### Prelude - IA-64 Compiler Overview

**Profiling-Guided Opt**
- inlining, procedure layout
- instruction selection, speculation

**Interprocedural**
- memory disambiguation
- inlining, cloning partial inlining

**Data Locality and Parallelization**
- loop interchange, skewing, scaling fusion, unroll and jam, distribution
- scalar replacement, data prefetch
- OpenMP parallel, vectorization

**Global Scalar Optimizations**
- modified PRE, dead code, ...

**Scheduling**
- Predication, SW pipelining, ...

*From "An Overview of the Intel IA-64 Compiler" (1999) by Dulong et al.*

---

### Static Single Assignment Form

**Last Time**
- Induction variable detection and elimination and strength reduction

**Today**
- Program representations
- Static single assignment (SSA) form
  - Program representation for sparse data-flow
  - Conversion to and from SSA

**Next Time**
- Applications of SSA

---

### Data Dependence

**Definition**
- Data dependences are constraints on the order in which statements may be executed

We say statement \( s_j \) depends on \( s_i \)
- **Flow (true) dependence**: \( s_i \) writes memory that \( s_j \) later reads (RAW)
- **Anti-dependence**: \( s_i \) reads memory that \( s_j \) later writes (WAR)
- **Output dependences**: \( s_i \) writes memory that \( s_j \) later writes (WAW)
- **Input dependences**: \( s_i \) reads memory that \( s_j \) later reads (RAR)

**True dependences**
- Flow dependences represent actual flow of data

**False dependences**
- Anti- and output dependences reflect reuse of memory, not actual data flow; can often be eliminated

---

### Example

\[
\begin{align*}
  s_1 & : a = b; \\
  s_2 & : b = c + d; \\
  s_3 & : e = a + d; \\
  s_4 & : b = 3; \\
  s_5 & : f = b * 2;
\end{align*}
\]
Representing Data Dependences

Implicitly
- Using variable defs and uses
- Pros: simple
- Cons: hides data dependence (analyses must find this info)

Def-use chains (du chains)
- Link each def to its uses
- Pros: explicit; therefore fast
- Cons: must be computed and updated, space consuming

Alternate representations
- e.g., Static single assignment form (SSA), Program Dependence Graph (PDG), dependence flow graphs (DFG), value dependence graphs (VDG),

DU Chains

Definition
- du chains link each def to its uses

Example

```
s_1     a = b;
s_2     b = c + d;
s_3     e = a + d;
s_4     b = 3;
s_5     f = b * 2;
```

UD Chains

Definition
- ud chains link each use to its defs

Example

```
s_1     a = b;
s_2     b = c + d;
s_3     e = a + d;
s_4     b = 3;
s_5     f = b * 2;
```

Role of Alternate Program Representations

Advantage
- Allow analyses and transformations to be simpler & more efficient/effective

Disadvantage
- May not be “executable” (requires extra translations to and from)
- May be expensive (in terms of time or space)

Process

```
Original Code (RTL) ➔ SSA Code1 ➔ SSA Code2 ➔ SSA Code3
Optimized Code (RTL)
```

T1 ➔ T2
Static Single Assignment (SSA) Form

Idea
– Each variable has only one static definition
– Makes it easier to reason about values instead of variables
– Similar to the notion of functional programming

Transformation to SSA
– Rename each definition
– Rename all uses reached by that assignment

Example
\[
\begin{align*}
  v & := \ldots \\
  \ldots & := \ldots v \ldots \\
  v & := \ldots \\
  \ldots & := \ldots v \ldots \\
  \end{align*}
\]

What do we do when there’s control flow?

SSA and Control Flow

Problem
– A use may be reached by several definitions

Merging Definitions
– \( \phi \)-functions merge multiple reaching definitions

Example
\[
\begin{align*}
  v & := \ldots \\
  v_0 & := \ldots \\
  v & := \ldots \\
  v & := \ldots \\
\end{align*}
\]

Another Example

Example
\[
\begin{align*}
  v & := 1 \\
  v_0 & := 1 \\
  v & := v+1 \\
  v_1 & := \phi(v_0, v_2) \\
  v_2 & := v_1 + 1 \\
\end{align*}
\]
SSA vs. ud/du Chains

SSA form is more constrained

Advantages of SSA
– More compact
– Some analyses become simpler when each use has only one def'
– Value merging is explicit
– Easier to update and manipulate?

Furthermore
– Eliminates false dependences (simplifying context)

for (i=0; i<n; i++)
    A[i] = i;

for (i=0; i<n; i++)
    print(foo(i));

SSA vs. ud/du Chains (cont)

Worst case du-chains?

switch (c1) {
    case 1:     x = 1; break;
    case 2:     x = 2; break;
    case 3:     x = 3; break;
}

switch (c2) {
    case 1:     y1 = x; break;
    case 2:     y2 = x; break;
    case 3:     y3 = x; break;
    case 4:     y4 = x; break;
}

m defs and n uses leads to m×n du chains

Transformation to SSA Form

Two steps
– Insert $\phi$-functions
– Rename variables

Where Do We Place $\phi$-Functions?

Basic Rule
– If two distinct (non-null) paths $x\rightarrow z$ and $y\rightarrow z$ converge at node $z$, and
  nodes $x$ and $y$ contain definitions of variable $v$, then a
  $\phi$-function for $v$ is inserted at $z$
Approaches to Placing $\phi$-Functions

**Minimal**
- As few as possible subject to the basic rule

**Briggs-Minimal**
- Same as minimal, except $v$ must be live across some edge of the CFG

**Pruned**
- Same as minimal, except dead $\phi$-functions are not inserted

What’s the difference between Briggs Minimal and Pruned SSA?

Briggs Minimal vs. Pruned

Briggs Minimal will add a $\phi$ function because $v$ is live across the blue edge, but Pruned SSA will not because the $\phi$ function is dead.

Neither Briggs Minimal nor Pruned SSA will place a $\phi$ function in this case because $v$ is not live across any CFG edge.

Why would we ever use Briggs Minimal instead of Pruned SSA?

Machinery for Placing $\phi$-Functions

Recall Dominators
- $d\text{ dom }i$ if all paths from entry to node $i$ include $d$
- $d\text{ sdom }i$ if $d\text{ dom }i$ and $d\neq i$

Dominance Frontiers
- The **dominance frontier** of a node $d$ is the set of nodes that are “just barely” not dominated by $d$; i.e., the set of nodes $n$, such that
  - $d$ dominates a predecessor $p$ of $n$, and
  - $d$ does not strictly dominate $n$
- $DF(d) = \{n | \exists p \in \text{pred}(n), d\text{ dom }p \text{ and } d\not\text{ sdom }n\}$

Notational Convenience
- $DF(S) = \bigcup_{s \in S} DF(s)$

Dominance Frontier Example

Nodes in Dom(5)

What’s significant about the Dominance Frontier?

In SSA form, definitions must dominate uses
**Dominance Frontier Example II**

\[ DF(d) = \{ n \mid \exists p \in \text{pred}(n), d \text{ dom } p \text{ and } d \not\text{sdom } n \} \]

\[ \text{Dom}(5) = \{ 5, 6, 7, 8 \} \]

\[ DF(5) = \{ 4, 5, 13 \} \]

In this graph, node 4 is the first point of convergence between the entry and node 5, so do we need a \( \phi \)-function at node 13?

**Dominance Frontiers Revisited**

Suppose that node 3 defines variable \( x \)

\[ DF(3) = \{ 5 \} \]

Do we need to insert a \( \phi \)-function for \( x \) anywhere else?

Yes. At node 6. Why?

**SRA Exercise**

\[ DF(8) = \{ 10 \} \]

\[ DF(9) = \{ 10 \} \]

\[ DF(2) = \{ 6 \} \]

\[ DF(\{ 8, 9 \}) = \{ 10 \} \]

\[ DF(10) = \{ 6 \} \]

\[ DF(\{ 2, 8, 9, 6, 10 \}) = \{ 6, 10 \} \]


**Dominance Frontiers and SSA**

Let

\[ DF_1(S) = DF(S) \]

\[ DF_{i+1}(S) = DF(S \cup DF_i(S)) \]

**Iterated Dominance Frontier**

\[ DF_n(S) \]

**Theorem**

\[ \text{If } S \text{ is the set of CFG nodes that define variable } v, \text{ then } DF_n(S) \text{ is the set of nodes that require } \phi \text{-functions for } v \]
Algorithm for Inserting $\phi$-Functions

for each variable $v$
  WorkList $\leftarrow \emptyset$
  EverOnWorkList $\leftarrow \emptyset$
  AlreadyHasPhiFunc $\leftarrow \emptyset$
for each node $n$ containing an assignment to $v$
  WorkList $\leftarrow$ WorkList $\cup$ $\{n\}$
  EverOnWorkList $\leftarrow$ WorkList
while WorkList $\neq \emptyset$
  Remove some node $n$ for WorkList
  for each $d \in \text{DF}(n)$
    if $d \notin$ AlreadyHasPhiFunc
      Insert a $\phi$-function for $v$ at $d$
      AlreadyHasPhiFunc $\leftarrow$ AlreadyHasPhiFunc $\cup$ $\{d\}$
    if $d \notin$ EverOnWorkList
      WorkList $\leftarrow$ WorkList $\cup$ $\{d\}$
      EverOnWorkList $\leftarrow$ EverOnWorkList $\cup$ $\{d\}$

Put all defs of $v$ on the worklist
Put all $v$ on the worklist

Concepts

Data dependences
- Three kinds of data dependences
- du-chains
Alternate representations
SSA form
Conversion to SSA form
- $\phi$-function placement

Variable Renaming

Basic idea
- When we see a variable on the LHS, create a new name for it
- When we see a variable on the RHS, use appropriate subscript

Easy for straightline code

Use a stack when there’s control flow
- For each use of $x$, find the definition of $x$ that dominates it

Traverse the dominance tree
The dominance tree shows the dominance relation.

```
1
2 3 4 5 6 7 8 9 10 11 12 13
```

### Variable Renaming Algorithm

```plaintext
procedure Rename(block b)
    if b previously visited return
    GenName(LHS(p)) and replace v with v_i, where i=Top(Stack[v])
    for each statement s in b (in order)
        for each variable v ∈ RHS(s)
            replace v by v_i, where i=Top(Stack[v])
    for each variable v ∈ LHS(s)
        GenName(v) and replace v with v_i, where i=Top(Stack[v])
    for each v_i ∈ LHS(t)
        Pop(Stack[v])
    for each φ-function or statement t in b
        Rename(s)
```

### Transformation from SSA Form

**Proposal**

- Restore original variable names (i.e., drop subscripts)
- Delete all φ-functions

**Complications**

- What if versions get out of order?
  (simultaneously live ranges)

**Desired Behavior**

- Perform dead code elimination (to prune φ-functions)
- Replace φ-functions with copies in predecessors
- Rely on register allocation coalescing to remove unnecessary copies