Electronic Notes in Theoretical Computer Science 70 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 22 pages

Translation and Run-Time Validation of Optimized Code [1,2]

Lenore Zuck [3] Amir Pnueli [4] Yi Fang [3] Benjamin Goldberg [3] Ying Hu [3]

Abstract

The paper presents approaches to the validation of optimizing compilers. The emphasis is on aggressive and architecture-targeted optimizations which try to obtain the highest performance from modern architectures, in particular EPIC-like microprocessors. Rather than verify the compiler, the approach of *translation validation* performs a validation check after every run of the compiler, producing a formal proof that the produced target code is a correct implementation of the source code.

First we survey the standard approach to validation of optimizations which preserve the loop structure of the code (though they may move code in and out of loops and radically modify individual statements), present a simulation-based general technique for validating such optimizations, and describe a tool, VOC-64, which implements these technique. For more aggressive optimizations which, typically, alter the loop structure of the code, such as loop distribution and fusion, loop tiling, and loop interchanges, we present a set of *permutation rules* which establish that the transformed code satisfies all the implied data dependencies necessary for the validity of the considered transformation. We describe the necessary extensions to the VOC-64 in order to validate these structure-modifying optimizations.

Finally, the paper discusses preliminary work on *run-time validation of speculative loop optimizations*, that involves using run-time tests to ensure the correctness of loop optimizations which neither the compiler nor compiler-validation techniques can guarantee the correctness of. Unlike compiler validation, run-time validation has not only the task of determining when an optimization has generated incorrect code, but also has the task of recovering from the optimization without aborting the program or producing an incorrect result. This technique has been applied to several loop optimizations, including loop interchange, loop tiling, and software pipelining and appears to be quite promising.

[1] This research was supported in part by NSF grant CCR-0098299, ONR grant N00014-99-1-0131, and the John von Neumann Minerva Center for Verification of Reactive Systems.

[2] Email: {zuck,amir,yifang,goldberg}@cs.nyu.edu

[3] Department of Computer Science, New York University

[4] Department of Computer Science, Weizmann Institute of Science

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 focus on verification of specification with respect to requirements, and high-level code with respect to specification. However, if one is to prove that the high-level specification is correctly implemented in low-level code, one needs to verify the compiler which performs the translations. Verifying the correctness of modern optimizing compilers is challenging because of the complexity and reconfigurability of the target architectures, as well as 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. Using the translation validation approach, rather than verify 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 families of microprocessor architectures, such as the EPIC family exemplified by the Intel IA-64 architecture, places an even heavier responsibility on optimizing compilers. Static compile-time dependence analysis and instruction scheduling is required to exploit instruction-level parallelism in order to compete with other architectures, such as the super-scalar class of machines where the hardware determines dependences and reorders instructions at run-time. As a result, a new family of sophisticated optimizations have been developed and incorporated into compilers targeted at EPIC architectures.

Prior work ([PSS98a]) developed a tool for translation validation, CVT, that succeeded in automatically verifying translations involving approximately 10,000 lines of source code in about 10 minutes. The success of CVT critically depends on some simplifying assumptions that restricts the source and target to programs with a single external loop, and assume 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* that are often 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.

Initial steps towards this goal are described in [ZPFG02]. There, we develop the theory of correct translation. We distinguish between *structure preserving* optimizations, that admit a clear mapping of control points in the target program to corresponding control points in the source program, and *structure modifying* optimizations that admit no such mapping. For Structure Preserving optimizations, that cover most high-level optimizations, we provide a general proof rule for translation validation of *structure preserving* transformations, present a tool (voc-64) that implements the proof rule on an EPIC compiler (SGI Pro-64). A summary of the work is in Section 2.

A more challenging class of optimizations does not guarantee such correspondence between control points in the target and the source. An important subclass of these optimizations, which is the focus of the rest of the this paper, is the *reordering transformations* that merely changes the order of execution of statements, without altering, deleting, and adding to them. Typical optimizations belonging to this class are *loop distribution* and *fusion*, *loop tiling*, and *loop interchange*.

For the validation of these reordering transformations we explore "permutation rules" in Section 3. The permutation rules call for calculating which are the reordered statements, and then proving that the reordering preserves the observable semantics.

In some cases, it is impossible to determine during compilation time, whether a desired optimization is legal. This is usually because of limited capability to check effectively that syntactically different index expressions refer to the same array location. One possible remedy to this situation is to adopt an aggressive optimization that performs the loop transformation nevertheless, but keep checking in run-time that no dangerous aliasing occurs. In case the runtime checks detect an aliasing that endanger the validity of the optimization, the code escapes to an unoptimized version of the original loop, where it completes the computation at a slower but guaranteed correct manner. This method, which builds on the theory of the permutation conditions is presented in Section 4.

Thus, the work reported here concerns "run-time verification" on two levels. First, the "translation validation" approach can be viewed as a run-time verification of the optimizing compiler, since in conjunction with every run of the compiler we run the validator tool to check upon the correctness of the compiler run. Then, the speculative optimization approach applies run-time monitoring and verification to the target code – continuously checking the validity of the applied transformation, and escaping to an unoptimized version as soon as this validity is compromised.

1.1 Related Work

The work here is an extension of the work in [ZPFG02]. The work in [Nec00] covers some important aspects of our work. For one, it extends the source programs considered from single-loop programs to 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. In contrast, our work can also be applied to structure-modifying optimizations, such as the ones associated with aggressive loop optimizations which are a major component of optimizations for modern architectures.

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.

2 Translation Validation of Optimizing Compilers

We outline voc, the general strategy for Validation of Optimizing Compilers and describe the theory of validation of structure preserving optimizations. A more detailed description is in [ZPFG02]. We conclude the section with an example of voc-64, our tool for voc of the global optimizations of the SGI PRO-64 compiler.

The compiler receives a *source program* written in some high-level language, translates it into an *Intermediate Representation (IR)*, and then applies a series of optimizations to the program – starting with classical architecture-independent *global* 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.

In order to prove that the target code is a translation of the source code, we 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 then defined in terms of *refinement*, stating that every computation of T corresponds to some computation of S with matching values of the corresponding variables.

The intermediate code is a three-address code. It is described by a *flow graph*, which is a graph representation of the three-address code. Each node in the flow graph represents a *basic block*, that is, a sequence of statements that is executed in its entirety and contains no branches. The edges of the graph represent the flow of control.

2.1 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*,
- Θ an *initial condition* characterizing the initial states of the system, and
- ρ 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. We require that all variables whose values are printed by the program 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 such that every two consecutive states are related by the transition relation.

A transition system $\mathcal{T}$ is called *deterministic* if the observable part of the initial condition uniquely determines the rest of the computation. We restrict our attention to deterministic transition systems and the programs which generate such systems. Thus, to simplify the presentation, we do not consider here programs whose behavior may depend on additional inputs which the program reads throughout the computation. It is straightforward to extend the theory and methods to such intermediate input-driven programs.

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ 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 one-to-one correspondence between the observables of P_S and those of P_T. To simplify the notation, we denote by $X \in \mathcal{O}_S$ and $x \in \mathcal{O}_T$ the corresponding observables in the two systems. A source state s is defined to be *compatible* with the target state t, if s and t agree on their observable parts. That is, $s[X] = t[x]$ for every $x \in \mathcal{O}_T$. We say that P_T is a *correct translation* (*refinement*) of P_S if they are comparable and, for every $\sigma_T : t_0, t_1, \ldots$ a computation of P_T and every $\sigma_S : s_0, s_1, \ldots$ a computation of P_S such that s_0 is compatible with t_0, then σ_T is terminating (finite) iff σ_S is and, in the case of termination, their final states are compatible.

2.2 Translation Validation of Structure Preserving Transformations

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be comparable TSs, where P_S is the *source* and P_T is the *target*. In order to establish that P_T is a correct translation of P_S for the cases that the structure ofP_T does not radically differ from the structure of P_S, we introduce a proof rule, Validate, which is inspired by the computational induction approach ([Flo67]), originally introduced for proving properties of a single program, Rule Validate provides a proof methodology by which one can prove that one program *refines* another. This is achieved by establishing a *control mapping* from target to source locations, a *data abstraction* mapping from source to target variables, and proving that these abstractions are maintained along basic execution paths of the target program.

The proof rule is presented in Fig. 1. There, each TS is assumed to have a *cut-point set* CP. This is a set of blocks that includes the initial and terminal block, as well as at least one block from each of the cycles in the programs' control flow graph. A *simple path* is a path connecting two cut-points, and containing no other cut-point as an intermediate node. We assume that there is at most one simple path between every two cut-points. For each simple path leading from `Bi` to `Bj`, ρ_{ij} describes the transition relation between blocks `Bi` and `Bj`. Typically, such a transition relation contains the condition which enables this path to be traversed, and the data transformation effected by the path. Note that, when the path from `Bi` to `Bj` passes through blocks that are not in the cut-point set, ρ_{ij} is a compressed transition relation that can be computed by the composition of the intermediate transition relation on the path from `Bi` to `Bj`.

The invariants φ_i in part (2) are program annotations that are expected to hold whenever execution visits block `Bi`. They often can be derived from the data flow analysis carried out by an optimizing compiler. Intuitively, their role is to carry information in between basic blocks.

The verification conditions assert that at each (target) transition from `Bi` to `Bj` [5], if the assertion φ_i and the data abstraction hold before the transition, and the transition takes place, then after the transition there exist new source

[5] Recall that we assume that a path described by the transition is simple.

(i) Establish a *control abstraction* κ: $\mathsf{CP}_{_T} \to \mathsf{CP}_{_S}$ that maps initial and terminal blocks of the target into the initial and terminal blocks of the source.

(ii) For each basic block `Bi` in $\mathsf{CP}_{_T}$, form an *invariant* φ_i that may refer only to concrete (target) variables.

(iii) Establish a *data abstraction*

$$\alpha : (p_1 \to V_1 = e_1) \ \wedge \ \cdots \ \wedge \ (p_n \to V_n = e_n)$$

assigning to *some* non-control source state variables $V_i \in V_{_S}$ an expression e_i over the target state variables, conditional on the (target) boolean expression p_i. Note that α may contain more than one clause for the same variable. It is required that for every *observable* source variable $V \in \mathcal{O}_{_S}$ (whose target counterpart is v) and every terminal target block `B,` α implies that $V = v$ at `B.`

(iv) For each pair of basic blocks `Bi` and `Bj` such that there is a simple path from `Bi` to `Bj` in the control graph of $P_{_T}$, let $Paths(\kappa(i), \kappa(j))$ be the set of simple source paths connecting block `B`κ`(i)` to `B`κ`(j)`. We form the verification condition

$$C_{ij}: \qquad \varphi_i \wedge \alpha \wedge \overset{T}{\rho}_{ij} \quad \to \quad \exists {V_{_S}}': (\bigvee_{\pi \in Paths(\kappa(i), \kappa(j))} \overset{S}{\rho}_{\pi}) \wedge \alpha' \wedge \varphi'_j,$$

where $\overset{S}{\rho}_{\pi}$ is the transition relation for the simple source path π.

(v) Establish the validity of all the generated verification conditions.

Fig. 1. The Proof Rule Validate

variables that reflect the corresponding transition in the source, and the data abstraction and the assertion φ_j hold in the new state. Hence, φ_i is used as a hypothesis at the antecedent of the implication C_{ij}. In return, the validator also has to establish that φ_j holds after the transition. Thus, as part of the verification effort, we confirm that the proposed assertions are indeed inductive and hold whenever the corresponding block is visited.

[ZPFG02] contains a discussion, soundness proof, and examples of applications of the of the rule.

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.

Parts 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.

2.3 *Example of* voc-64

The intermediate language of SGI Pro-64 (or SGI for short) is WHIRL. After each round of optimization, the compiler outputs ASCII formatted WHIRL code, which can be read by a parser and translated back into a graphic representation. A description of the tool is in [ZPFG02].

In Fig. 2 we present a program and several of the optimizations applied to it.

```
B0
        n <- 500
        y <- 0
        w <- 1
B1
        WHILE (n >= w)
B2
        w <- w + y * 2 + 3
        w <- y + 1
B3
        RETURN
```

(a) Input Program

```
B0
      n <- 500
      y <- 0
      w <- 1
      IF !(n >= w) GOTO L2
B1
L1:   w <- w + y * 2 + 3
      y <- y + 1
      IF (n >= w) GOTO L1
B2
L2:   RETURN
```

(b) After Loop Inversion

```
B0
      n <- 500
      y <- 0
      w <- 1
B1
L3:   w <- w + y * 2 + 3
      y <- y + 1
      IF (n >= w) GOTO L3
B2
      RETURN
```

(c) After Constant Folding

```
B0
      y <- 0
      w <- 1
B1
L3:   w <- w + y * 2 + 3
      y <- y + 1
      IF (w <= 500) GOTO L3
B2
      RETURN
```

(d) After Copy Propagation and Dead Code Elimination

```
B0
      y <- 0
      w <- 1
      .t1 <- 0
B1
L3:   w <- .t1 + w + 3
      y <- y + 1
      .t1 <- .t1 + 2
      IF (w <= 500) GOTO L3
B2
      RETURN
```

(e) After Strength Reduction

Fig. 2. Stages of Optimization

In Appendix A we present the verification conditions that voc-64 produced for each stage of the optimization. There, we used annotation of (a)–(e) to denote the state of the optimizations the variables/conditions refer to. So, for example, C_{01}^{cd} refers to the verification condition of the target of program (d) above going from `B0` to `B1` relative to the source program (c), and y_e denotes the y variable of program (e). The variable π is the control variable denoting the program counter of the program. The data mapping and assertion are given by:

$$
\begin{array}{ll}
\alpha_{ab}: (\pi_b \in \{1,2\} & \rightarrow \;\; (n_a = n_b) \wedge (w_a = w_b) \wedge (y_a = y_b)) \\
\alpha_{bc}: (\pi_c \in \{1,2\} & \rightarrow \;\; (n_b = n_c) \wedge (w_b = w_c) \wedge (y_b = y_c)) \\
\alpha_{cd}: (\pi_d \in \{1,2\} & \rightarrow \;\; (n_c = 500) \wedge (w_c = w_d) \wedge (y_c = y_d)) \\
\alpha_{de}: (\pi_d \in \{1,2\} & \rightarrow \;\; (w_d = w_e) \wedge (y_d = y_e)) \\
\varphi_3: \;\; (.t1 = 2 \cdot y_e) &
\end{array}
$$

3 Validating Loop Reordering Transformations

A *reordering transformation* is any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statement [AK02]. It preserves a dependence if it preserves the relative execution order of the source and target of that dependence, and thus preserves the meaning of the program. Reordering transformations cover many of the loop transformations, including fusion, distribution, interchange, tiling, unrolling, and reordering of statements within a loop body.

In this section we review the reordering loop transformation and propose "permutation rules" that the validator may use to deal with these transformations.

3.1 Overview of Reordering Loop Transformations

Consider a loop of the form described in Fig. 3. We denote by $\mathcal{I} = \{\boldsymbol{i_1}, \ldots, \boldsymbol{i_N}\}$ the set of values of $\boldsymbol{i} = (i_1, \ldots, i_k)$. We use $<_S$ to denote lexicographical ordering between $\mathcal{I}$ entries, and assume that $\boldsymbol{i_1} <_S \ldots <_S \boldsymbol{i_N}$.

```
for i₁ = 1 to k₁ do
    ...
    for iₘ = 1 to kₘ do
        B1(i₁,...,iₘ)
        ...
        Bℓ(i₁,...,iₘ)
    end
    ...
end
```

Fig. 3. A General Loop

An execution of the loop in Fig. 3 can be described by:

$$\underbrace{\mathtt{B1}(\boldsymbol{i_1}), \mathtt{B2}(\boldsymbol{i_1}), \ldots, \mathtt{B}\ell(\boldsymbol{i_1})}_{\mathtt{B}(\boldsymbol{i_1})}, \quad \ldots, \quad \underbrace{\mathtt{B1}(\boldsymbol{i_N}), \mathtt{B2}(\boldsymbol{i_N}) \ldots, \mathtt{B}\ell(\boldsymbol{i_N})}_{\mathtt{B}(\boldsymbol{i_N})} \qquad (1)$$

A *reordering* transformation is a transformation that causes the execution of the loop to be a permutation of the sequence in (1).

Reordering transformations can be partitioned into two classes: The first reorders the executions of the $\mathtt{B}(\boldsymbol{i})$'s, but leaves each $\mathtt{B}(\boldsymbol{i})$ intact. Examples of such transformations are loop reversal, interchange, tiling, and unrolling.

The second class of reordering transformations permute the execution order of the components $\mathtt{Bj}(\boldsymbol{i})$ inside each $\mathtt{B}(\boldsymbol{i})$, and leaves the order of the $\boldsymbol{i}$'s intact. Transformations belonging to this class are loop fusion and distribution.

There are also hybrid transformations. For example, software pipelining interleaves the executions of the sub-bodies of $\mathtt{B}(\boldsymbol{i})$, and, when projected to each $\boldsymbol{i}$, the $\mathtt{Bj}(\boldsymbol{i})$'s appear in order.

In this paper, we focus on the first class of reordering optimizations. Some common examples of transformations of the first type are presented in Fig. 4. The second column describes the source loop, the third column describes the target loop. We explain the fourth column shortly. Note that for unrolling and tiling we assume that c divides n. For tiling we also assume that d divides m.

Tranf-mation	**Source** ($\mathtt{B}\boldsymbol{i}$)	**Target** ($\mathtt{B}\boldsymbol{j}$)	$P(\boldsymbol{i})$
reversal	`for i = 1,n do` `B(i)`	`for j = n,1 do` `B(j)`	$(n+1-i)$

inter-change	`for i1 = 1,n do`
	`for i2 = 1,m do`
	`B(i1,i2)`
		`for j1 = 1,m do`
		`for j2 = 1,n do`
		`B(j2,j1)`
			(i_2, i_1)

unrolling	`for i = 1,n do`
	`B(i)`
		`for j1 = 1,n by c do`
		`for j2 = j1,j1 + c - 1 do`
		`B(j2)`
			$(\lfloor\frac{i-1}{c}\rfloor+1, i)$

tiling	`for i1 = 1,n do`
	`for i2 = 1,m do`
	`B(i1,i2)`
		`for j1 = 1,n by c`
		`for j2 = 1,m by d`
		`for j3 = j1,j1 + c - 1`
		`for j4 = j2,j2 + d - 1`
		`B(j3,j4)`
			$(c\lfloor\frac{i_1-1}{c}\rfloor+1,$ $d\lfloor\frac{i_2-1}{d}\rfloor+1,$ $i_1, i_2)$

Fig. 4. Some Loop Transformation

For the loop transformations being considered here, let $\mathcal{I}$ denote the vector space of the target loop. Assume that $\mathcal{I} = \{\boldsymbol{j_1}, \ldots, \boldsymbol{j_N}\}$ and that $<_T$ is the order between $\mathcal{I}$'s elements, with $\boldsymbol{j_1} <_T \ldots <_T \boldsymbol{j_N}$. Each transformation in the class can be associated with a *characteristic permutation* P on $[1..N]$, that indicates the order in which the source control elements are executed in the target. That is, if $P(\ell_1) = \ell_2$, then $\mathtt{B}(\boldsymbol{i_{\ell_1}})$ is executed when the target control is $\boldsymbol{j_{\ell_2}}$. It follows that the characteristic permutation P induces a mapping from $\mathcal{I}$ to $\mathcal{I}$. We abuse notation and denote this mapping also by P. Thus, if $P(\ell_1) = \ell_2$, then we also write $P(\boldsymbol{i_{\ell_1}}) = \boldsymbol{j_{\ell_2}}$. The fourth column in Fig. 4 shows the permutation P for each of the examples.

Consider a transformation of the first type. Let $\mathcal{I}$ denote the vector space of the target loop. Assume that $\mathcal{I} = \{\boldsymbol{j_1}, \ldots, \boldsymbol{j_N}\}$ and that $<_T$ is the order between $\mathcal{I}$'s elements, with $\boldsymbol{j_1} <_T \ldots <_T \boldsymbol{j_N}$. Each transformation in the class can by associated with a *characteristic permutation* P on $[1..N]$, that indicates the order in which the source control elements are executed in the target. I.e., if $P(\ell_1) = \ell_2$, then $\mathtt{B}(\boldsymbol{i_{\ell_1}})$ is executed when the target control is $\boldsymbol{j_{\ell_2}}$. It follows that the characteristic permutation P induces a mapping from $\mathcal{I}$ to $\mathcal{I}$. We abuse notation and denote this mapping also by P. Thus, if $P(\ell_1) = \ell_2$, then we also write $P(\boldsymbol{i_{\ell_1}}) = \boldsymbol{j_{\ell_2}}$. The fourth column in Fig. 4 shows the permutation P for each of the examples.

Before venturing into the permutation rules, we need to define the structure of the loop body more carefully: Each `B` consists of a group of statements, some may include (definition/use)-references to array variables. Suppose the array referenced is X (the extension to several arrays is trivial and omitted here). For each `B`, for each statement that *defines* (i.e. writes to) some element in X, let D be a function that maps $\boldsymbol{i}$ (in the source loop) to the array index defined in D. E.g., suppose $\mathtt{B}\ell(i_1, i_2)$ consists of the statement "$\mathtt{X}[\mathtt{i}_1] = \mathtt{X}[\mathtt{i}_1 - 1] + \mathtt{X}[\mathtt{i}_1]$". Then, $D_\ell(i_1, i_2) = i_1$. Similarly, for each statement that *uses* some element in X, let U be the function that maps $\boldsymbol{i}$ to the array index used in U. In the previous example, we have two uses, for the first we have $U_\ell^1(i_1, i_2) = i_2 - 1$ and $U_\ell^2(i_1, i_2) = i_1$.

3.2 Permutation Rules

The simulation-based proof method discussed in previous sections assumes that the source and target have similar structures, thus rule Validate cannot be used to validate many of the loop optimizations. In this section we propose "permutation rules" for validating reordering transformations. The soundness of the permutation 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.

For the sake of this discussion, we only consider *flow dependences* in our permutation rule. The other types of dependence, *anti-* and *output dependence*, can be handled in exactly the same manner. Considering only flow dependences, a reordering transformation is valid if the order of uses of a variable or array element, relative to its definitions,is unchanged. That is, for every ℓ, if in the original loop $X[\ell]$ is defined before it is used, then so it is in the transformed loop. In other words, if for some occurrences of D and U and $\ell_1, \ell_2 \in [1..N]$ such that $\ell_1 < \ell_2$, $D(\boldsymbol{i_{\ell_1}}) = U(\boldsymbol{i_{\ell_2}})$, then in the transformed loop $\mathtt{B}(\boldsymbol{i_{\ell_1}})$ appears before$\mathtt{B}(\boldsymbol{i_{\ell_2}})$, or, equivalently, $P(\boldsymbol{i_{\ell_1}}) <_T P(\boldsymbol{i_{\ell_2}})$.

Consequently, our first permutation rule claims that if $\ell_1 < \ell_2$ and $D(\boldsymbol{i_{\ell_1}}) = U(\boldsymbol{i_{\ell_2}})$, then $P(\ell_1) < P(\ell_2)$. The rule is described in Fig. 5.

If for all $\ell_1, \ell_2 \leq N$,

$$\ell_1 < \ell_2 \ \wedge \ D(\boldsymbol{i_{\ell_1}}) = U(\boldsymbol{i_{\ell_2}}) \quad \Longrightarrow \quad P(\ell_1) < P(\ell_2)$$

then the transformation preserves the meaning of the program

Fig. 5. Permutation Rule for Type-1 Transformations

For example, for loop reversal, since $P(i) = n + 1 - i$ (see Fig. 4), the rule claims that the transformation is valid if whenever $\ell_1 < \ell_2$ and $D(\ell_1) = U(\ell_2)$, $\ell_2 < \ell_1$ should hold, thus the transformation is not valid if there exists ℓ_1 and ℓ_2, $1 \leq \ell_1 \neq \ell_2 \leq n$ such that $D(\ell_1) = U(\ell_2)$.

3.3 Automatic Validation

Validating transformations based on proof rule as in Fig. 5 involves:

(i) Proving the soundness of the permutation rules. This can be accomplished by theorem provers such as PVS [SOR93].

(ii) Proving that the transformations are of the right form, and that P is a permutation (when involved). We ignore such issues, and assume that they compiler handles a fixed known set of transformations, all are easily verified to be of the correct form and using a proper permutation.

(iii) Solving the linear equations to find when the Ds and the Us intersect. Again, we assume that the equations involved are rather simple and can be handled. We also trust to have the solutions supplied by the compiler and to check them for soundness.

In [ZPL00] we proposed permutation rules that deal with general unimodular transformations. While the permutation rules there deal with a much wider set of transformations, including non-reordering transformations and transformations that may alter the loop body, validating the transformations requires much more powerful tools than the transformations here. We believe that, at the price of dealing with a more restricted set of transformations, we obtain a more efficient methodology that can deal with the most commonly used loop transformations. In the next section we use the permutation rules developed here to obtain run-time validation of the reordering transformations.

4 Run-time Validation of Speculative Optimizations

This section gives an overview of *run-time validation of speculative loop optimizations.* That is, using run-time tests to ensure the correctness of loop optimizations when neither the compiler nor a validation tool are able to.

This technique is particularly useful when memory aliasing, due to the use of pointers or arrays, inhibits the static dependence analysis that loop optimizations rely on.

Unlike compiler validation, as discussed in previous sections, run-time validation has not only the task of determining when an optimization has generated incorrect code, but also has the task of recovering from the optimization without aborting the program or producing an incorrect result. It will be possible in some instances to simply adjust the behavior of the optimized code based on run-time tests, so that correctness is preserved while also maintaining much of the performance benefit of the optimization. In other instances, it will be necessary to jump to an unoptimized version of the code.

The particular optimizations to be addressed here are the ones discussed in Section 3. As shown in [GHCP02], software pipelining has also been shown to be particularly amenable to run-time validation.

The work presented here is somewhat preliminary. Thus, we proceed primarily by example, without presenting an entire formalism for validation of speculative optimizations.

4.1 Formal Basis

Suppose we have a loop reordering that performs the transformation

`for I = 1 to` N `do`
`A[`D`(I)] = ...`
`... = ... A[`U`(I)] ...`
`end`
	$\Longrightarrow$	`for I = ` $P^{-1}(1)\ldots P^{-1}(N)$ `do`
		`A[`D`(I)] = ...`
		`... = ... A[`U`(I)] ...`
		`end`

where P is a permutation determining the sequence of values taken on by the index variable `I` in the transformed loop. That is, $P(i) = j$ iff the index variable takes on the value i in the jth iteration of the transformed loop. Thus, $P^{-1}(j)$ gives the value of the index variable in the jth iteration of the transformed loop.

For run-time validation of this transformation we will use, as an invariant to be maintained at run-time, a simplification of the permutation rule presented in Fig. 5. This simple rule is:

$$\forall i, j \leq N, \qquad (i < j) \ \wedge \ D(i) = U(j) \quad \Longrightarrow \quad P(i) < P(j) \tag{2}$$

Intuitively, this says that if `A[`$D(i)$`]` and `A[`$U(j)$`]` refer to the same location, such that the write to that location would have occurred before the read from that location in the original loop, the write must also occur before the read in the transformed loop.

This rule accounts for perfectly nested loops as well, of the form,

```
for i_1 = 1 to N_1 do
 ...
  for i_m = 1 to N_m do
```

```
      A[D(i_1,...,i_m)] = ...
        ... = ... A[U(i_1,...,i_m)] ...
    end
  ...
  end
```

where D and U each return a vector giving the indices into multi-dimensional array `A`.

Rule (2), above, actually governs only the preservation of flow dependence (*aka* true dependence). The complete rule, accounting for anti- and output dependence as well is:

$$\forall i, j \leq N : \ (i < j) \ \wedge \ (D(i) = U(j) \ \vee \ U(i) = D(j) \ \vee \ D(i) = D(j)) \implies \quad P(i) < P(j)$$

For simplicity of the presentation, and without loss of generality, we'll only consider the simpler rule here.

4.2 Safety Properties of Run-Time Validation

The safety properties that must be preserved by the run-time test are:

(i) The test must be able to determine, either precisely or conservatively, if a dependence may be violated by the optimized loop.

(ii) Once a run-time test determines that a dependence may be violated by the optimized loop, there must be an execution path that can be taken to produce the correct result.

(iii) The run-time test must be able to determine that a dependence may be violated *before* the dependence has actually been violated.

This last property, which we refer to as the *testability property*, is, perhaps, overly strict. One can imagine detecting a dependence violation after the violation has occurred, and executing patch-up code to undo the effect of the violation. We do not take this approach, however.

4.3 Efficiency Issues for Run-Time Validation

In order to be worthwhile, run-time validation should satisfy the following qualitative properties:

(i) The run-time test should occur as infrequently as possible.

(ii) The test should be as inexpensive as possible, in terms of time and space.

(iii) The cost of executing the loop when a potential dependence violation is detected should be no greater than the cost of the orginal (unoptimized) loop.

We've developed a set of run-time validation techniques which satisfy these properties to various extents. In this paper, though, we concentrate on describing how run-time tests are used to preserve dependences, rather than addressing their efficiency.

4.4 The Testability Property

The testability property, which states that a potential dependence violation must be detected before the violation actually occurs, is what makes run-time validation particularly difficult in many cases. Without this constraint, there is a reasonably efficient algorithm for testing to see if a transformed loop satisfies rule (2), above. This algorithm is:

Input: Permutation P, functions D, U with ranges $\{1..m\}$
Output: "success" or "failure"
Data structure: `MARK: array[1..`m`] of integer`, with all elements initialized to zero.
Algorithm:

```
for k := P^{-1}(1), ..., P^{-1}(N) do
   MARK[U(k)] := max(Mark[U(k)],k)
   if MARK[D(k)] > k then exit with "failure"
end
exit with "success"
```

To see why this algorithm will detect if rule (2) is violated, note that the only way the rule can be violated is when there exists i, j such that $(i < j) \wedge (D(i) = U(j)) \wedge (P(j) < P(i))$. This is exactly the situation detected by the algorithm, because:

(i) If `k` takes on a value i such that `MARK[`$D(i)$`]` contains a non-zero value j, then it must be the case that $D(i) = U(j)$ because they produce the same index into the `MARK` array. Further, since j was written into `MARK[`$U(j)$`]` before being read as `MARK[`$D(i)$`]` by the algorithm, `k` must have taken on the value j before it took on the value i. Therefore, $P(j) < P(i)$.

(ii) If $j > i$, the rule is violated and the algorithm exits with failure.

The use of `max` in the algorithm is necessary if U(`k`) could produce the same value for several different values of `k`. This algorithm has several desirable properties, namely

(i) The loop index variable `k` iterates over the sequence $P^{-1}(1) \dots P^{-1}(N)$, just as it does in the transformed loop, above.

(ii) The computation of D(`k`) and U(`k`) is the same computation performed by the transformed loop.

Together, these properties mean that a run-time algorithm satisifying them can be integrated into the transformed loop code without a) changing the order of the transformed loop or adding an additional loop and b) having to compute $D(i)$ or $U(i)$ for the sole purpose of the run-time test. In particular, the transformed loop with the run-time test might look like:

```
for i = P^{-1}(1)...P^{-1}(N) do
  w = D(i)
  r = U(i)
  MARK[r] = max(MARK[r],i)
  if MARK[w] > i then goto patchup_code
  A[w] = ...
    ...  = ... A[r] ...
end
```

However, the algorithm has two undesirable properties, namely:

(i) It requires an array MARK of a size equal to the size of A, as well as $2N$ accesses to MARK.

(ii) Worst of all, in the cases where the rule is violated, the algorithm doesn't detect the violation until the write is about to happen – which isn't until after the read has already occurred on a previous iteration.

This last characteristic renders the algorithm useless as a tool for run-time validation, because it does not exhibit the testability property. Intuitively, the problem is that the incorrect value read in a previous iteration may have been used in subsequent iterations, thus rendering the program incorrect.

Does there exist an algorithm that has the desirable properties we want without the undesirable properties listed above? If, as in the code above, we require that the run-time test be integrated into the optimized loop and that each $D(j)$ be computed only during the iteration in which the loop index has the value j, then the answer is "no". To see this, it is sufficient to note that to satisfy the testability property, we would have to be able, in each ith iteration, to answer the question

> "Is there a value $j \in \{P^{-1}(i+1), \ldots, P^{-1}(N)\}$ such that $j < P^{-1}(i)$ and $D(j) = U(P^{-1}(i))$?"

without computing $D(j)$. For an arbitrary function D, this is not possible.

4.5 Restricting D

So far, we've placed no restrictions on the functions D and U. That is, we allow D and U to be sufficiently complex as to prevent static analysis. However in most loops it is not the case that both D and U cannot be analyzed. For example, the transformation of a loop of the form

```
for i = 1 to N do                        for i = P_1 to P_N do
  A[i] = ...                    ==>        A[i] = ...
    ...  = ... A[U(i)] ...                   ...  = ... A[U(i)] ...
end                                      end
```

where $U(i)$ is arbitrarily complex, is not a candidate for static analysis but does satisfy the testability property [6] . A version of the transformed code with the appropriate run-time test might be:

```
for i = P^{-1}(1) to P^{-1}(N) do
  r = U(i)
  if test(i,r) goto escape_code
  A[i] = ...
    ...  = ... A[r] ...
end
```

where

$$\texttt{test}(i,r) \;=\; r < i \;\wedge\; P(r) > P(i)$$

Although this test function appears prohibitively expensive, in practice it is not. The compiler, having created the permutation P, can generate efficient code for the test that is specific for P. For example, if the transformation is loop reversal then the test function is simply $\texttt{test}(i,r) = (r < i)$

If the transformation is loop interchange for two nested loops, then the test function is

$$\begin{aligned}\texttt{test}((i_1,i_2),(r_1,r_2)) &= (r_1,r_2) \prec_{lex} (i_1,i_2) \;\wedge\; (i_2,i_1) \prec_{lex} (r_2,r_1)\\ &= r_1 < i_1 \;\wedge\; r_2 > i_2\end{aligned}$$

4.6 Examples

Dynamic Loop Interchange

In this section, we show how the run-time test for a particular loop interchange example can be automatically derived from the formulation of the general interchange test given above. Suppose the tranformation is:

```
for i = 1 to N                          for j = 1 to M
 for j = 1 to M                           k = 10 - j
   k = 10 - j                             for i = 1 to N
   A[i, j] = A[i-1, j-k] + C   ==>           A[i, j] = A[i-1, j-k] + C
 end                                        end
end                                     end
```

The benefits to this loop interchange are 1) the computation of `k` was able to be moved out of the inner loop and 2) in a language with column-major arrays (such as Fortran), the transformed loop has better locality. Using the formulation of the general validation test in the for loop interchange, a test is inserted as follows:

[6] Note that, in this case, the testability property only holds if restrict we our concern to flow dependence and ignore anti-dependence. A constraint that would allow us to ignore anti-dependence, for example, is $U(i) < i$.

```
for j = 1 to M
  k = 10 - j
  for i = 1 to N
     if test((i, j),(i-1,j-k)) goto escape_code
     A[i, j] = A[i-1, j-k] + C
  end
end
```

where, it is east to see that test$((i,j),(i-1,j-k>) = (i-1<i) \wedge (j-k>j) = (k<0)$. Thus, after inserting the actual test and moving it to the outer loop (since the test is invariant in the inner loop), we arrive at:

```
for j = 1 to M
  k = 10 - j
  if (k < 0)  goto escape_code
  for i = 1 to N
     A[i, j] = A[i-1, j-k] + C
  end
end
```

If the dependence is about to be violated because `k < 0`, then the loop to jump to is the orginal loop to execute the remainder of the iterations. The iterations of the original loop that are left to be executed can be executed by:

```
escape_code:

for ii = 1 to N
 for jj = j to M
     k = 10 - jj
     A[ii, jj] = A[ii-1, jj-k] + C
  end
end
```

where `j` is the same variable as in the transformed loop, above, containing the last value that `j` took on within the optimized loop. This escape code is not difficult to derive from the formal specification of the problem, but space constraints force us to leave it to the reader.

Dynamic Tiling

In the previous example, we were able to generate a version of the original loop that could be executed once the test in the transformed loop detected an impending dependence violation. For a more complicated loop optimization, such as tiling, it may be easier to simply adjust the transformed loop based on a run-time test prior to entering the transformed loop. For example, consider the loop tiling transformation in Fig. 6.

where `k` is a variable whose value is unknown at compile-time. Without a run-time test, this transformation is not necessarily correct. However, it is correct under the condition that `k` $\geq$ `B`. Thus, if a fixed tile size `B` is required,

```
for i = 1 to N                              for ii = 1 to N step B
 for j = 1 to N                              for jj = 1 to N step B
  A[i, j] = A[i-1, j-k] + C      ⟹           for i = ii to ii+B-1
 end                                           for j = jj to jj+B-1
end                                             A[i, j] = A[i-1, j-k] + C
                                               end
                                              end
                                             end
                                            end
```

Fig. 6. Tiling Transformation

the compiler can generate code to test the value of `k` prior to entering the tiled loop and to jump to the original loop if $k < B$. Another alternative is to simply add the assignment `B = min(k,`$maxTileSize$`)` before the tiled loop.

4.7 Architectural Considerations

Adding run-time tests to validate compiler optimizations is becoming increasingly tractable due to, among other things, the emergence of new classes of processors that exhibit 1) the ability to exploit instruction-level parallelism (ILP) and 2) hardware features that reduce the cost of run-time tests and the compensation code when a dependence is about to be violated.

Most modern processors have multiple functional units for exploiting instruction level parallelism, either via dynamic scheduling of multiple instructions simultaneously on superscalar machines or via compiler-specified multi-operation instructions on VLIW/EPIC processors. In fact, the challenge with these machines has been to find sufficient ILP in programs in order to fully utilize all functional units each cycle. Thus, the additional tests required for run-time validation can often be scheduled in unused slots in the instruction schedule.

Some hardware features that aid in the testing for, and compensating for, dependence violations can be found on the VLIW/EPIC class of machines exemplified by the Intel IA-64 architecture. The *dynamic disambiguation* feature, which provides several instructions that test for aliasing, can be used to implement low-cost run-time validation tests. Once a run-time test has determined that a dependence violation is about to occur, the *predication* feature of the IA-64 can be used to disable instructions in such as way as to preserve the correctness of the code. For more detail on this last point, as applied to run-time validation of software pipelining, see [GHCP02].

5 Conclusion

We reviewed the translation validation approach, its advantages, the main proof rule used for structure preserving transformations, and described a tool, voc-64, that generates verification conditions for SGI Pro-64.

We then turn to optimizations that cause major changes in the structure of the code. For reordering transformations we propose special permutation loops that can easily deal with the most common optimizations. For transformations whose validity is hard, or even impossible, to check at compile-time, we propose a *run-time translation validation* that allows for aggressive optimization that while continuously guaranteeing that no dangerous aliasing occurs. When a problem is detected, the code escapes to an unoptimized version of the original loop, where it completes the computation at a slower but guaranteed correct manner.

References

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

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

[GHCP02] B. Goldberg, C. Huneycutt, E. Chapman, and K. Palem. Software bubbles: Using predication to compensate for aliasing in software pipelines. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002.

[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.

[Nec97] G.C. Necula. Proof-carrying code. In *POPL'97*, 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 *CAV'99*, 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 *TACAS'98*, 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.

[ZPFG02] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjaming Goldberg. Voc: A translation validator for optimizing compilers. *ENTCS*, 65(2), 2002.

[ZPL00] L. Zuck, A. Pnueli, and R. Leviathan. Validations of optimizing compliers. Technical report, Weizmann Institute of Science, 2000.

A Control Mapping Generated by voc-64

The control mapping generated by voc-64 for the example in Section 2.3 is:

C^{ab}_{01} : $(\pi_b = 0 \wedge \pi'_b = 1 \wedge n'_b = 500 \wedge w'_b = 1 \wedge y'_b = 0 \wedge 500 \geq 1 \wedge \alpha_{ab}) \rightarrow$
$(\pi_a = 0 \wedge \pi'_a = 2 \wedge n'_a = 500 \wedge w'_a = 1 \wedge y'_a = 0 \wedge 500 \geq 1 \wedge \alpha'_{ab})$
C^{ab}_{02} : $(\pi_b = 0 \wedge \pi'_b = 2 \wedge n'_b = 500 \wedge w'_b = 1 \wedge y'_b = 0 \wedge \neg(500 \geq 1) \wedge \alpha_{ab}) \rightarrow$
$(\pi_a = 0 \wedge \pi'_a = 3 \wedge n'_a = 500 \wedge w'_a = 1 \wedge y'_a = 0 \wedge \neg(500 \geq 1) \wedge \alpha'_{ab})$
C^{ab}_{11} : $(\pi_b = 1 \wedge \pi'_b = 1 \wedge w'_b = w_b + 2 \cdot y_b + 3 \wedge y'_b = y_b + 1 \wedge (n_b \geq w_b + 2 \cdot y_b + 3)$
$\wedge \alpha_{ab}) \rightarrow (\pi_a = 2 \wedge \pi'_a = 2 \wedge w'_a = w_a + 2 \cdot y_a + 3 \wedge y'_a = y_a + 1$
$\wedge (n_a \geq w_a + 2 \cdot y_a + 3) \wedge \alpha'_{ab})$
C^{ab}_{12} : $(\pi_b = 1 \wedge \pi'_b = 2 \wedge w'_b = w_b + 2 \cdot y_b + 3 \wedge y'_b = y_b + 1 \wedge \neg(n_b \geq w_b + 2 \cdot y_b + 3)$
$\wedge \alpha_{ab}) \rightarrow (\pi_a = 2 \wedge \pi'_a = 3 \wedge w'_a = w_a + 2 \cdot y_a + 3 \wedge y'_a = y_a + 1$
$\wedge \neg(n_a \geq w_a + 2 \cdot y_a + 3) \wedge \alpha'_{ab})$
C^{bc}_{01} : $(\pi_c = 0 \wedge \pi'_c = 1 \wedge n'_c = 500 \wedge w'_c = 1 \wedge y'_c = 0 \wedge \alpha_{bc}) \rightarrow$
$(\pi_b = 0 \wedge \pi'_b = 1 \wedge n'_b = 500 \wedge w'_b = 1 \wedge y'_b = 0 \wedge 500 \geq 1 \wedge \alpha'_{bc})$
C^{bc}_{11} : $(\pi_c = 1 \wedge \pi'_c = 1 \wedge w'_c = w_c + 2 \cdot y_c + 3 \wedge y'_c = y_c + 1 \wedge (n_c \geq w_c + 2 \cdot y_c + 3)$
$\wedge \alpha_{bc}) \rightarrow (\pi_b = 1 \wedge \pi'_b = 1 \wedge w'_b = w_b + 2 \cdot y_b + 3 \wedge y'_b = y_b + 1$
$\wedge (n_b \geq w_b + 2 \cdot y_b + 3) \wedge \alpha'_{bc})$
C^{bc}_{12} : $(\pi_c = 1 \wedge \pi'_c = 2 \wedge w'_c = w_c + 2 \cdot y_c + 3 \wedge y'_c = y_c + 1 \wedge \neg(n_b \geq w_c + 2 \cdot y_c + 3)$
$\wedge \alpha_{bc}) \rightarrow (\pi_b = 1 \wedge \pi'_b = 2 \wedge w'_b = w_b + 2 \cdot y_b + 3 \wedge y'_b = y_b + 1 \wedge$
$\neg(n_b \geq w_b + 2 \cdot y_b + 3) \wedge \alpha'_{bc})$
C^{cd}_{01} : $(\pi_d = 0 \wedge \pi'_d = 1 \wedge w'_d = 1 \wedge y'_d = 0 \wedge \alpha_{cd}) \rightarrow$
$(\pi_c = 0 \wedge \pi'_c = 1 \wedge n'_c = 500 \wedge w'_c = 1 \wedge y'_c = 0 \wedge \alpha'_{cd})$
C^{cd}_{11} : $(\pi_d = 1 \wedge \pi'_d = 1 \wedge w'_d = w_d + 2 \cdot y_d + 3 \wedge y'_d = y_d + 1 \wedge (500 \geq w_d + 2 \cdot y_d + 3)$
$\wedge \alpha_{cd}) \rightarrow (\pi_c = 1 \wedge \pi'_c = 1 \wedge w'_c = w_c + 2 \cdot y_c + 3 \wedge y'_c = y_c + 1$
$\wedge (n_c \geq w_c + 2 \cdot y_c + 3) \wedge \alpha'_{cd})$
C^{cd}_{12} : $(\pi_d = 1 \wedge \pi'_d = 2 \wedge w'_d = w_d + 2 \cdot y_d + 3 \wedge y'_d = y_d + 1 \wedge \neg(500 \geq w_d + 2 \cdot y_d + 3)$
$\wedge \alpha_{cd}) \rightarrow (\pi_c = 1 \wedge \pi'_c = 2 \wedge w'_c = w_c + 2 \cdot y_c + 3 \wedge y'_c = y_c + 1 \wedge$
$\neg(n_c \geq w_c + 2 \cdot y_c + 3) \wedge \alpha'_{cd})$
C^{de}_{01} : $(\pi_e = 0 \wedge \pi'_e = 1 \wedge .t1' = 0 \wedge w'_e = 1 \wedge y'_e = 0 \wedge \alpha_{de}) \rightarrow$
$(\pi_d = 0 \wedge \pi'_d = 1 \wedge w'_d = 1 \wedge y'_d = 0 \wedge \alpha'_{de} \wedge \varphi')$
C^{de}_{11} : $(\pi_e = 1 \wedge \pi'_e = 1 \wedge .t1' = .t1 + 2 \wedge w'_e = .t1 + w_e + 3 \wedge y'_e = y_e + 1 \wedge$
$(500 \geq .t1 + w + 3) \wedge \alpha_{de} \wedge \varphi)(\pi_d = 1 \wedge \pi'_d = 1 \wedge w'_d = w_d + 2 \cdot y_d + 3 \rightarrow$
$\wedge y'_d = y_d + 1 \wedge (500 \geq w_d + 2 \cdot y_d + 3) \wedge \alpha'_{de} \wedge \varphi')$
C^{de}_{12} : $(\pi_e = 1 \wedge \pi'_e = 2 \wedge .t1' = .t1 + 2 \wedge w'_e = .t1 + w_e + 3 \wedge y'_e = y_e + 1 \wedge$
$\neg(500 \geq .t1 + w + 3) \wedge \alpha_{de} \wedge \varphi) \rightarrow (\pi_d = 1 \wedge \pi'_d = 2 \wedge w'_d = w_d + 2 \cdot y_d + 3 \wedge$
$y'_d = y_d + 1 \wedge \neg(500 \geq w_d + 2 \cdot y_d + 3) \wedge \alpha'_{de})$