated refinement mappings. The main limitation apparent in [Nec00] is that, as is implied by the single proof method described in the report, it can only be applied to structure-preserving optimizations. Since structure-modifying optimizations, such as the ones associated with aggressive loop optimizations are a major component of optimizations for modern architectures, we make this a central element of our work.

Another related work is [RM00] which proposes a comparable approach to translation validation, where an important contribution is the ability to handle pointers in the source program. However, the method proposed there assumes *full* instrumentation of the compiler, which is not assumed here or in [Nec00].

More weakly related are the works reported in [Nec97] and [NL98], which do not purport to establish full correctness of a translation but are only interested in certain "safety" properties. However, the techniques of program analysis described there are very relevant to the automatic generation of refinement mappings and auxiliary invariants.

### 1.2 General Strategy for Translation Validation of Optimizing Compilers

The compiler receives a *source program* written in some high-level language. It translates the source code into an *Intermediate Representation (IR)* or *intermediate code*. The compiler then applies a series of optimizations to the program, starting with classical architecture-independent optimizations, and then architecture-dependent ones, such as register allocation and instruction scheduling. Typically, these optimizations are performed in several passes (up to 15 in some compilers), where each pass applies a certain type of optimization. Translation validation provides a proof for the correctness of each such optimization pass, where a successful validation results in a proof-script confirmation, and an unsuccessful validation results in a counterexample.

The general approach to establishing a correct correspondence between target and source is based on *refinement* and is proved by *simulation*. According to this approach, we establish a *refinement mapping* indicating how the relevant source variables correspond to an appropriate target variables or expressions. The proof is then broken into a set of *verification conditions* (also called *proof obligations*), each of which claiming that a segment of target execution corresponds to a segment of source execution. In some of the cases, the proof obligations are not valid by themselves, and then it is necessary to introduce *auxiliary invariants* which provably hold at selected points in the program. The proof obligations are the shown to be valid under the assumption of the auxiliary invariants.

In general terms, our strategy is to first give common semantics to the source and target languages using the formalism of *Transition Systems* (TS's). The notion of a target code $T$ being a correct implementation of a source code $S$ is that of *refinement*, stating that every computation of $T$ corresponds to some computation of $S$ with matching values of the corresponding variables. In Figure 1 we present the process of refinement as completion of a mapping diagram.

If only minor optimizations, or no optimizations at all (a debugging mode supported by most compilers), are performed, the proof that the target code refines the source program is reduced to the proof of the validity of a set of automatically generated *verification conditions* (proof obligations) which are implications in first order logic. For this simpler case all that is required is to establish the validity of the set of these verification conditions. Under the (realistic) assumption that only restricted optimization is applied to arithmetic expressions, the proof obligations are in a restricted form of first order logic, called *equational formulae*, using uninterpreted functions to represent all arithmetical operations. Recent work ([PRSS99]) has shown the feasibility of building a tool for checking the validity of such formulae. This tool is based upon finding small domain instantiations of equational formulae and then using BDD based representation to check for validity.

When the optimization switches are turned on, it is no longer sufficient to use the verification conditions which can be generated automatically. The validating tool will need additional information which specifies which optimizing transformations have been applied in the current translation. This additional information can either be provided by the compiler or inferred by a set of heuristics and analysis techniques we plan to develop. A major component of our research is the identification of a modest and self-contained instrumentation for optimizing compilers that will provide this essential information in the form of *program annotation*. This annotation will

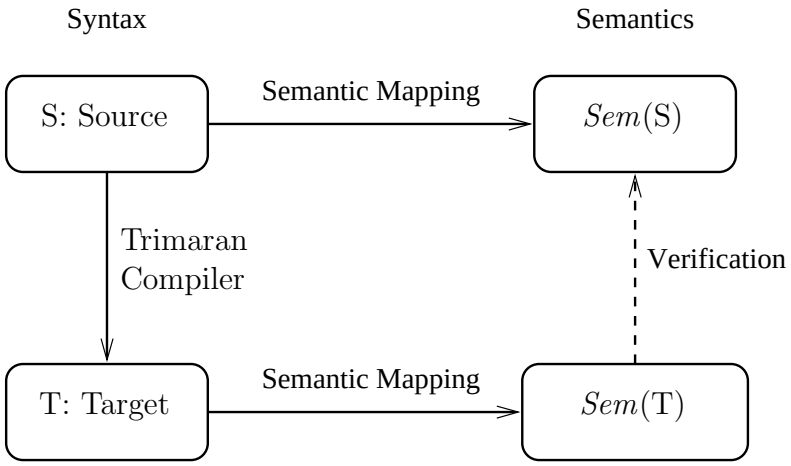


**Fig. 1.** Refinement Completes the Picture

be processed by the validation tool to form *invariant assertions* at selected control points. The verification conditions will then be augmented by being allowed to use these invariants as assumptions but also by the need to establish that these assertions are indeed invariants.

## 2 Transition Systems

In order to present the formal semantics of source and intermediate code we introduce *transition systems* TS's, a variant of the *transition systems* of [PSS98b]. A *Transition System* $S = \langle V, \mathcal{O}, \Theta, \rho \rangle$ is a state machine consisting of $V$ a set of *state variables*, $\mathcal{O} \subseteq V$ a set of *observable variables*, $\Theta$ an *initial condition* characterizing the initial states of the system, and $\rho$ a *transition relation*, relating a state to its possible successors. The variables are typed, and a *state* of a TS is a type-consistent interpretation of the variables. For a state $s$ and a variable $x \in V$, we denote by $s[x]$ the value that $s$ assigns to $x$. The transition relation refers to both unprimed and primed versions of the variables, where the primed versions refer to the values of the variables in the successor states, while unprimed versions of variables refer to their value in the pre-transition state. Thus, e.g., the transition relation may include "$y' = y + 1$" to denote that the value of the variable $y$ in the successor state is greater by one than its value in the old (pre-transition) state.

The observable variables are the variables we care about. When comparing two systems, we will require that the observable variables in the two systems match. Typically, we require that the output file of a program, i.e. the list of values printed through its execution, be identified as an observable variables. If desired, we can also include among the observables the history of external procedure calls for a selected set of procedures.

A computation of a TS is a maximal finite or infinite sequence of states, $\sigma : s_0, s_1, \ldots$ , starting with a state that satisfies the initial condition, i.e., $s_0 \models \Theta$, and every two consecutive states are related by the transitions relation, i.e. $\langle s_i, s_{i+1} \rangle \models \rho$ for every $i$, $0 \leq i + 1 < |\sigma|$[1].

**Example 1** Consider the source program FACT&DUP in the left of Figure 2 which computes in $s$ the factorial of $n = 4$, and in $j$ the product $2 \times n = 8$. On the right, we have the IR of the program, where the `while` loop is transformed into explicit tests and increments.
Here `L0` is the initial block, `L2` is the terminal block, and each label maps to a basic block.

We next translate the program in Figure 2 into a TS. The set of state variables is $V = \{\pi, i, s, j, n\}$, where $\pi$ is a control variable (program counter) which points to the next statement to be executed. The range of $\pi$ is

[1] $|\sigma|$, the *length of* $\sigma$, is the number of states in $\sigma$. When $\sigma$ is infinite, its length is $\omega$.

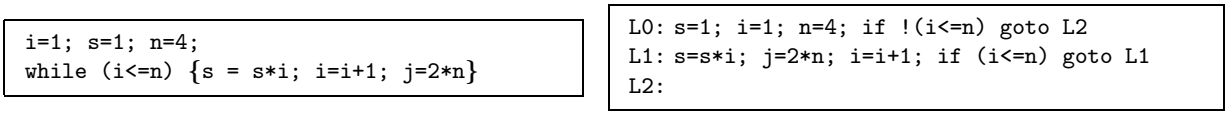


**Fig. 2.** Program FACT&DUP and its IR

$\{\texttt{L0}, \texttt{L1}, \texttt{L2}\}$. The other variables, $i, s, j$ and $n$, are all integers. The initial condition, given by $\Theta\colon \pi = \texttt{L0}$, states that the program starts at location L0. As observables, we take $\mathcal{O} = \{s, j\}$.

The transition relation $\rho$ can be presented as the disjunction of four disjuncts

$$\rho \quad = \quad \rho_{01} \ \vee \ \rho_{02} \ \vee \ \rho_{11} \ \vee \ \rho_{12},$$

where $\rho_{ij}$ describes all possible moves from Li to Lj.

In the following table we present the four disjuncts of the transition relations of program FACT&DUP:

| | |
|---|---|
| $\rho_{01}$ | $(\pi = \texttt{L0}) \ \wedge \ (s' = 1) \ \wedge \ (i' = 1) \ \wedge \ (n' = 4) \ \wedge \ (j' = j) \ \wedge \ (i' \leq n') \ \wedge \ (\pi' = \texttt{L1})$ |
| $\rho_{02}$ | $(\pi = \texttt{L0}) \ \wedge \ (s' = 1) \ \wedge \ (i' = 1) \ \wedge \ (n' = 4) \ \wedge \ (j' = j) \ \wedge \ (i' > n') \ \wedge \ (\pi' = \texttt{L2})$ |
| $\rho_{11}$ | $(\pi = \texttt{L1}) \ \wedge \ (s' = s \times i) \ \wedge \ (n' = n) \ \wedge \ (i' \leq n') \ \wedge \ (i' = i + 1) \ \wedge \ (j' = 2 \times n) \ \wedge \ (\pi' = \texttt{L1})$ |
| $\rho_{12}$ | $(\pi = \texttt{L1}) \ \wedge \ (s' = s \times i) \ \wedge \ (n' = n) \ \wedge \ (i' > n') \ \wedge \ (i' = i + 1) \ \wedge \ (j' = 2 \times n) \ \wedge \ (\pi' = \texttt{L2})$ |

The only computation of the program is:

$$\langle \texttt{L0}, -, -, -, - \rangle \xrightarrow{\rho_{01}} \langle \texttt{L1}, 1, 1, -, 4 \rangle \xrightarrow{\rho_{11}} \langle \texttt{L1}, 2, 1, 8, 4 \rangle \xrightarrow{\rho_{11}} \langle \texttt{L1}, 3, 2, 8, 4 \rangle \xrightarrow{\rho_{11}} \langle \texttt{L1}, 4, 6, 8, 4 \rangle$$
$$\xrightarrow{\rho_{11}} \langle \texttt{L1}, 5, 24, 8, 4 \rangle \xrightarrow{\rho_{12}} \langle \texttt{L2}, 5, 24, 8, 4 \rangle$$

where each state is described by the values it assigns to $\langle \pi, i, s, j, n \rangle$.

Let $P_{\overset{S}{}} = \langle V_{\overset{S}{}}, \mathcal{O}_{\overset{S}{}}, \Theta_{\overset{S}{}}, \rho_{\overset{S}{}} \rangle$ and $P_{\overset{T}{}} = \langle V_{\overset{T}{}}, \mathcal{O}_{\overset{T}{}}, \Theta_{\overset{T}{}}, \rho_{\overset{T}{}} \rangle$ be two TS's, to which we refer as the *source* and *target* TS's, respectively. Such two systems are called *comparable* if there exists a 1-1 correspondence between the observables of $P_{\overset{S}{}}$ and those of $P_{\overset{T}{}}$. To simplify the notation, we denote by $X \in \mathcal{O}_{\overset{S}{}}$ and $x \in \mathcal{O}_{\overset{T}{}}$ the corresponding observables in the two systems. We say that $P_{\overset{T}{}}$ is a *correct translation* (*refinement*) of $P_{\overset{S}{}}$ if for every finite (i.e., terminating) $P_{\overset{T}{}}$-computation $\sigma^{\overset{T}{}}\colon \sigma^{\overset{T}{}}_0, \ldots, \sigma^{\overset{T}{}}_m$, there exists a finite $P_{\overset{S}{}}$-computation $\sigma^{\overset{S}{}}\colon \sigma^{\overset{S}{}}_0, \ldots, \sigma^{\overset{S}{}}_k$, such that $s^{\overset{T}{}}_m[x] = s^{\overset{S}{}}_k[X]$ for every $x \in V_{\overset{T}{}}$.

## 3 Translation Validation for Structure-Preserving Transformations

In Figure 3 we introduce VALIDATE —the main proof rule used by the validator to establish that target codes are translations of their source codes for the case of transformations that preserve the structure of the program. VALIDATE is an elaboration of the computational induction approach ([Flo67]) that offers a proof methodology to validate that one program *refines* another by establishing control mapping from target to source locations, data abstraction mapping from target to source variables, and proving that they are preserved with each step of the target program.

The assertions $\varphi_i$ in step (2) are program annotations that are expected to be provided by the compiler. Thus, $\varphi_i$ is used as a hypothesis at the antecedent of the implication $C_{ij}$. In return, the validator also has to establish that $\varphi_j$ holds after the transition. Thus, we do not trust the annotation provided by the instrumented compiler but, as part of the verification effort, we confirm that the proposed assertions are indeed inductive and hold whenever visited. Since the assertion mention only target variables, their validity should depend solely on the target code.

In most cases, the existential quantification in verification condition (4) can be eliminated. This is based on the observations that the implication $p \rightarrow \exists x' : (x' = E) \wedge q$ is validity-equivalent to the implication $p \wedge (x' = E) \rightarrow q$.

In [PZP00] we proved a theorem stating that the proof rule VALIDATE is sound.

1. Establish a *control abstraction* $\kappa$: $Dom(\texttt{pc}) \rightarrow Dom(\pi)$ mapping basic blocks into a corresponding value of the source control variable $\pi$. The control abstraction should map the initial and terminal blocks into the initial and terminal locations of the source programs.
2. For each basic block `B(i)`, form an *invariant* $\varphi_i$ that may refer only to concrete (target) variables.
3. Establish a *data abstraction*

   $$\alpha : (p_1 \rightarrow v_1 = E_1) \ \wedge \ \cdots \ \wedge \ (p_n \rightarrow v_n = E_n)$$

   assigning to some source state variables $v_i \in V_S - \{\pi\}$ an expression $E_i$ over the target state variables, conditional on the (target) boolean expression $p_i$. Note, that $\alpha$ may contain more than one clause for the same variable.
4. For each basic block `B(i)` and `B(j)` such that `B(j)` is a successor of `B(i)` in the IR-graph, form the verification condition

   $$C_{ij}: \qquad \varphi_i \ \wedge \ \alpha \ \wedge \ \overset{T}{\rho_{ij}} \quad \rightarrow \quad \exists {V_S}' : \overset{S}{\rho_{\kappa(i),\kappa(j)}} \ \wedge \ \alpha' \ \wedge \ \varphi'_j.$$

5. Establish the validity of all the generated verification conditions.

**Fig. 3.** VALIDATE –A procedure to validate translations

### 3.1 Validating the Verification Conditions

Following the generation of the verification conditions whose validity implies that the target $T$ is a correct translation of the source program $S$, it only remains to check that these implications are indeed valid. The approach promoted here will make sense only if this validation (as well as the preceding steps of the conditions' generation) can be done in a fully automatic manner with no user intervention.

Pars of the validation task can be performed using CVT tool developed for the *Sacres* project [PRSS99] (see [PZP00] for an overview.) For other parts, we need some arithmetical capabilities for which we used the STeP system ([MAB+94].) We are currently exploring other packages that can provide similar capabilities.

### 3.2 Example of Application of VALIDATE

Consider the program FACT&DUP of Figure 2 after a series of optimizations: constant propagation and folding within a basic block, constant propagation between basic blocks, dead code elimination, and loop invariant code motion. The resulting code is in Figure 4; the underlined instructions there are assignments to *auxiliary* (boolean) variables that are added by the validator.

```
L0: j_rel = 0; i = 1; s = 1; j = 8;
L1: j_rel = 1; s = s*i; i = i+1; if (i<=4) goto L1
L2:
```

**Fig. 4.** Annotated Program After Loop Invariant Code Motion

To validate the program, we use the trivial control mapping, an invariant assertion $\varphi_1 = \{j = 8\}$ at `L1`, and the following data abstraction:

$$\alpha: \ \ (N = 4) \ \wedge \ (I = i) \ \wedge \ (S = s) \ \wedge \ (j_{rel} \wedge \texttt{pc} = \texttt{L1} \rightarrow J = j)$$

where in both $\varphi_1$ and $\alpha$, we use capital letters to denote the source versions of the variables $n$, $i$, $s$ and $j$.

Armed with $\varphi_1$ and $\alpha$, the verification conditions are straightforward to construct and prove.

## 4 Validating Structure-Modifying (Loop) Optimizations

Since the simulation proof method assumes that the source and target have similar structures, rule VALIDATE cannot be used to validate many of the loop optimizations. Therefore, we propose an alternate methodology for validating loop optimizations, that consists of a set of "meta-rules", each dealing with a *set* of loop optimizations. The soundness of the meta-rules is established separately. Usually, structure-modifying optimizations are applied to small localized sections of the source program, while the rest of the program is only optimized by structure-preserving transformations. Therefore, the general validation of a translation will combine these two techniques.

The first meta-rule covers a wide range of optimizations. In fact, it covers all optimizations in which the loop body itself is not altered (except for substitution of indices.) The obvious cases that are not covered by this meta-rule are loop distribution and fusion, that are covered by a different meta-rule.

### 4.1 Loop Body Preserving Optimizations

This meta rule covers all cases where the only changes the optimizations imposes on the loop body are those caused by substitution of control variables. Thus, it covers all unimodular loop transformations, as well as tiling and combinations of the above.

A loop transformation has the general form of Figure 5.

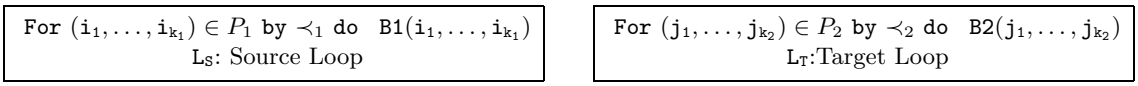


**Fig. 5.** A General Loop Transformation

In this representation, we assume that each of the loop bodies, `B1` and `B2`, have some occurrences of the variable $\vec{i} = (i_1, \ldots, i_{k_1})$ and $\vec{j} = (j_1, \ldots, j_{k_2})$ respectively; these variables are not modified in either `B1` or `B2`. We use the notation $\mathtt{B}(\vec{k})$ to indicate an instance of block `B` where the loop control variables $\vec{i}$ have the value $\vec{k}$ upon entrance to `B`.

Our first meta rule, that applies to transformation as in Figure 5, is in Figure 6.

For some $k_1 \times k_2$ *transformation matrix* $\mathbf{T}$ and vector $\vec{b} = (b_1, \ldots, b_{k_1})$,

1. $P_1 = \mathbf{T} \cdot P_2 + \vec{b}$;
2. For every $\vec{j} \in P_2$, $\mathtt{B2}(\vec{j}) \approx \mathtt{B1}(\mathbf{T} \cdot \vec{j} + \vec{b})$;
3. For every $\vec{j_1}, \vec{j_2} \in P_2$ such that $\vec{j_1} \prec_2 \vec{j_2} \in P_2$ and $\mathbf{T} \cdot \vec{j_2} \prec_1 \mathbf{T} \cdot \vec{j_1}$, $\mathtt{B1}(\mathbf{T} \cdot \vec{j_1} + \vec{b}); \mathtt{B1}(\mathbf{T} \cdot \vec{j_2} + \vec{b}) \approx \mathtt{B1}(\mathbf{T} \cdot \vec{j_2} + \vec{b}); \mathtt{B1}(\mathbf{T} \cdot \vec{j_1} + \vec{b})$

---

$\mathtt{For}\ \vec{i} \in P_1\ \mathtt{by}\ \prec_1\ \mathtt{do}\ \ \mathtt{B1}(\vec{i}) \ \approx\ \mathtt{For}\ \vec{j} \in P_2\ \mathtt{by}\ \prec_2\ \mathtt{do}\ \ \mathtt{B2}(\vec{j})$

**Fig. 6.** Meta Proof Rule for Transformation of Figure 5

Condition (1) of the rule states that there is a transformation such that (the polyhedron) $P_1$ is the result of applying the transformation to $P_2$, thus, the two loops are defined over the same vectors. Condition (2) of the rule states the loop body of each iteration of the source is obtained by the loop body of the transformed iteration of the target, thus, the loop body of two corresponding iterations is the same. Condition (3) guarantees that any two iterations that are executed in different order in the source and target loops do not depend on one another.

Below are some examples of application of the rule. Unless otherwise stated, we assume that in each of the transformations $\vec{b} = \mathbf{0}$.

*A Simple Loop Interchange.* The source and target loops are described in figure 7.

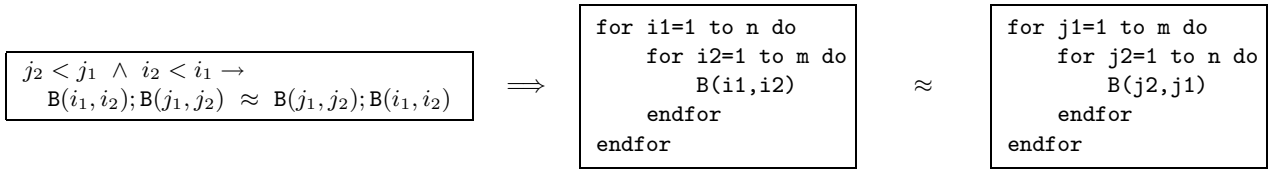


**Fig. 7.** A Loop Interchange Example

Here, $P_1 = \{(i_1, i_2) : 1 \leq i_1 \leq n, 1 \leq i_2 \leq m\}$, $P_2 = \{(j_1, j_2) : 1 \leq j_1 \leq m, 1 \leq j_2 \leq n\}$, $\prec_1$ is lexicographic ordering over $(i_1, i_2)$ and $\prec_2$ is lexicographic ordering over $(j_1, j_2)$.The transformation matrix is

$$\mathbf{T} \ = \ \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

To verify the correctness of the transformation, note that for every $(j, i) \in P_2$, $\mathbf{T} \cdot (j, i) = (i, j)$. Conditions (1) and (2) of the rule are obvious. For condition (3), we have to verify that $\mathtt{B}(i_1, i_2); \mathtt{B}(j_1, j_2) \approx \mathtt{B}(j_1, j_2); \mathtt{B}(i_1, i_2)$ for every for every $(i_1, i_2)$ $(j_1, j_2) \in P_2$ such that $j_2 < j_1$ and $i_2 > i_1$.

For example (taken from [Muc97]), let the loop body be "$\mathtt{a[i, j]} := \mathtt{b[i]} + 0.5; \mathtt{a[i + 1, j]} := \mathtt{b[i]} - 0.5$". The validity of the transformation then follows since the transition relation corresponding to $\mathtt{B}(i_1, i_2); \mathtt{B}(j_1, j_2)$ is

$$a'[i_1, i_2] = b[i_1] + 0.5 \ \wedge \ a'[i_1 + 1, i_2] = b[i_1] - 0.5 \ \wedge \ a'[j_1, j_2] = b[j_1] + 0.5 \ \wedge \ a'[j_1 + 1, j_2] = b[j_1] - 0.5$$
$$\wedge \ \forall (i, j) \notin \{(i_1, i_2), (j_1, j_2), (i_1 + 1, i_2), (j_1 + 1, j_2)\}. a'[i, j] = a[i, j]$$

which is equivalent to the transition relation corresponding to $\mathtt{B}(j_1, j_2); \mathtt{B}(i_1, i_2)$ when $j_2 < j_1$ and $i_2 > i_1$ (since then $\{(i_1, j_1), (i_1 + 1, j_1)\}$ are pairwise disjoint from $\{(i_2, j_2), (i_1 + 1, j_1)\}$.)

The equivalence between the two code fragments computed at the last step can be established by computing their respective transition relations and submitting then to the decision procedure underlying CVT.

*A Simple Loop Skewing.* The source and target loops are described in figure 8.

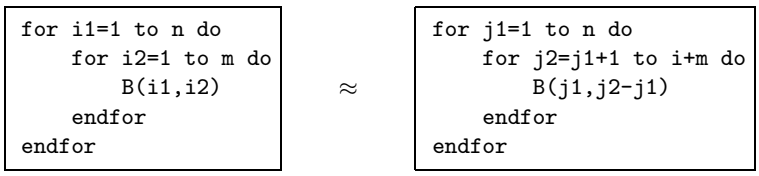


**Fig. 8.** A Loop Skewing Example

Here, $P_1 = \{(i_1, i_1) : 1 \leq i_1 \leq n, 1 \leq i_2 \leq m\}$,$P_2 = \{(j_1, j_2) : 1 \leq j_1 \leq n, j_1 < j_2 \leq i + m\}$, and both $\prec_1$ and $\prec_2$ are lexicographical. The transformation matrix is

$$\mathbf{T} \ = \ \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$$

To verify the correctness of the transformation, note that for every $(j_1, j_2) \in P_2$, $\mathbf{T} \cdot (j_1, j_2) = (j_1, j_2 - j_1)$. For condition (1) of the rule, we have that $\mathbf{T} \cdot P_2 = \{(j_1, j_2 - j_1) : 1 \leq j_1 \leq n, j_1 < j_2 \leq j_1 + m\} = \{(i_1, i_2) : 1 \leq i_1, i_2 \leq n\} = P_1$. As for condition (3), note that it is trivially true since $(j_1^1, j_2^1) \prec (j_1^2, j_2^2) \Longleftrightarrow (j_1^1, j_2^1 - j_1^1) \prec (j_1^2, j_2^2 - j_1^2)$.

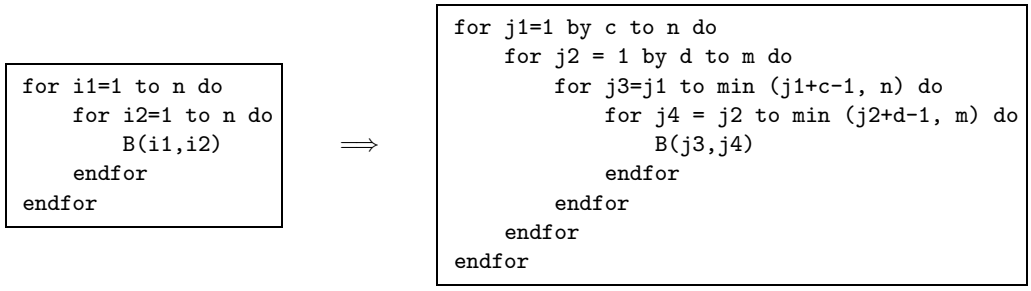


**Fig. 9.** A Loop Tiling Example

*A depth-2 Loop Tiling.* Consider the source and target loop in Figure 9.

Here, $P_1 = \{(i_1, i_2) : 1 \leq i_1 \leq n, \leq i_2 \leq m\}$, $P_2 = \{(j_1, j_2, j_3, j_4) : j_1 = 1 + c\alpha$ for $0 \leq \alpha \leq \lfloor \frac{n-1}{c} \rfloor, j_2 = 1 + d\beta$ for $0 \leq \beta \leq \lfloor \frac{m-1}{d} \rfloor, j_1 \leq j_3 \leq \min(j_1 + c - 1, n), j_2 \leq j_4 \leq \min(j_2 + d - 1, m)\}$, and both $\prec_1$ and $\prec_2$ are lexicographical. The transformation matrix is

$$\mathbf{T} = \begin{pmatrix} 0\,0\,1\,0 \\ 0\,0\,0\,1 \end{pmatrix}$$

To verify the correctness of the transformation, note that for every $(j_1, j_2, j_3, j_4) \in P_2$, $\mathbf{T} \cdot (j_1, j_2, j_3, j_4) = (j_3, j_4)$. Condition (1) of the rule is trivially met. Similarly for condition (2). As for condition (3), note that the set of pairs of $P_2$-vectors that are ordered by $\prec_2$, and after applying $\mathbf{T}$ to them are ordered by $\succ_1$, are the pairs in the set

$$\{((j_1, j_2, j_3, j_4), (\hat{j_1}, \hat{j_2}, \hat{j_3}, \hat{j_4})) : (j_1, j_2, j_3, j_4), (\hat{j_1}, \hat{j_2}, \hat{j_3}, \hat{j_4}) \in P_2 \ \wedge \ \hat{j_1} = j_1 \ \wedge \ \hat{j_2} > j_2 \ \wedge \ \hat{j_3} < j_3\} \qquad (1)$$

Taking a concrete example (from [Muc97]), assume that $m = n$, $c = d = 2$, and the loop body is "`a[j3] := a[j3+j4]+1`". To verify condition (3), we need to validate $\mathtt{B}(j_3, j_4); \mathtt{B}(\hat{j_3}, \hat{j_4}) \approx \mathtt{B}(\hat{j_3}, \hat{j_4}); \mathtt{B}(j_3, j_4)$ for every pair $(j_3, j_4), (\hat{j_3}, \hat{j_4})$ such that $\{((j_1, j_2, j_3, j_4), (\hat{j_1}, \hat{j_2}, \hat{j_3}, \hat{j_4})\}$ is in the set of Equation 1. That is, the equivalence of the transition relation has to be verified for every

$$\{((j_3, j_4), (\hat{i_3}, \hat{j_4})) : 1 \leq j_3 < n \ \wedge \ i_3 = 0 \bmod 2 \ \wedge \ \hat{j_3} = j_3 + 1 \ \wedge \ 1 \leq j_4 \leq n \ \wedge \ j_4 > \hat{j_4} + (\hat{j_4} \bmod 2)\}$$

The transition relation for $\mathtt{B}(j_3, j_4); \mathtt{B}(\hat{j_3}, \hat{j_4})$ when $\hat{j_3} > j_3$ (and, therefore, $j_3 \neq \hat{j_3} + \hat{j_4}$) is

$$a'[j_3] = a[j_3 + j_4] + 1 \ \wedge \ a'[\hat{j_3}] = a[\hat{j_3} + \hat{j_4}] \ \wedge \ \forall k \neq j_3, \hat{j_3}. a'[k] = a[k]$$

When $j_4 > \hat{j_4}$ (and $\hat{j_3} \neq j_3 + j_4$) the transition relation for $\mathtt{B}(\hat{j_3}, \hat{j_4}); \mathtt{B}(j_3, j_4)$ is exactly the same. Hence, condition (3) of the rule is met for this example.

### 4.2 Automatic Validation

Validating the proof rule in Figure 5 involves three steps:

1. Generating $P_1$, $P_2$, $\mathbf{T}$, and $\vec{b}$ and proving that $P_1 = \mathbf{T} \cdot P_2 + \vec{b}$: The polyhedra $P_1$ and $P_2$ can be automatically obtained from the syntactic form of the iterations at hand; the transformation matrix $\mathbf{T}$ and the vector $\vec{b}$ can be obtained by proper instrumentation from the compiler; it is also conceivable that they can be obtained by simple heuristic method, especially when only a single transformation is applied at a time. The equivalence of $P_1$ and $\mathbf{T} \cdot P_2 + \vec{b}$ can be checked by any suitable mathematic package, e.g., Mathematica [Wol99].
2. Checking that $B_1(\mathbf{T} \cdot \vec{j} + \vec{b}) \approx B_2(\vec{j})$ for every $\vec{j} \in P_2$: Often, once $\mathbf{T}$ and $\vec{b}$ are known, this amounts to merely checking that $B_1(\mathbf{T} \cdot \vec{j} + \vec{b}) = B_2(\vec{j})$. When the loops bodies are significantly altered, the equivalence may be established by CVT-like tools.

3. Validating that any two iterations whose relative order in the source and the destination are different do not depend on one another: The first step is to find the "potentially offensive" pairs of iterations. This can be done by using the same mathematical package used in step (1). For each of these pairs (sets of them may be represented symbolically), the equivalence of the two bodies can be established using the decision procedure underlying CVT.

### 4.3 Loop Distribution and Fusion

Another set of loop transformations alters the loop body itself and is therefore not covered by the rule above. For an example of such a transformation, consider loop distribution and fusion consider the programs $P_{fus}$ and $P_{dis}$ in Figure 10.

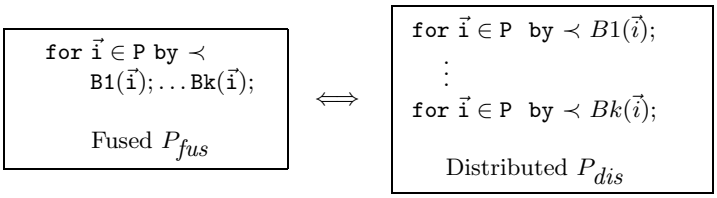


**Fig. 10.** Distribution/Fusion

When considering loop distribution, the source is $P_{fus}$ and the target is $P_{dis}$. Loop distribution is considered legal if it does not result in breaking any cycles in the dependence graph of the original loops [Muc97]. Alternatively, loop distribution is legal if executing all the B1's before all the B2's is equivalent to executing the B1's and B2's in an interleaved fashion. When considering loop fusion the source is $P_{dis}$ and the target is $P_{fus}$. Obviously, fusing $P_{dis}$ into $P_{fus}$ is legal only when distributing $P_{fus}$ into $P_{dis}$ is legal.

Let $P_{fus}$ and $P_{dis}$ be as in Figure 10. Figure 11 presents a rule for validating loop distribution and fusion for the case that $k = 2$. Proving that the equivalence appearing in the premise may require invocation of some

$$\frac{\text{For every } \vec{i}, \vec{j} \in P,\ \vec{i} \prec \vec{j} \quad \longrightarrow \quad \texttt{B2}(\vec{i}); \texttt{B1}(\vec{j}) \ \approx \ \texttt{B1}(\vec{j}); \texttt{B2}(\vec{i}).}{P_{fus} \ \approx \ P_{dis}}$$

**Fig. 11.** Proof Rule for Loop Distribution and Fusion

other proof rules (VALIDATE, e.g.).

Consider the following example.

B1($i$): `X[i]=X[i-1]+X[i]` B2($i$): `Y[i]=X[i]+1`

To prove that we can apply distribution/fusion for these loop bodies, according to our proof rule we have to establish the validity of:

$$\begin{aligned} i_0 < j_0 \ \rightarrow \Big( & (X'[j_0] = X[j_0 - 1] + X[j_0]) \ \wedge \ (\forall k \neq j . X'[k] = X[k]) \ \wedge \\ & (Y'[i_0] = X'[i_0] + 1) \ \wedge \ (\forall k \neq i_0 . Y'[k] = Y[k]) \quad \approx \\ & (Y'[i_0] = X[i_0] + 1) \ \wedge \ (\forall k \neq i . Y'[k] = Y[k]) \ \wedge \\ & (X'[j_0] = X[j_0 - 1] + X[j_0]) \ \wedge \ (\forall k \neq j_0 . X'[k] = X[k]) \Big) \end{aligned}$$

which is true since $X'[j_0] = X[j_0]$.

Assume, however, that B1($i$) is replaced with B3($i$): `X[i-1]=X[i-1]+X[i]`. We then no longer have the equivalence between the programs, since B3($j_0$);B2($i_0$) implies that $X'[j_0] = X[j_0]$ only when $j_0 > i_0 + 1$, and

thus the value of $Y[j_0]$ resulting from the execution of $\texttt{B3}(j_0)$ before $\texttt{B2}(i_0)$ may differ from the resulting from the execution of $\texttt{B3}(j_0)$ after $\texttt{B2}(i_0)$.

The soundness of the meta proof rule of Figure 11 was established using PVS ([SOR93].

## 5 A Note on Loop Unrolling

The IR of Loop unrolling is represented in figure 12 (where we assume $n \geq c > 1$.) There are several strategies

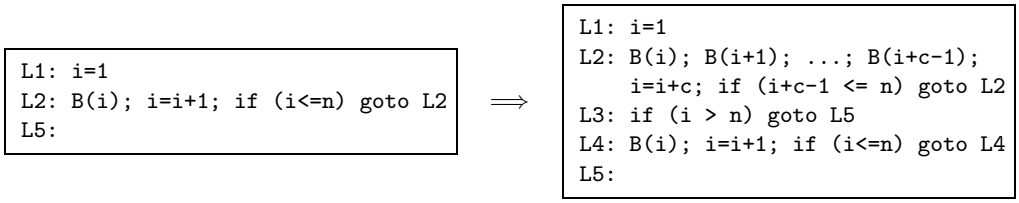


**Fig. 12.** Loop Unrolling

for dealing with loop unrolling. One is to design a meta-rule that deals with it directly. Another is to consider loop unrolling as a special case of tiling an $n \times 1$ array with tiles of size $c$, and then unrolling the innermost loop. A third approach, which we sketch here, it to consider loop unrolling as a structure-preserving transformation and apply VALIDATE to it.

We first "split" the target block L2 into $c$ sub-blocks as follows:

```
L2.1:   B(i);
L2.2:   B(i+1);
⋮       ⋮
L2.c-1: B(i+c-2);
L2.c:   B(i+c-1); i=i+c; if (i+c-1 <= n) goto L2
```

We then define auxiliary assertions:

$$\varphi_2 = i = 1 \bmod c \ \wedge \ 1 \leq i \leq n$$
$$\varphi_4 = n - c + 1 \leq i \leq n$$

For the control mapping we take $1 \rightarrow 1,\ 2.j \rightarrow 2$ for every $j = 1, \ldots, c$, $4 \rightarrow 2$, and $5 \rightarrow 5$. For the data abstraction we take:

$$(N = n) \ \wedge \ ((\texttt{pc} = 1 \vee \texttt{pc} = 2.1 \vee \texttt{pc} = 4 \vee \texttt{pc} = 5) \rightarrow I = i) \ \wedge \\ (\texttt{pc} = 2.2 \rightarrow I = i + 1) \ \wedge \ \ldots \ \wedge \ (\texttt{pc} = 2.c \rightarrow I = i + c - 1)$$

where, as before, we use capital letters to denote the source versions of the variables.
The verification conditions are now straightforward to construct and verify.

In the Appendix we give an example of a compilation of the Trimaran compiler that involves loop unrolling, and verify it using VALIDATE.

**Acknowledgment** We gratefully acknowledge the help of Ben Goldberg who provided us with examples and explanations of modern optimizations techniques for architecture targeted compilation. Paritosh Pandya helped formulate perliminary versions of rule VALIDATE. Jessie Xu verified some of the meta rules, using PVS, and Henny Sipma added automatic generation of verification conditions to STeP.

## References

[Flo67] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

[Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[Nec97] G.C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. Princ. of Prog. Lang.*, pages 106–119, 1997.

[Nec00] G. Necula. Translation validation of an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 2000*, pages 83–95, 2000.

[NL98] G.C. Necula and P. Lee. The design and implementation of a certifying compilers. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 1998*, pages 333–344, 1998.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. In *In N. Halbwachs and D. Peled, editors, Proc.* $11^{st}$ *Intl. Conference on Computer Aided Verification (CAV'99), volume 1633 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 455–469, 1999.

[PSS98a] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[PSS98b] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *In B. Steffen, editor, Proc.* $4^{th}$ *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), volume 1384 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 151–166, 1998.

[PZP00] A. Pnueli, L. Zuck, and P. Pandya. Translation validation of optimizing compilers by computational induction. Technical report, Courant Institute of Mathematical Sciences, New York University, 2000.

[RM00] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Run-Time Result Verification Workshop*, Trento, July 2000.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (draft). Technical report, Comp. Sci.,Laboratory, SRI International, Menlo Park, CA, 1993.

[Wol99] S. Wolfram. *The Mathematica Book*. Cambridge University Press, 1999.

## A Translation Validation of Machine-Specific Optimizations

In this section we present an example of translation validation of low-level code produced by back end of the Trimaran compiler. The verification conditions were obtained manually and verified using STeP ([MAB+94].) The original C program and its Trimaran IR translation (in a more readable form) are given in Figure 13. The program gets as an input an array `a[0..99]` of (4-byte) integers and a memory address `M` whose value is to be added to every array element.

In this example, Trimaran translates the intermediate code to machine code, assigns the $r$ registers to store intermediate values of the program, the $tr$ branch registers to store addresses of branch locations, and the $p$ predicate registers to store results of conditionals. It also performs loop inversion to change the "while" to a "repeat", loop unrolling, strength reduction, and dead code elimination (of the loop variable.) It also schedules the instructions. We present the optimized code, with the scheduling within each block, in Figure 14. Labels BB3.1 and BB3.2 were added manually for clarity.

To prove that the target code of Figure 14 is a translation of the source code—the IR code of Figure 13—we use VALIDATE with the following mappings: the control mapping maps target `BB3, BB3.1` and `BB3.2` to `L2`, while `BB6` is mapped to `L1` and `BB4` is mapped to `L3`. The data abstraction $\alpha$ is described by:

$$(0 \le j \le 99 \rightarrow a[j] = \mathsf{mem}(adrs\_a + 4 \cdot j)) \ \wedge \ (M = \mathsf{mem}(adrs\_M)) \ \wedge \ (\mathtt{pc} = 3 \vee (\mathtt{pc} = 4 \wedge p_4)) \rightarrow i = r_3 - 2)$$
$$\wedge \ (\mathtt{pc} = 3.1 \vee (\mathtt{pc} = 4 \wedge p_2)) \rightarrow i = r_3 - 4) \ \wedge \ (\mathtt{pc} = 3.2 \vee (\mathtt{pc} = 4 \wedge p_3)) \rightarrow i = r_3 - 3)$$

```
int M;
int a[100];
main () {
  int i;
  for (i=0;i < 100;i++){
    a[i] += M;}
}
```

```
extern int IMPACT_EDG_GENERATED;
extern int main();
{
int M;
int a[100];
int i;
LL_1 :
   i = 0;
   if (i < 100) goto LL_2;
   else goto LL_3;
```

```
LL_2:
   (a[i]) += M;
   i++ ;
   if (i < 100) goto LL_2;
LL_3 :
   return(0);
}
```

**Fig. 13.** Original C-Code and its Trimaran IR

| | | | |
|---|---|---|---|
| BB6: | r5 = _M + 0 | r2 = _a + 8 | r3 = 2 |
| | r4 = ld r5 | | |
| BB3: | r5 = r2 + -8 | r6 = r2 + -4 | r11 = r2 + -8 tr2 = pbrr BB4 r7 = ld r2 |
| | r8 = ld r5 | r9 = ld r6 | r13 = r2 + -4 tr3 = pbrr BB4 |
| | | p2 u = cmpp.>=.UN.UN r3 101 | p3 u = cmpp.>=.UN.UN r3 100 |
| | r3 = r3 + 3 | tr4 = pbrr BB3 r14 = r7 + r4 | |
| | r10 = r8 + r4 | r12 = r9 + r4 | p4 u = cmpp.<.UN.UN r3 102 |
| | st r11 r10 | brct tr2 p2 | |
| (BB3.1:) | st r13 r12 | brct tr3 p3 | |
| (BB3.2:) | st r2 r14 | | |
| | r2 = r2 + 12 | brct tr4 p4 | |
| BB4: | Exit | | |

**Fig. 14.** An Optimized Trimaran Code (with scheduling)

When dealing with machine-level code, it is necessary to make some assumption about the memory allocation. Here, the memory mem is assumed to be an array $[0 \dots N]$ of bytes; the array $a$ resides in some 400 consecutive bytes of the memory, thus $0 \leq adrs\_a \leq N - 400$. Similarly, the 4-byte input $M$ resides in the memory, and does not intersect with the array. We ignore here the issue of how values are computed from memory addresses. We do note, however, that it is the validator's task to guarantee that the code produced by the compiler does not access out-of-bounds memory cells. In particular, every load or store instruction has to reference memory cells in the range $[0 \dots N]$.

To validate the translation, we annotate the program with the following invariants:

$\varphi_3$: $\{r_3 \equiv 2 \bmod 3 \ \wedge \ 2 \leq r_3 \leq 101 \ \wedge \ r_2 = adrs\_a + 4(r_3 - 2) \ \wedge \ r_5 = adrs\_M\}$

$\varphi_{3.1}$: $\{r_3 \equiv 2 \bmod 3 \ \wedge \ 5 \leq r_3 \leq 103 \ \wedge \ r_2 = adrs\_a + 4(r_3 - 5) \ \wedge \ r_5 = adrs\_M \ \wedge$
$p_3 = (r_3 \geq 103) \ \wedge \ p_4 = (r_2 < 102) \ \wedge \ tr_3 = \mathtt{BB4} \ \wedge \ tr_4 = \mathtt{BB3} \ \wedge$
$r_{12} = \mathsf{mem}(adrs\_a + 4(r_3 - 4)) + \mathsf{mem}(adrs\_M) \ \wedge$
$r_{14} = \mathsf{mem}(adrs\_a + 4(r_3 - 3)) + \mathsf{mem}(adrs\_M)\}$

$\varphi_{3.2}$: $\{r_3 \equiv 2 \bmod 3 \ \wedge \ 5 \leq r_3 \leq 102 \ \wedge \ r_2 = adrs\_a + 4(r_3 - 5) \ \wedge \ r_5 = adrs\_M \ \wedge$
$p_4 = (r_2 < 102) \ \wedge \ tr_4 = \mathtt{BB3} \ \wedge \ r_{14} = \mathsf{mem}(adrs\_a + 4(r_3 - 3)) + \mathsf{mem}(adrs\_M)\}$

The transition relation, while tedious, is rather straightforward to derive. Note that store instructions update the relevant memory cells, while no other instruction alters the memory content. Using the mappings and invariants above, we managed to prove the derived verification conditions using STeP. However, we obtained a counter-example for the claim that the memory cells accessed are in the correct range. In fact, in the last execution of BB3, the value of $r_6$ exceeds $N$, thus the instruction `r9 = ld r6` could cause a segmentation fault in this case.