Exploring the Landscape of SSA-based Program Representations

Philip Brisk
École Polytechnique Fédérale de Lausanne (EPFL)

SSA Seminar, Autrans, France
April 27, 2009
What is a Mathematician?

• A machine that converts coffee into theorems
• Beer-related variations exist as well
• More specifically
  – Defines something
  – Derives properties thereof via theorems and proofs
  – With luck, a useful application is found eventually
  – With a lot of luck, the application is found before said mathematician dies
Disclaimers

• Some of this talk has not been peer reviewed or published

• Conjectures abound
I Assume…

• You know what is SSA Form, and that you care

• This talk is not about SSA-based register allocation…
  – Except for the fact that, somehow, it is…

• “SSA Form” implies “Pruned SSA Form”
  – See above
  – DCE converts minimal or semi-pruned to pruned

• “SSA Form” implies Strict SSA
  – See the next slide…
Strict vs. Non-strict SSA Form

- Each definition dominates each use
- Arguably, we can eliminate this $\varphi$-function

- Fewer $\varphi$-functions
- Lose properties involving dominance, chordal interference graphs, etc.
Key Points of SSA Form

• Each definition dominates each use…
  – And each point where each variable is live

• Each variable live range is a subtree of the dominator tree

• A chordal graph is the intersection graph of a set of subtrees of a tree
  – $O(|V| + |E|)$-time algorithms for
    • coloring, clique, independent set, clique partition

• Should we do register allocation in SSA Form?
SSA Form is Plural

• You are probably familiar with φ-functions…
  – And Cytron et al.’s SSA construction method…
  – And maybe a few other equivalent construction methods too…

• φ-functions are just a way to split variable live ranges at convergence points in the control flow graph
  – With very specific parallel copy semantics

• Why stop there?
Other Ways to Split

\[ X_1 \leftarrow \ldots \quad X_2 \leftarrow \ldots \]
\[ X_3 \leftarrow \varphi(X_1, X_2) \]

\[ X_2, X_3 \leftarrow \sigma(X_1) \]
\[ \ldots \leftarrow X_2 \]
\[ \ldots \leftarrow X_3 \]

Instr-1
\[(Y_1, \ldots, Y_n) \leftarrow \text{Parallel Copy}(X_1, \ldots, X_n)\]
Instr-2
Elementary Form

Split each variable at every place where it is live
  - $\phi$-functions for all variables live at a merge point
  - $\sigma$-functions for all variables live at split points
  - Parallel copies for each variable live between two instructions

Elementary graphs
  - Each connected component is the interference graph of one instruction
    - Technically, a clique substitution of $P_3$
  - A subclass of chordal graphs
    - Stronger theoretical properties than chordal graphs
  - Details will be provided in Jens Palsberg’s talk on puzzle solving
The Interference Graph for One Instruction

- Two intersecting cliques

  - The variables whose lifetimes die at the instruction
  - The variables defined by the current instruction that become live

Variables live across the instruction
Conjecture Time...

- Let CHO be the class of chordal graphs
- Let ELEM be the class of elementary graphs
- Obviously, ELEM ⊂ CHO
Problem 1

• Let $X$ be a class of graphs such that
  – $\text{ELEM} \subseteq X \subseteq \text{CHO}$

• I want an SSA-based representation whose interference graph belongs to class $X$, because $X$ has some favorable property

• **Answer**: Build Elementary Form
  – You get the property you want, and more…
  – If you care about the number of $\phi$-functions, $\sigma$-functions, and parallel copies, reformulate the problem
Problem 2

• Let X, Y be classes of graphs such that
  – ELEM ⊆ X ⊆ Y ⊆ CHO

• I want an SSA-based representation whose interference graph belongs to class Y, because Y has some favorable property

• I do not want to build an SSA-based representation whose interference graph belongs to class X, because doing so will introduce more φ-function, σ-functions, and parallel copies than I want to deal with

• Conjecture(s):
  – An “efficient” algorithm exists to do this
  – The algorithm is sufficiently general for any pair of classes X and Y as defined above.
Problem 3

• Let $X$ be a class of graphs such that
  – $\text{ELEM} \subseteq X \subseteq \text{CHO}$

• I want an SSA-based representation whose interference graph belongs to class $X$, because $X$ has some favorable property

• I want to ensure this algorithm inserts the minimal number of $\varphi$-functions, $\sigma$-functions, and parallel copies
  – i.e., I won’t settle for Elementary Form

• **Conjecture:**
  – An “efficient” algorithm exists to do this
More Rambling…

• Elementary graphs appear to be a hard lower bound
  – Given the instruction sets of today’s processors

• No rule says that the upper bound in the preceding problem statements must be chordal graphs
  – Weakly chordal graphs
  – Perfect graphs
  – Any graph?

• Going beyond chordal graphs may require us to relax the notion of “efficient algorithm”
  – Last I heard, perfect graph recognition takes $O(|V|^9)$ time.
Limitations

- There are some classes of graphs that cannot be characterized as the interference graph of a program in any realistic SSA-based representation that we know of.

- Example: Split graphs
  - A chordal graph whose vertices can be partitioned into an independent set and a clique
  - Or, equivalently, the class of chordal graphs whose complements are chordal
  - Easy to find interference graphs for one instruction that are not Split graphs
Spilling Does Not Preserve the Cytron et al. SSA Form You Know

\[ X_1 \leftarrow \ldots \]
\[ \ldots \leftarrow X_1 \]
\[ \ldots \leftarrow X_1 \]
\[ \ldots \leftarrow X_1 \]

\[ X_1 \leftarrow \ldots \]
\[ \ldots \leftarrow X_1 \]
\[ M \leftarrow \text{str}[X_1] \]
\[ X_2 \leftarrow \text{ld}[M] \]
\[ \ldots \leftarrow X_2 \]

\[ X_1 \leftarrow \ldots \]
\[ \ldots \leftarrow X_1 \]
\[ M \leftarrow \text{str}[X_1] \]
\[ X_2 \leftarrow \text{ld}[M] \]
\[ \ldots \leftarrow X_2 \]
\[ X_3 \leftarrow \varphi(X_1, X_2) \]
\[ \ldots \leftarrow X_3 \]
Key Points of SSA-based Spilling

• For simplicity, I ignore the finer details of spill code placement

• The issue of rebuilding SSA Form does not arrive under a spill-everywhere model
  – So, assume we don’t spill everywhere
  – Good idea, as this reduces the amount of spill code

• Every use of a variable in SSA Form is the placeholder for a potential new definition, after spilling
  – The load placed before the use is the new definition
Cytron et al.’s SSA Construction Algorithm

- $D_v$ – the set of basic blocks containing definitions of $v$

- IDF(...) – the iterated dominance frontier of a set of basic blocks

- Place $\phi$-functions for $v$ at the entry of every basic block in IDF($D_v$)
  - Yields minimal form
  - Filtering yields semi-pruned form
    - See [Briggs et al., SPE 1998]
  - Dead code elimination converts to pruned form
    - Folklore, but easy to prove
SSRO: Yet-another SSA Variant (Acronym to be Explained Later)

- $D_v$ – the set of basic block containing definitions of $v$
- $D_{Uv}$ – the set of basic blocks containing definitions or uses of $v$

- IDF(...) – the iterated dominance frontier of a set of basic blocks

- Place $\phi$-functions for $v$ at the entry of every basic block in IDF($D_{Uv}$)
  - In contrast, IDF($D_v$) for SSA Form
  - Minimal, semi-pruned, pruned variants exist
SSA on the Left / SSRO on the Right
SSA on the Left / SSRO on the Right
Definitions

- **Occurrence** – a definition or use of a variable

- An occurrence $O_1$ of variable $v$ **reaches** a second occurrence $O_2$ of $v$ if there is a path in the CFG from $O_1$ to $O_2$ that does not pass through any other occurrence of $v$.

- $\text{ReachOcc}_v[O_i]$ – the set of reaching occurrences of $v$ that reach $O_i$
Reaching Definitions

ReachOcc\(_x\)[B] = \{A\}
ReachOcc\(_x\)[C] = \{B\}
ReachOcc\(_x\)[D] = \{A\}
ReachOcc\(_x\)[E] = \{A, E\}
ReachOcc\(_x\)[F] = \{D, E\}

ReachOcc\(_{x1}\)[B] = \{A\}
ReachOcc\(_{x1}\)[C] = \{B\}
ReachOcc\(_{x1}\)[D] = \{A\}
ReachOcc\(_{x1}\)[E] = \{D\}
ReachOcc\(_{x1}\)[F] = \{A\}

ReachOcc\(_{x2}\)[H] = \{G\}
ReachOcc\(_{x2}\)[I] = \{H\}
ReachOcc\(_{x2}\)[J] = \{H\}
ReachOcc\(_{x3}\)[L] = \{K\}
The Def-Use Tree

• In SSA Form, the definition of each variable dominates all of its uses

• Organize definitions and uses as a tree
  – idom \( O_i \) – immediate dominating occurrence of use \( O_i \)
  – i.e., the parent of \( O_i \) in the DU-tree
ReachOcc_x[B] = \{A\}
ReachOcc_x[C] = \{B\}
ReachOcc_x[D] = \{A\}
ReachOcc_x[E] = \{A, E\}
ReachOcc_x[F] = \{D, E\}
Leaves and Death Points

Reach\text{Occ}_{x_1}[B] = \{A\}
Reach\text{Occ}_{x_1}[C] = \{B\}
Reach\text{Occ}_{x_1}[D] = \{A\}
Reach\text{Occ}_{x_1}[E] = \{D\}
Reach\text{Occ}_{x_1}[F] = \{A\}
Reach\text{Occ}_{x_2}[H] = \{G\}
Reach\text{Occ}_{x_2}[I] = \{H\}
Reach\text{Occ}_{x_2}[J] = \{H\}
Reach\text{Occ}_{x_3}[L] = \{K\}
The Static Single Reaching Occurrence (SSRO) Property

- **Theorem**: In SSRO Form, the set of reaching occurrences for each use is a singleton
  - Specifically, $\text{ReachOcc}_x[O_i] = \{\text{idom } O_i\}$

- **Theorem**: In SSRO Form, the death point of each variable corresponds to a leaf in the def-use tree
  - If not, there is a path from $O_i$ to itself, so $|\text{ReachOcc}_x[O_i]| > 1$
  - Contradicts the theorem above
Spilling Under SSRO Form

\[ \text{Rename uses of } v \text{ to } v' \]
Spilling Under SSRO Form

• **Theorem**: There is no need to insert any additional φ-functions if spilling is applied under SSRO Form.
  – For each use of v, \( \text{ReachOcc}_v[O_i] = \{\text{idom } O_i\} \)
  – Any path from occurrence \( O_j \) to use \( O_i \) must pass through \( \text{idom } O_i \)

• Practical Issues
  – Simplifies process of SSA-based register allocation
  – Additional φ-functions suggest...
    • Live range splitting on a finer granularity than SSA Form
    • Probably better for spilling, but worse for coalescing
Summary:
Key Properties of SSRO Form

• The set of reaching occurrences of each use is a singleton

• Each death point of a variable corresponds to a use
  – Organize the definition and uses of each variable into a tree
  – Each death point is a leaf, and vice-versa

• No additional $\varphi$-functions must be inserted after spilling
  – i.e., a procedure in SSRO Form remains in SSRO Form after spilling

• Like SSA Form, the interference graph is chordal
  – i.e., given a chordal graph, I can construct an SSRO Form procedure
    whose interference graph is the same as the given graph.
Going Interprocedural

- It is possible to build a whole program representation such that the interprocedural interference graph is chordal
  - Only works if I can resolve all function pointers in advance
  - Paper published at ICCAD 2007

- Extensions are necessary to extend the result to Elementary Form
  - I have worked them out in my head
  - Call it a conjecture for now
Recursive Calls

• How to handle variables live across calls in a recursive chain?
  – Pushed onto stack
  – Cannot use registers

• Call graph becomes a DAG
  – Strongly connected components – $O(|V| + |E|)$
  – Collapse each SCC into a single node
Local and Global Interference

- Local Interferences
  - Variables in the same procedure
  - Overlapping lifetimes

- Global Interferences
  - Variables live across procedure calls
  - Interferences are transitive

```
X ← ...  --------  X
      
Y ← ...  --------  Y
      
... ← X
      
Z ← ...  --------  Z
      
... ← Y
      
... ← Z
```

```
Main:
V ← ...

Call P

P:
...

Call Q

...

V

Main → P → Q
```
Launch and Landing Pads

• When $P_i$ is called
  - The maximum stack size is $m = \delta_i$
    • Taken among all paths in the call graph leading to $P_i$
  - Global registers $T_1 \ldots T_m$ store variables live across calls in the chain

• $P_i$ calls $P_j$ at call point $c_k$
  - $L(c_k)$ – set of variables live across the call
  - Let $n = |L(c_k)|$ be the number of variables

• Launch and Landing Pads
  - Parallel copy $(T_{m+1} \ldots T_{m+n}) \leftarrow \psi(L(c_k))$ inserted before the call
  - Parallel copy $L(c_k) \leftarrow \psi^{-1}(T_{m+1} \ldots T_{m+n})$ inserted after the call
The Interprocedural Interference Graph is Chordal

CLIQUE

N = 6

G1
δ1 = 0

G2
δ2 = 2

G3
δ3 = 3

G4
δ4 = 2

G5
δ5 = 6

G6
δ6 = 5

Tj interferes with each local variable in Gi

Gi is chordal
Conclusions

• If you think in terms of classes of interference graphs, there are a wide variety of SSA-based representations that have yet to be explored
  – Not clear if they are useful for register allocation
  – Not clear if they provide superior facilities for dataflow analysis

• SSRO Form is somehow orthogonal to the above
  – I invented it when thinking about spilling under SSA
  – Eliminates the need to insert additional φ-functions after spilling

• Interprocedural extensions
  – Only if we can resolve function pointers