Beyond Scalar SSA: Compilers for manycore processors Need Dynamic SA and some form of Stream SSA

Albert Cohen and many other (in)direct contributors

ALCHEMY group
INRIA Saclay and LRI, Paris-Sud 11 University, Orsay, France

April 28, 2009
Position of the Problem

Claim 1: Lost Portability

- Compilers (and runtime systems) have lost a round, and we cannot afford to concede the game
  - Fundamental point: we still don’t really know how to optimize (parallel) programs for non-uniform memory hierarchies, assuming we reasonably understand scalar optimization
  - Applied point: software developers are in dire need for an answer

Claim 2: Our Research Area is Hot

- The problem will not be solved by advances in compiler construction alone, but the compiler side of the story is the most interesting challenge for the manycore era

Goal

Regaining the lost performance portability
 Scalar Data Flow

Motivation

\[ x_0 = 0; \]

\[ \text{while (1) {} } \]
\[ \begin{align*}
  x_1 &= \Phi(x_0, x_2); \\
  x_2 &= f(x_1); &\text{// Sequential} \\
  g(x_2); &\text{// May pipeline f() and g() if } x_2 \text{ is privatized}
\end{align*} \]

- Trivial to extract plenty of data and pipeline parallelism
- But what about the effective exploitation of this parallelism?
Array Data Flow

Coarsening Synchronization/Computation Ratio

\[ x_0 = 0; \]
\[ \text{while (1) { } for (i=0; i<n; i++) { } } \]
\[ x_1 = \Phi(x_0, x_2); \]
\[ x_2 = f(x_1); \]
\[ a[i] = x_2; \]
\[ } \]
\[ \text{for (i=0; i<n; i++) } // \text{Should align concurrent iterations of } f() \text{ and } g() \text{ to exploit locality } \]
\[ g(a[i]); \]

- This is not sufficient
- \( x_2 \) is fundamentally a well-behaved (single-assignment) stream of data, not a random access array with nasty side-effects, and a circular window of size \( n \) even less
Array Data Flow

Synchronization at Merge Point

```c
x0 = 0;
while (1) {
    for (i=0; i<n; i++) {
        x1 = Φ(x0, x2);
        x2 = f(x1);
        a[i] = x2;
    }
    x3 = Φ(x0, x2); // Needed in general if x is live beyond its use in g()
}
for (i=0; i<n; i++)
    g[a[i]];
```

- In fact, this is really bad...
- Critical issue: sequentialization induced by a scalar Cond-Φ node
Array Data Flow

Synchronization at Merge Point

\[ x_0 = 0; \]
\[ \text{while (1) { } } \]
\[ \text{for (i=0; i<n; i+) { } } \]
\[ x_1 = \Phi(x_0, x_2); \quad \text{// Loop-\Phi node: ‘‘pre’’ operator in the data-flow synchronous language Lustre } \]
\[ x_2 = f(x_1); \]
\[ a[i] = x_2; \]
\]
\[ x_3 = \Phi(x_0, x_2); \quad \text{// Cond-\Phi node: ‘‘mux’’ operator of logic circuits } \]
\[ \text{for (i=0; i<n; i++) } \]
\[ g(a[i]); \]
\]

- In fact, this is really bad...
- Critical issue: sequentialization induced by a scalar Cond-\Phi node
- Need to distinguish between “pre” and “mux” semantics
- An instance of a not-so-well-understood aliasing pitfall in the history of data-flow computing and parallel functional languages
Does Polyhedral Compilation Help?

Dynamic Single Assignment

x = 0;
// Peeled one iteration of the global loop
a[0] = f(x);
for (i=1; i<n; i++)
  a[i] = f(a[i-1]);
for (i=0; i<n; i++)
g(a[i]);

while (1) {
  a[0] = f(a[n-1]);
  for (i=1; i<n; i++)
    a[i] = f(a[i-1]);
  for (i=0; i<n; i++)
    g(a[i]);
}

- Feautrier's Array Dataflow Analysis and Array Expansion (ICS’88)
  - Static control programs, reaching production with IBM XL (in progress) and GCC 4.4
- Beyond static control: Collard, Griebl, Wonnacott, Barthou, Cohen et al. 94–99
  - E.g., Maximal Static Expansion (POPL’98), no runtime data-flow recollection overhead
  - New results in polyhedral code generation and affine transformation for arbitrary control flow (intraproc.), but still many complexity issues, submitted for publication
Towards Stream SSA

\[ x_0 = 0; \]
\[ \text{while (1) {} } \]
\[ \quad \text{for (i=0; i<n; i++) {{} } } \]
\[ \quad \quad x_1 = \Phi(x_0, x_2); \quad \text{// Identical to ‘pre’ in Lustre} \]
\[ \quad \quad x_2 = f(x_1); \quad \text{// Iterative definition of stream x} \]
\[ \quad } \]
\[ x_3 = \Phi(x_0, x_2); \quad \text{// Pointwise extension of Cond-\Phi to streams} \]
\[ \text{for (i=0; i<n; i++)} \]
\[ \quad g(x_3); \quad \text{// Iterative use of stream x} \]

- Aim for a denotational definition: e.g., Pop’s formalism (and distinction between loop- and merge- \( \Phi \) nodes)
  - Leverage Kahn semantics: continuous functions over the prefix ordering of streams
  - Leverage synchronous clocks to establish the pointwise mapping from definitions to uses of streams, and to generate efficient sequential code from the concurrent streaming representation: see Lustre and extensions in Lucid Synchrone, \( n \)-synchronous clocks at POPL’06, etc.
Data-Flow Computing on Streams

Optimizations on Stream SSA

\[ x_0 = 0; \]
// Anticipate computation of \( f() \) for latency-hiding

\[ \text{for (i=0; i<n; i++)} \{
  x_1 = \Phi(x_0, x_2);
  x_2 = f(x_1); // Sequential execution
\}
\]

\[ x_3 = \Phi(x_0, x_2); \]

\[ \text{while (1)} \{ // May require extra ‘‘task’’ decoration to make parallelism explicit
  \text{for (i=0; i<n; i++)} \{
    x_4 = \Phi(x_3, x_5);
    x_5 = f(x_4); // Sequential execution
  \}
\]

\[ x_6 = \Phi(x_3, x_5); \]

\[ \text{while (1)} // May require extra ‘‘task’’ decoration to make parallelism explicit
  \text{for (i=0; i<2*n; i++)} // Further coarsening for load-balancing purposes
    g(x_6); // Could be executed in parallel
\]

- Express aggressive transformations on data- and pipeline-parallel programs
- Serious liveness/boundedness challenges: much to learn from synchronous languages, with the huge advantage that the original code is causal and has bounded memory!
### Research Directions

**Conjecture 1**
Stream SSA subsumes SSA for all classical analysis and optimization purposes.

**Conjecture 2**
Stream SSA enables seamless extension of classical optimizations to concurrent programs.
(forget about interleaving and memory models... for a moment at least, it strikes back at a lower level)

**Conjecture 3**
Stream SSA is good enough for common parallelizing compilation purposes.
(good = expressive, robust to transformations and complexity-effective)
Work Program

- Define Stream SSA (and name it properly)
- Revisit analysis and optimization problems on Stream SSA
- Glue it with polyhedral compilation as seamlessly as possible (graceful degradation of accuracy and aggressiveness)

- Implement in GCC (see related projects on OpenMP + streams, Graphite for polyhedral compilation, and transactional memory support)
Thank You