Induction Variable Analysis with Delayed Abstractions

Sebastian Pop   Albert Cohen   Georges-André Silber

CRI, Mines Paris, Fontainebleau, France
ALCHEMY, INRIA Futurs, Orsay, France

November 8, 2005
HiPEAC 2005

What are the problems?

Is this Loop Parallel? Can we remove the condition?

```
k = 4;
for (i = 7; i < 10; i++) {
 if (0 < k && k < 10)
  A[k++] = A[k-3] + 1;
}
```

Need several informations

- data dependences
- number of iterations
- induction variables (IV)

Analysis of induction variables is central to loop optimizations

- constant/range propagation (check elimination)
- IV selection
- strength reduction
- vectorization
- parallelization
- loop nest transformations

State of the Art for IV Analysis

- on Static Single Assignment (SSA) (Wolfe 1992)
- interpret first iterations (Haghighat Polychronopoulos 1992)
- in production compiler MIPSPro (Liu Lo Chow 1996)
- monotone evolutions (Wu Cohen Padua 2001)
- chains of recurrences (van Engelen 2001)
- hybrid static + dynamic (Rus Rauchwerger 2002)

Example: SSA Representation

```
k = 4;
for (i = 7; i < 10; i++) {
  if (0 < k && k < 10)
    A[k++] = A[k-3] + 1;
}
```

SSA representation →

```
a = 7
b = 4
 next:
 c = phi (a, g) // "i"
 d = phi (b, d, f) // "k"
 if (c >= 10) goto end
 if (d <= 0) goto next
 if (d >= 10) goto next
 e = d - 3
 A[d] = A[e]
 f = d + 1
 g = c + 1
 goto next
end:
```

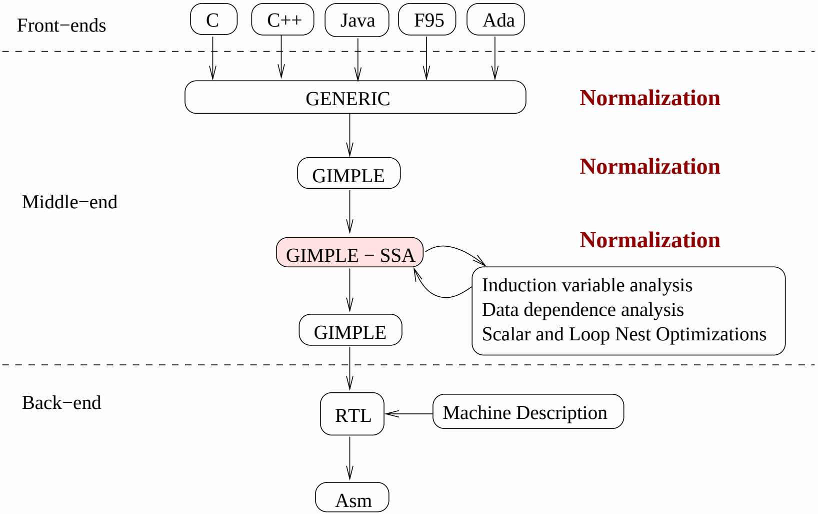
SSA (Static Single Assignment) links scalar uses to defs.

Open Problems on IV Analysis

Induction Variable (IV) Analysis

- on low level representation (for modern LNO: McCAT, LLVM)
- code scrambled by previous optimizations
- typed scalar variables with overflow
- low complexity for production compilers
- provide the right abstraction level

Why on low level? The case of GCC



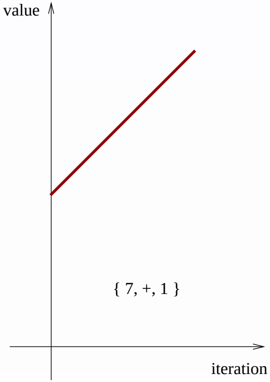
Our Solution for IV Analysis

Linear unification + delayed abstraction selection

- on low level SSA (three address code, loops as “if + gotos”)
- avoid syntactic matching (reduces complexity of matching)
- pattern matching on the SSA graph
- representation: extension of chains of recurrences
- complexity: linear in number of SSA scalar variables
- fits the needs of several optimizations in GCC

```
next:
  c = phi (7, g)
  d = phi (4, d, f)
  if (c >= 10) goto end
  if (d <= 0) goto next
  if (d >= 10) goto next
  e = d - 3
  A[d] = A[e]
  f = d + 1
  g = c + 1
  goto next
end:
```

c starts at 7 with step 1

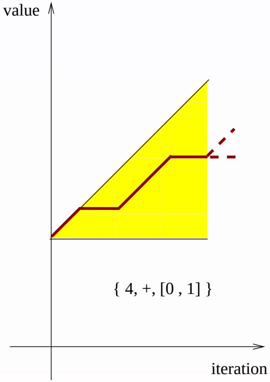


Detection of self defined variables

Example: evolution envelope

```
next:
   c = phi (7, g)
   d = phi (4, d, f)
   if (c >= 10) goto end
   if (d <= 0) goto next
   if (d >= 10) goto next
   e = d - 3
   A[d] = A[e]
   f = d + 1
   g = c + 1
   goto next
end:
```

d starts at 4 with step 0 or 1



Detection of self defined variables

Evolution envelopes

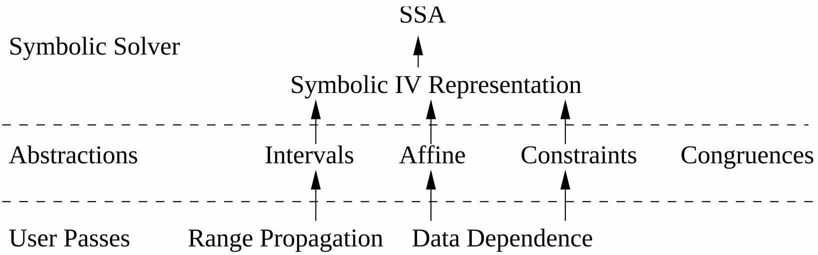
Extension of chains of recurrences to handle

- symbolic expressions: trees of recurrences TREC $a = \{1, +, a\}$
- scalar envelopes (sign, intervals, polyhedra) $b = \{0, +, [1, 3]\}$
- scalar types and overflow effects $(\mathit{unsigned\ char})\{0, +, 1\}$
- wrap-around and periodic evolutions $c = (1, 2, 3, c)$

Algorithm extracting TREC

Lazy resolution of symbols

- similar to linear unification (Patterson Wegman 1976)
- partially solve recurrences (self defines)
- rewriting = collapsing of cycles
- compute symbolic of trip counts and inner loop effects
- leave as many symbols as possible (lazy = precise)



Why delaying abstractions?

Complex (partially solved) recurrences can be simplified (reduced to closed form) by characterization of other variables.

Optimizers need different abstractions

- IV opts: affine evolutions (constant base constant step)
- vectorizer and pointer dependence analysis:
  - symbolic initial values (base pointer)
  - affine evolutions
- value range propagation: estimation of #iters, intervals

Rule: keep precise symbolic representations as long as possible.
Instantiate symbols only when impossible to do otherwise.
User passes instantiate symbols to fit their needs.

Outline

- introduction and motivation
- representations and delayed instantiations
- types and overflows
- application to data dependence analysis
- experiments
- future work

Scalar Types and Overflow Effects

Is "c" affine?

```
int i;
unsigned char c = 0;
for (i = 0; i < 1000; i++)
 c++;
```

No. "c" is periodic

$c = \{0, 1, \ldots, 255, 0, 1, \ldots\}$

Cast to types require (estimations of) number of iterations.

Data Dependence Analysis on Delayed Abstractions

classic data dependence analyzers

- Banerjee test (dependence vectors + dependence domains)
- Omega test (solve a system of constraints)

Bounding the iterations domains:

- exact #iter (first iteration satisfying exit conditions)
- estimations from undefined behavior
  - array accesses should be in statically allocated area
    (on SPEC2000.swim bound on #iter from size of static data)
  - overflowing of signed IV

Example

```
if (x) N = 0;
else N = 10;
for (i = 4; i < 8; i++) {
 int k = i + N;
 A[k] = A[k - 4] + 1;
}
}
```

- instantiating k gives [4, 17]
  $[4, 17] \cap [0, 13] = [4, 13]$: failed to prove independence
- delay instantiation to data dependence analysis time
  $[4 + N, 7 + N] \cap [N, 3 + N] = \emptyset$ proved independent

Experiments

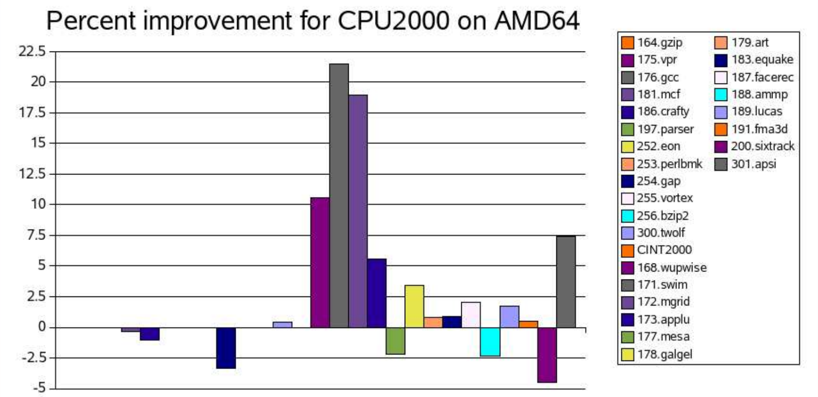
Base and peak compilers:

- GCC version 4.1 as of 2005-Nov-04
- options: "-O3 -msse2 -ftree-vectorize -ftree-loop-linear"
- base: our analyzer is disabled
- peak: GCC with no modifications

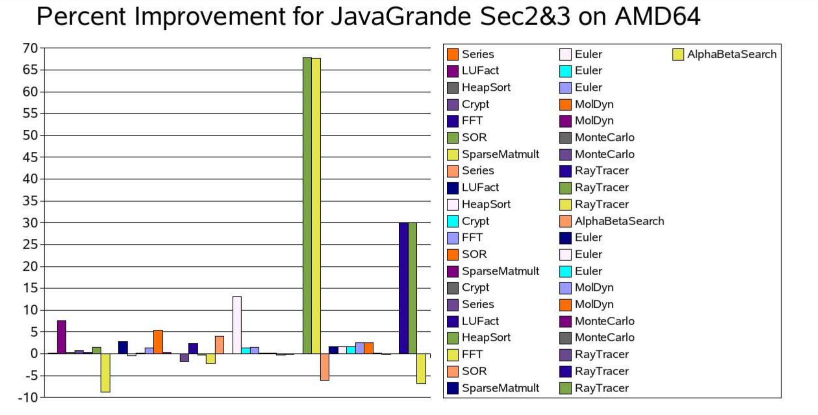
Benchmarks:

- CPU2000 and JavaGrande on AMD64 3700 Linux 2.6.13
- MiBench on ARM XScale-IXP42x 133 Linux 2.6.12

Experiments

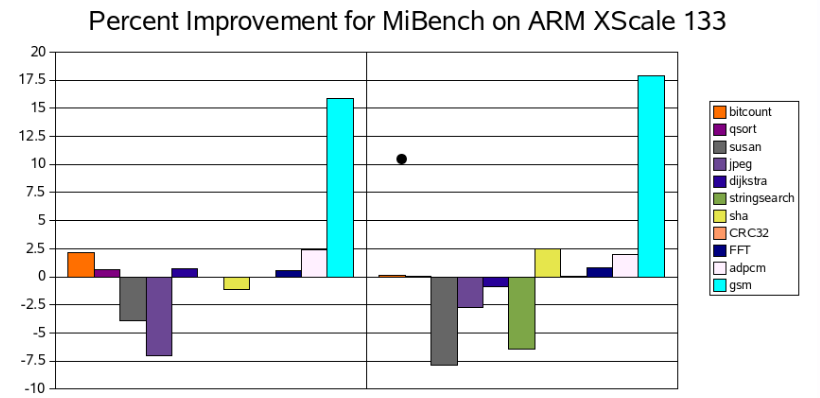


Experiments



- RayTracer: 3871.3 vs 6497.26 (pixels/s)
- RayTracer: 3989.09 vs 5186.18 (pixels/s)
- Euler: 214166.67 vs 242101.45 (gridpoints/s)

Experiments



- Susan1: 0.110 vs 0.105 (s) Susan2: 1.313 vs 1.210 (s)
- Stringsearch2: 0.067 vs 0.062 (s)
- Gsm1: 0.32 vs 0.37 (s) Gsm2: 15.56 vs 18.35 (s)

What's Next?

Analyzer is fast, brings good results,
implementation is stable: 1 year in production

chains of recurrences = subset of the SSA graphs
Abstractions over SSA graphs: see our paper at CPC'06
(http://cri.ensmp.fr/people/pop/papers/cpc2006.pdf)

Future work:

- improving the analyzers case by case (missed optimizations)
- data dependences on abstractions
- more optimizations (parallelization)
- hybrid static + dynamic optimizations