Translation and Run-Time Validation of Loop Transformations

COL731 Course Presentation

Kavya Chopra

IIT Delhi

Nov 2023

Introduction

Translation Validation

- Formally verifying compilers is difficult, due to a perpetually evolving and bulky codebase.
- Solution: Instead of verifying the translator (in this case, the compiler), verify the *translation* from the source to the target instead after every run of the compiler.
- Explicitly Parallel Instruction Computing (EPIC) refers to architectures in which features are provided to facilitate compiler enhancements of instruction-level parallelism (ILP) in all programs, while keeping hardware complexity relatively low .
- Compilers targeted at these architectures typically perform more aggressive optimizations, especially those facilitating parallelism, as the onus to schedule instructions is on the compiler and not the CPU hardware

Story so Far

- **Alive**: Verifies peephole optimizations
  - Focuses more on "clever" mathematical/instruction-level optimizations
  - Doesn't validate optimisations in which there's greater variance in program structure
  - No support for constructs like loops and branches
- **CoVaC**: Verifies *structure preserving* optimizations.
  - The framework demonstrated is mostly used for consonant programs (each loop in the target program corresponds to a loop in the source program), and uses simulation relations between source and target code.
  - Correspondence between certain points in the target and the source are needed (cut-point mapping)
  - Doesn't validate transformations that modify structure in any way (eg loop tiling)

Dealing with Structure Modifying optimizations

- **VOC64**: Verifies *structure modifying* optimizations
    - Here, we discuss loop reordering transformations, which change the order of the execution of statements without adding to or deleting from them (eg loop interchange and tiling)
    - There's no direct correspondence between control points in the source and target code, so the CoVaC framework is difficult to apply here.
    - We define a set of "permutation rules", that prove that the reordered statements preserve the semantics. They, however, require the ability to prove that 2 loop free code fragments are equivalent, and this is something CoVaC does address, so we abstract this functionality out as a simulation relation $\sim$
    - Additionally, these transformations may only be correct subject to certain runtime checks, so we provide a mechanism to do runtime validation as well.

Validating SPOs

Defining $\sim$

- We wish to define a simulation relation between two program fragments P and Q such that $P \sim Q$ implies that P and Q are equivalent. We use a new rule, `VALIDATE`, to accomplish this.
- Transition Systems (TS): State machines characterised by:
  - $\mathcal{V}$: a set of state variables
  - $\mathcal{O} \subset \mathcal{V}$: a set of observable variables
  - Θ: an initial condition characterising the initial states of the system
  - ρ: a transition relation, relating a state to its possible successors (edges)
- Each TS has a variable *pc*, which represents the program location counter.
- We use the transition relations to determine a generalized transition relation, which describes the effect of an execution path of a program.

Example

```
B0:
n <- 500
y <- 0
w <- w + 1
IF !(n >= w) GOTO B2
B1: ...
B2:
```

We have 2 transition relations associated with this block, namely

$$pc = B0 \wedge n' = 500 \wedge y' = 0 \wedge w' = w + 1 \wedge n' \geq w' \wedge pc' = B1$$

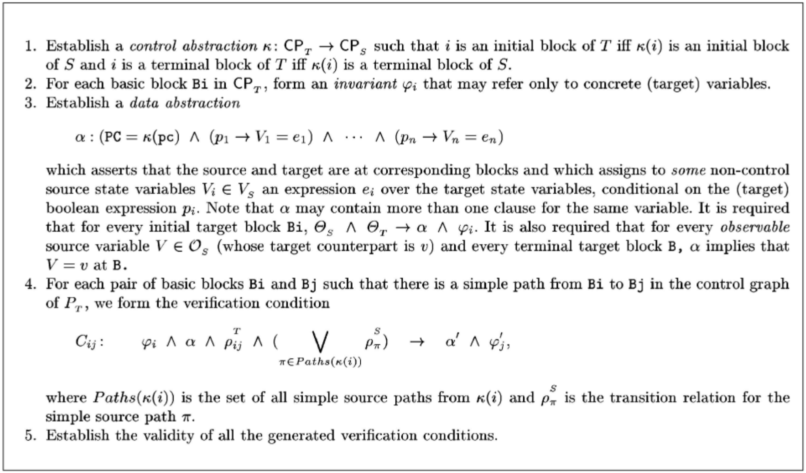
$$pc = B0 \wedge n' = 500 \wedge y' = 0 \wedge w' = w + 1 \wedge n' < w' \wedge pc' = B2$$

The complete transition relation is the disjunct of all these generalised transition relations.

Some Notation:

- Computation: a sequence of states
- 2 transition systems are said to be **comparable** if we have a bijective mapping between their observables
- A source state *s* is said to be **compatible** with a target state *t* if *s* and *t* agree on their observables
- A TS P_t is said to be a correct translation/refinement of a TS P_s if
  1. P_t and P_s are comparable
  2. For all computations in σ_s in source and σ_t in target such that their initial states are **compatible**, σ_t terminates iff σ_s does, and if they do terminate, their final states are **compatible**.
- Each TS has a cut-point set CP, which is a set of blocks that includes all initial and terminal blocks, as well as atleast one block from each of the cycles in the programs' CFG.
- A **simple path** is a path connecting 2 cutpoints with no other cut-point as an intermediate node.

The VALIDATE Rule



1. Establish a *control abstraction* $\kappa\colon \mathsf{CP}_T \to \mathsf{CP}_S$ such that i is an initial block of T iff $\kappa(i)$ is an initial block of S and i is a terminal block of T iff $\kappa(i)$ is a terminal block of S.
2. For each basic block `Bi` in CP_T, form an *invariant* φ_i that may refer only to concrete (target) variables.
3. Establish a *data abstraction*

   $$\alpha : (\mathtt{PC} = \kappa(\mathtt{pc})\ \wedge\ (p_1 \to V_1 = e_1)\ \wedge\ \cdots\ \wedge\ (p_n \to V_n = e_n))$$

   which asserts that the source and target are at corresponding blocks and which assigns 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 initial target block `Bi`, $\Theta_S\ \wedge\ \Theta_T \to \alpha\ \wedge\ \varphi_i$. It is also 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`.
4. 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, we form the verification condition

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

   where $Paths(\kappa(i))$ is the set of all simple source paths from $\kappa(i)$ and $\overset{S}{\rho}_{\pi}$ is the transition relation for the simple source path π.
5. Establish the validity of all the generated verification conditions.

Validating SMOs

Notation

- Mathematical formulation of a loop:

$$\textbf{for } \vec{i} \in \mathcal{I} \textbf{ by } < ¡ \textbf{ do } \mathcal{B}(\vec{i})$$

where

- $\mathcal{I}$ is the iteration space
- $\vec{i}$ denotes an iteration vector
- $<$ ¡ is the ordering by which the various points of $\mathcal{I}$ are traversed.
- We can characterise $\mathcal{I}$ via inequalities
- $\mathcal{I} = \{(i_1, i_2....i_m) \mid L_1 \leq i_1 \leq H_1 \wedge L_2 \leq i_2 \leq H_2... \wedge L_m \leq i_m \leq H_m\}$

Notation: Contd

- Mathematical formulation of a loop transformation

  for $\vec{i} \in \mathcal{I}$ **by** $<_{\mathrm{i}}$ **do** $\mathcal{B}(\vec{i}) \implies$ **for** $\vec{j} \in \mathcal{J}$ **by** $<_{\mathrm{j}}$ **do** $\mathcal{B}(F(\vec{j}))$

  where F is a transformation mapping $\mathcal{J}$ to $\mathcal{I}$ (and is *generally* linear)
- Since these are reordering transformations and the number of statements are preserved, we want $\mid \mathcal{I} \mid = \mid \mathcal{J} \mid$, and since we should have only 1 instance in the target corresponding to each instance in the source, we need $F : \mathcal{J} \mapsto \mathcal{I}$ to be a bijective mapping. We can ensure this by defining an inverse mapping $F^{-1} : \mathcal{I} \mapsto \mathcal{J}$

Examples

	Interchange	**skewing**

Source	`for` $\mathtt{i}_1 = 1,\mathtt{n}$ `do`
	    `for` $\mathtt{i}_2 = 1,\mathtt{m}$ `do`
	        $\mathtt{B}(i_1, i_2)$
		`for` $\mathtt{i}_1 = 1,\mathtt{n}$ `do`
		    `for` $\mathtt{i}_2 = 1,\mathtt{n}$ `do`
		        $\mathtt{B}(i_1, i_2)$

Target	`for` $\mathtt{j}_1 = 1,\mathtt{m}$ `do`
	    `for` $\mathtt{j}_2 = 1,\mathtt{n}$ `do`
	        $\mathtt{B}(j_2, j_1)$
		`for` $\mathtt{j}_1 = 1,\mathtt{n}$ `do`
		    `for` $\mathtt{j}_2 = \mathtt{j}_1 + 1, \mathtt{j}_1 + \mathtt{m}$ `do`
		        $\mathtt{B}(j_1, j_2 - j_1)$

$\mathcal{I}$	$\{1,\ldots,n\} \times \{1,\ldots,m\}$	$\{1,\ldots,n\} \times \{1,\ldots,n\}$
$\mathcal{J}$	$\{1,\ldots,m\} \times \{1,\ldots,n\}$	$\{(j_1, j_2) : 1 \le j_1 \le n \;\wedge\; j_1 + 1 \le j_2 \le j_1 + n\}$
$\vec{i} \prec_{\mathcal{I}} \vec{i'}$	$\vec{i} <_{\text{lex}} \vec{i'}$	$\vec{i} <_{\text{lex}} \vec{i'}$
$\vec{j} \prec_{\mathcal{J}} \vec{j'}$	$\vec{j} <_{\text{lex}} \vec{j'}$	$\vec{j} <_{\text{lex}} \vec{j'}$
$F(\vec{j})$	(j_2, j_1)	$(j_1, j_2 - j_1)$
$F^{-1}(\vec{i})$	(i_2, i_1)	$(i_1, i_1 + i_2)$

More Examples

	Reversal	**Tiling**

Source	for i = 1, n do
	  B(i)
		for i$_1$ = 1, n do
		  for i$_2$ = 1, m do
		    B(i_1, i_2)

Target	for j = n, 1 do
	  B(j)
		for j$_1$ = 1, n by c do
		  for j$_2$ = 1, m by d do
		    for j$_3$ = j$_1$, j$_1$ + c − 1 do
		      for j$_4$ = j$_2$, j$_2$ + d − 1 do
		        B(j_3, j_4)

$\mathcal{I}$	$\{1, \ldots, n\}$	$\{1, \ldots, n\} \times \{1, \ldots, m\}$
$\mathcal{J}$	$\{1, \ldots, n\}$	$\{(j_1, j_2, j_3, j_4) : 1 \leq j_1 \leq n \ \wedge \ j_1 \equiv 1 \bmod c \ \wedge$ $1 \leq j_2 \leq m \ \wedge \ j_2 \equiv 1 \bmod d \ \wedge$ $j_1 \leq j_3 < j_1 + c \ \wedge$ $j_2 \leq j_4 < j_2 + d\}$
$\vec{i} \prec_{\mathcal{I}} \vec{i'}$	$i < i'$	$\vec{i} <_{\text{lex}} \vec{i'}$
$\vec{j} \prec_{\mathcal{J}} \vec{j'}$	$j > j'$	$\vec{j} <_{\text{lex}} \vec{j'}$
$F(\vec{j})$	j	(j_3, j_4)
$F^{-1}(\vec{i})$	i	$(c\lfloor \frac{i_1-1}{c} \rfloor + 1, d\lfloor \frac{i_2-1}{d} \rfloor + 1, i_1, i_2)$

Permutation Rules

- Assume that we have a simulation relation between two program fragments P and Q, where $P \sim Q$ denotes that the result of executing P and Q starting from an arbitrary state is the same (more on this later)
- $\sim$ is reflexive, transitive, and closed under sequential composition, i.e. $P \sim Q \implies P; R \sim Q; R$
- We say that a transformation is valid when:
  - $F : \mathcal{J} \mapsto \mathcal{I}$ is a bijective mapping.
  - $\forall \vec{i_1}, \vec{i_2} \in \mathcal{I} : \vec{i_1} <_{\text{i}} \vec{i_2} \wedge F^{-1}(\vec{i_2}) <_{\text{j}} F^{-1}(\vec{i_1}) \implies \mathcal{B}(\vec{i_1}); \mathcal{B}(\vec{i_2}) \sim \mathcal{B}(\vec{i_2}); \mathcal{B}(\vec{i_1})$
  - Intuitively, the second rule says that if we're reversing the relative order of execution 2 statements while moving from the source to the target, then the result after their cumulative execution should be the same.

Soundness of permute:

Permute has 3 premises:

$$\frac{\begin{array}{lll} \text{R1. } \forall \vec{i} \in \mathcal{I} : \exists \vec{j} \in \mathcal{J} : & \vec{i} = F(\vec{j}) & \\ \text{R2. } \forall \vec{j}_1 \neq \vec{j}_2 \in \mathcal{J} : & F(\vec{j}_1) \neq F(\vec{j}_2) & \\ \text{R3. } \forall \vec{i}_1, \vec{i}_2 \in \mathcal{I} : & \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \wedge F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1) & \Longrightarrow \quad \mathtt{B}(\vec{i}_1);\mathtt{B}(\vec{i}_2) \sim \mathtt{B}(\vec{i}_2);\mathtt{B}(\vec{i}_1) \end{array}}{\textbf{for } \vec{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\vec{i}) \quad \sim \quad \textbf{for } \vec{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\vec{j}))}$$

- To prove its soundness, we assume that the iteration space has a given size (say m), and then assume that our transformation is valid for a k-length prefix of the ordered iteration space, i.e.

$$\textbf{for } \vec{i} \in \mathcal{I}_k \textbf{ by } <_{\text{i}} \textbf{ do } \mathcal{B}(\vec{i}) \sim \textbf{for } \vec{j} \in \mathcal{J}_k \textbf{ by } <_{\text{j}} \textbf{ do } \mathcal{B}(F(\vec{j}))$$

where

- $\mathcal{I} = \{\vec{i_1}, \vec{i_2} \ldots . \vec{i_m}\}$ and $\vec{i_1} <_{\text{i}} \vec{i_2} \ldots . <_{\text{i}} \vec{i_m}$
- $\mathcal{I}_k = \{\vec{i_1}, \vec{i_2} \ldots . \vec{i_k}\}$
- $\mathcal{J}_k = F^{-1}(\mathcal{I}_k)$

Soundness of permute (Contd):

$$\textbf{for}\ \vec{i} \in \mathcal{I}_{k+1}\ \textbf{by} <_{\mathrm{i}}\ \textbf{do}\ \mathcal{B}(\vec{i}) \sim \textbf{for}\ \vec{i} \in \mathcal{I}_{k}\ \textbf{by} <_{\mathrm{i}}\ \textbf{do}\ \mathcal{B}(F(\vec{i})); \mathcal{B}(\vec{i}_{k+1})$$

$$\textbf{for}\ \vec{i} \in \mathcal{I}_{k}\ \textbf{by} <_{\mathrm{j}}\ \textbf{do}\ \mathcal{B}(\vec{i}); \mathcal{B}(\vec{i}_{k+1}) \sim$$

$$\textbf{for}\ \vec{j} \in \mathcal{J}_{k}\ \textbf{by} <_{\mathrm{j}}\ \textbf{do}\ \mathcal{B}(F(\vec{i})); \mathcal{B}(\vec{i}_{k+1})$$

$$\textbf{for}\ \vec{i} \in \mathcal{I}_{k}\ \textbf{by} <_{\mathrm{j}}\ \textbf{do}\ \mathcal{B}(F(\vec{i})); \mathcal{B}(\vec{i}_{k+1}) \sim$$

$$\textbf{for}\ \vec{j} \in \mathcal{J}_{k}\ \textbf{by} <_{\mathrm{j}}\ \textbf{do}\ \mathcal{B}(F(\vec{i})); \mathcal{B}(F(F^{-1}(\vec{i}_{k+1})))$$

If $F^{-1}(\vec{i}_{k+1}) \succ_{\mathrm{j}} \vec{j}_a$ where $1 \leq a \leq k$, then we're done, otherwise, we find the minimal index l such that $F^{-1}(\vec{i}_{k+1}) <_{\mathrm{j}} \vec{j}_l$, and then shift $F^{-1}(\vec{i}_{k+1})$ between $\vec{j}_{l-1}$ and $\vec{j}_l$. This transformation is sound due to P3 of `PERMUTE` , and proves the desired claim.

Permute in Action: Loop Interchange

Consider the following example:

```
for i = 1, N do                              for j = 2, M do
    for j = 2, M do              ⟹               for i = 1, N do
        a[i,j] = a[i-1, j-1] + c                     a[i,j] = a[i-1, j-1] + c
```

What conditions do we give to an automated theorem prover so that it can validate this transformation?

Specifying the transformation

1. Domains:
   1.1 $\mathcal{I} = \{(i,j) \mid 1 \leq i \leq N, 2 \leq j \leq M\}$
   1.2 $\mathcal{J} = \{(j,i) \mid 1 \leq i \leq N, 2 \leq j \leq M\}$
2. Relations:

$$(i_1,j_1) <_{\text{i}} (i_2,j_2) \iff (i_1 < i_2) \vee ((i_1 = i_2) \wedge (j_1 < j_2))$$

$$(j_1,i_1) <_{\text{j}} (j_2,i_2) \iff (j_1 < j_2) \vee ((j_1 = j_2) \wedge (i_1 < i_2))$$

3. Mappings:

$$F(j,i) = (i,j)$$

$$F^{-1}(i,j) = (j,i)$$

4. Conditions:

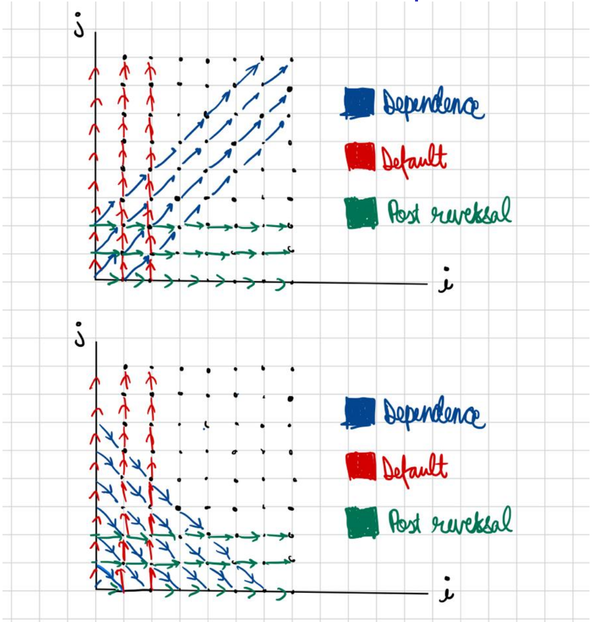
$$\forall (i_1,j_1),(i_2,j_2) \in \mathcal{I}$$

$$(i_1,j_1) <_{\text{i}} (i_2,j_2) \wedge (j_2,i_2) <_{\text{i}} (j_1,i_1) \implies$$

$$\mathcal{B}(i_1,j_1);\mathcal{B}(i_2,j_2) \sim \mathcal{B}(i_2,j_2);\mathcal{B}(i_1,j_1)$$

where $\mathcal{B}(i,j)$ is `a[i,j] = a[i-1, j-1]+ c`

Visual Explanation



Permute in Action: Tiling

Consider the following example:

```
for i = 1, N do
    for j = 1,M do
        for k = 1,L do
            a[i,j] = a[i,j] + e[i,k] * b[k,j]
```

$\Longrightarrow$

```
for ii = 1,N by c do
    for jj = 1,M by c do
        for kk = 1,L by c do
            for i = ii, ii+c-1 do
                for j = jj,jj+c-1 do
                    for k = 1,kk+c-1 do
                        a[i,j] = a[i,j] + e[i,k] * b[k,j]
```

What conditions do we give to an automated theorem prover so that it can validate this transformation?

Specifying the transformation

1. Domains: $\mathcal{I} = \{(i,j,k) \mid 1 \le i \le N, 1 \le j \le M, 1 \le k \le N\}$
$\mathcal{J} = \{(ii,jj,kk,i,j,k) \mid 1 \le i \le N, 1 \le j \le M, 1 \le k \le N, ii \le i \le ii+c, jj \le j \le jj+c, kk \le k \le kk+c, ii \equiv 1 \pmod{c}, jj \equiv 1 \pmod{c}, kk \equiv 1 \pmod{c}\}$
2. Relations:

$$(i_1,j_1,k_1) <_{\mathrm{i}} (i_2,j_2,k_2) \iff (i_1 < i_2) \vee ((i_1 = i_2) \wedge (j_1 < j_2))$$
$$\vee ((i_1 = i_2) \wedge (j_1 = j_2) \wedge (k_1 < k_2))$$
$$(ii_1,jj_1,kk_1,i_1,j_1,k_1) <_{\mathrm{j}} (ii_2,jj_2,kk_2,i_2,j_2,k_2) \iff ((i_1 < i_2) \vee$$
$$((i_1 = i_2) \wedge (j_1 < j_2)) \vee ((i_1 = i_2) \wedge (j_1 = j_2) \wedge (k_1 < k_2))) \vee$$
$$((i_1 = i_2) \wedge (j_1 = j_2) \wedge (k_1 = k_2) \wedge (ii_1 < ii_2)) \vee$$
$$((i_1 = i_2) \wedge (j_1 = j_2) \wedge (k_1 = k_2) \wedge (ii_1 = ii_2) \wedge (jj_1 < jj_2)) \vee$$
$$((i_1 = i_2) \wedge (j_1 = j_2) \wedge (k_1 = k_2) \wedge (ii_1 = ii_2) \wedge (jj_1 = jj_2) \wedge (kk_1 < kk_2)$$

Specifying the transformation (Contd)

3. Mappings:

$$F(ii, jj, kk, i, j, k) = (i, j, k)$$

$$F^{-1}(i, j, k) = (c * \lfloor \frac{i-1}{c} \rfloor + 1, c * \lfloor \frac{j-1}{c} \rfloor + 1, c * \lfloor \frac{k-1}{c} \rfloor + 1, i, j, k)$$

Motivating a simplified Permute

- The current version of the Permute rule is a more general statement than the typical dependence ordering rules we've studied in class
- For instance, consider the following example:

```
for i = 1,n do
    for j = 1,m do
        a[i,j] = a[i-1,j+1] - a[i-1,j+1]
```

$\Longrightarrow$

```
for j = 1,m do
    for i = 1,n do
        a[i,j] = a[i-1,j+1] - a[i-1,j+1]
```

- The theorem prover reasons purely on observable behaviour instead of analyzing the data dependencies between arrays, so it'd validate this correctly.
- However, since compilers don't perform these kind of transformations even if the observable behaviour between source and target is the same, we simplify our permute rule

The simplified Permute Rule

```
for $\vec{i} \in \mathcal{I}$ by $\prec_{\mathcal{I}}$ do
    a[$D(\vec{i})$] = ...
    ...= a[$U(\vec{i})$]
```

$$\Longrightarrow$$

```
for $\vec{j} \in \mathcal{J}$ by $\prec_{\mathcal{J}}$ do
    a[$D(F(\vec{j}))$] = ...
    ...= a[$U(F(\vec{j}))$]
```

where D is the array element corresponding to the "write", U is the array element corresponding to the "read", and F is a bijection from $\mathcal{I}$ to $\mathcal{J}$. There are no other reads or writes to arrays in the loop. Then, we have:

$$\forall \vec{i_1}, \vec{i_2} \in \mathcal{I} : \vec{i_1} <_{¡} \vec{i_2} \wedge (D(\vec{i_1}) = U(\vec{i_2}) \vee U(\vec{i_1}) = D(\vec{i_2}) \vee$$

$$D(\vec{i_1}) = D(\vec{i_2})) \implies F^{-\vec{1}}(i_1) <_j F^{-\vec{1}}(i_2)$$

Runtime Validation

What is Runtime Validation

- If neither the compiler nor a validation tool can ascertain if the transformation is correct at runtime, then we use runtime tests to do so.
- Eg: aliasing may inhibit the static dependence analysis that loop optimizations rely on
- Two-fold job:
    1. Determine when an optimization has generated incorrect code
    2. Recover from the optimization without aborting the program or producing a wrong result

Formal Basis:

```
for I = 1, $N$ do                      for i = $P^{-1}(1) \ldots P^{-1}(N)$ do
    a[$D$(I)] = ...         $\Longrightarrow$       a[$D$(i)] = ...
    ...= ...a[$U$(I)] ...                   ...= ...a[$U$(i)] ...
```

- P here is a permutation that determines the sequence of values that the index variable takes on in the transformed loop, i.e. $P(i) = j$ iff the index variable takes on the value i in the jth iteration of the transformed loop. - For this transformation, a version of the simplified permute rule that addresses only flow dependence is

$$\forall i, j \le N \mid i < j \wedge D(i) = U(j) \implies P(i) < P(j)$$

- The complete rule, accounting for all dependencies is :

$$\forall i, j \le N \mid i < j \wedge (D(i) = U(j) \vee U(i) = D(j) \vee D(i) = D(j)) \implies P(i) < P(j)$$

- We'll just consider the one accounting for flow dependence for simplicity

Safety Properties of Runtime Validation:

1. The test must be able to determine, either precisely or conservatively, if a dependence may be violated by the optimized loop
2. 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.
3. **Testability Property** : The run-time test must be able to determine that a dependence may be violated before the dependence has actually been violated. (This isn't a hard constraint; one can execute patch-up code that makes up for the effect of the violation. But we impose it here for generalisability.)

Efficiency Issues for Runtime Validation

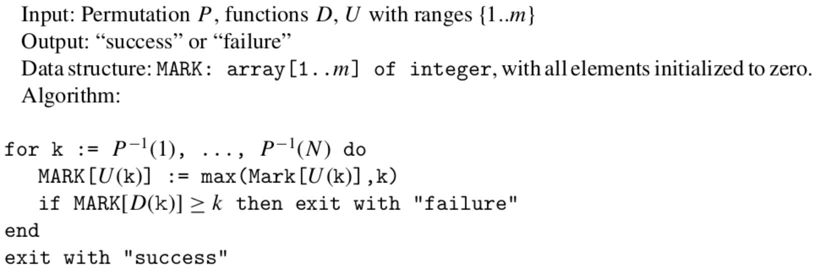
1. The runtime tests should occur as infrequently as possible
2. The test should be as inexpensive as possible, in terms of time and space.
3. The cost of executing the loop when a potential dependence violation is detected should be no greater than the cost of the orginal (unoptimized) loop.

The Testability Property

It is this property that makes runtime validation particularly difficult, otherwise we have a reasonably efficient way to check if the transformed loop satisfies our initial flow-dependence constraint, aka

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

If we relax this, we have an algorithm that detects these dependence violations



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)] $\geq k$ then exit with "failure"
end
exit with "success"
```

Correctness

1. 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 `MAK` 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, $P(j) < P(i)$.
2. If $j > i$, the rule is violated and the algorithm exits with failure.

Pros and Cons

- Pros:
  1. Similar Access Pattern: The loop index variable k accesses the loop indices in the transformed loop's manner.
  2. The computation of $D(k)$ and $U(k)$ is the same computation performed by the trans- formed loop.
- Cons:
  1. Memory Overhead: It requires an additional array `MARK` of the size of the actual array being accessed.
  2. **Testability Violated**: 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.
- To satisfy the testability property, we would need to answer "Is there a value $j \in \{P^{-1}(i+1), ..., P^{-1}(N)\}$ such that $j < P^{-1}(i)$ and $D(j) = U(P^{-1}(i))$?" in every ith iteration, without computing $D(j)$. For arbitrary function D and permutation P, this is not possible.

Restricting D

Consider the transformation:

```
for i = 1, N do                  for i = $P^{-1}(1), P^{-1}(N)$ do
a[i] = ...              $\Longrightarrow$          a[i] = ...
...= ...a[$U(i)$] ...                 ...= ...a[$U(i)$] ...
```

Here, we see that D is simple and *invertible*

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

where test$(i, r) = r < i \wedge P(r) > P(i)$

Expensive Tests?

- Since P is generated by the compiler itself, it can also generate efficient code for the test specific to P.
- **Loop Reversal**:
$$test(i, r) = r < i$$
- **Loop Interchange**:
$$test((i_1, i_2), (r_1, r_2)) = (r_1, r_2) < (i_1, i_2) \wedge (r_2, r_1) > (i_2, i_1)$$

  which reduces to
$$r_1 < i_1 \wedge r_2 > i_2$$

Dynamic Loop Interchange

```
for i = 1, N do                              for j = 1, M do
   for j = 1, M do               ⟹               k = 10 - j
      k = 10 - j                                 for i = 1, N do
      a[i, j] = a[i-1, j-k] + c                    a[i, j] = a[i-1, j-k] + c
```

We insert the runtime test as follows:

```
for j = 1, M do
     k = 10 - j
     for i = 1, N do
          if test((i, j),(i-1,j-k)) goto escape_code
          a[i, j] = a[i-1, j-k] + c
```

where

$$(r_1, r_2) = (i_1 - 1, j_1 - k)$$

so

$$test((i_1, i_2), (r_1, r_2)) = r_1 < i_1 \wedge r_2 > i_2$$

which in turn reduces to

$$test((i_1, i_2), (r_1, r_2)) = k < 0$$

Dynamic Loop Interchange: Recovery

```
for j = 1, M
    k = 10 - j
    if (k < 0) goto escape_code
    for i = 1, N
        a[i, j] = a[i-1, j-k] + c
```

On dependence violation, the program transfers control to escape code, which turns out to be surprisingly nice in this case:

```
for ii = 1, N do
    for jj = j, M do
        k = 10 - jj
        a[ii, jj] = a[ii-1, jj-k] + c
```

Dynamic Tiling

Recovery was fairly straightforward in the previous example, but it is suggested that in this example, we should just execute either the original or the transformed loop in all its completeness.

```
                                              for ii = 1, N by b do
for i = 1, N do                                   for jj = 1, N by b do
    for j = 1, N do              ⟹                    for i = ii to ii+b-1
    a[i, j] = a[i-k, j+10] + c                            for j = jj, jj+b-1 do
                                                              a[i, j] = a[i-k, j+10] + c
```

For this transformation to be valid, we must have $b \leq k$ (k is a variable whose value is unknown at compile time). So, we can jump to the original loop if we have $k < b$, alternatively, we can modify b to `b = min(k, maxTileSize)`.

Conclusion

Conclusion

- Due to the emergence of new classes of processors that can exploit ILP, and have hardware features that reduce the cost of runtime tests and the compensation code when a dependence violation is incurred, adding runtime tests to validate compiler optimizations is becoming increasingly tractable.
- In VLIW/EPIC machines, there are specific features, such as *dynamic disambiguation* (for optimised runtime aliasing checks), and *predication* (executing if and else branches in parallel, and only allowing the result to be written if the predicate is true) that aid in testing and compensating for dependence violations.
- In summary, we've looked at **Translation Validation** of structurally modifying optimizations, and for compilations whose validity is difficult to check at compile time, we've also looked at a run-time escape mechanism that completes the computation in a slower, but guaranteedly correct manner on encountering a dependence violation.