The New Framework for Loop Nest Optimization in GCC: from Prototyping to Evaluation

Sebastian Pop Albert Cohen Pierre Jouvelot
Georges-André Silber

CRI, Ecole des mines de Paris, Fontainebleau, France
ALCHEMY, INRIA Futurs, Orsay, France

January 10, 2006
CPC 2006

Motivations

We show that pattern matching algorithms on SSA graphs can naturally be described using the unification algorithm of PROLOG.

- graph transformations in PROLOG are interpretable and potentially shorter
- written communication of graph transformations simplified
- teaching compiler algorithms

Impact on the LNO of induction variable analysis that I integrated in GCC.

Outline

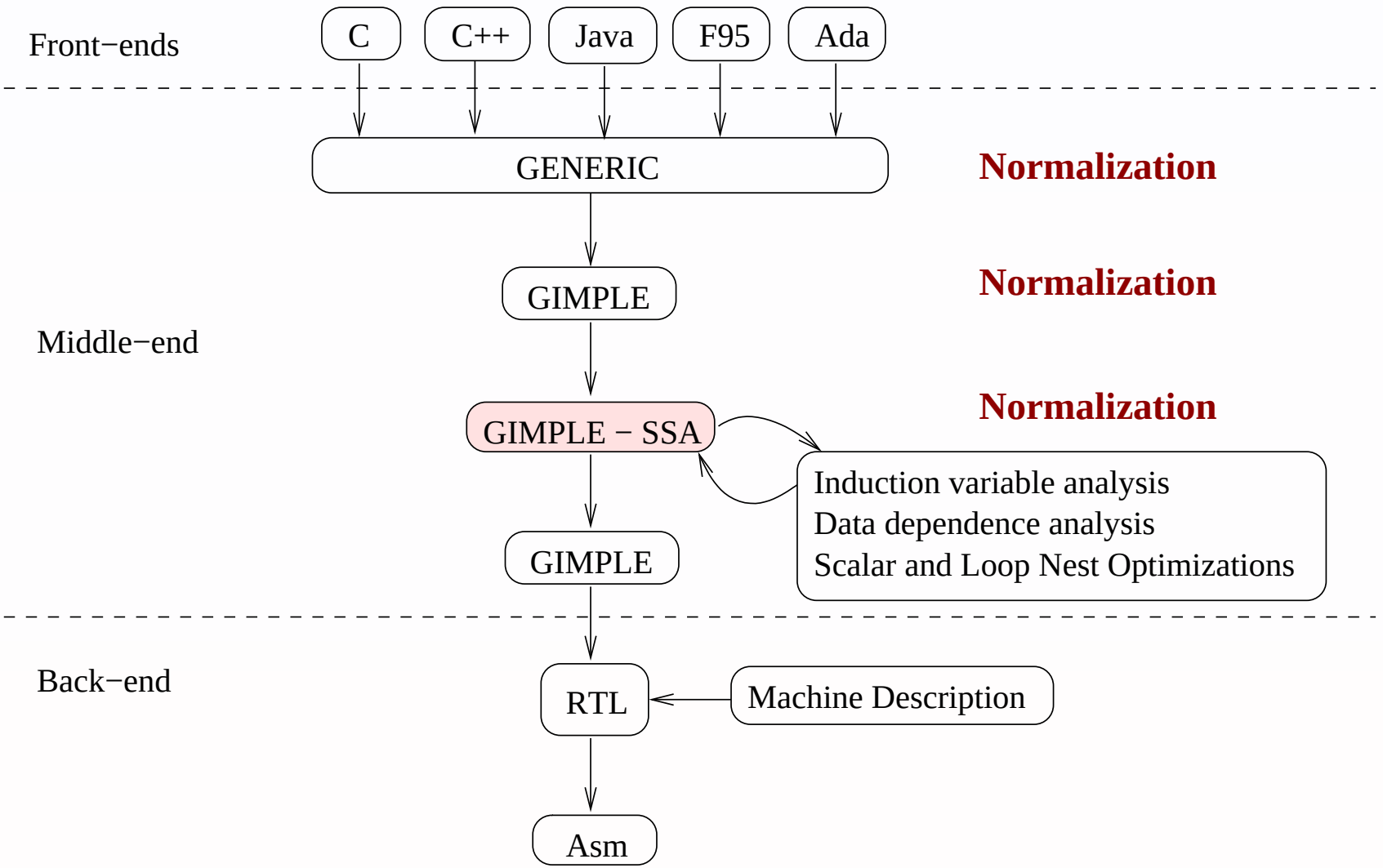
1. Context: intermediate representations and GCC
2. Fast prototyping of SSA transformations using PROLOG
3. Example: induction variable analysis.
4. Implementation and evaluation in GCC.

Context

Static analysis of programs in SSA form.
Working on the semantics of SSA programs with loops for:

- constant/range propagation
- vectorization, parallelization
- loop nest transformations

How is the SSA obtained in GCC?



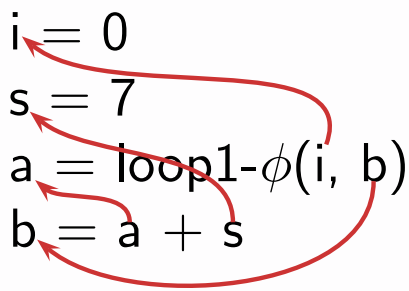
Example of GIMPLE in SSA Form

```
i = 0;
s = 7;
do {
  i = i + s;
} while (E);
```

GIMPLE / SSA ⟶

```
i = 0
s = 7
begin1:
  a = loop1-φ(i, b)
  b = a + s
  if (E)
    then goto begin1
    else goto end1
  endif
end1:
```

SSA ⟶



SSA Graphs in PROLOG

An SSA graph can be represented by a set of PROLOG facts.

$$\begin{array}{l} i = 0 \\ s = 7 \\ a = \text{loop1-}\phi(i, b) \\ b = a + s \end{array} \xrightarrow{\text{PROLOG}} \begin{array}{l} \text{assign(i, 0).} \\ \text{assign(s, 7).} \\ \text{assign(a, lphi(l1, i, b)).} \\ \text{assign(b, a + s).} \end{array}$$

Prototyping SSA Graph Transformations

Transformations of the SSA graphs can be defined by predicates using the unification engine of PROLOG. For example:

```
constProp(assign(X, A + B), assign(X, A + ValB)) :-
  assign(B, ValB), integer(ValB), !.
constProp(Default, Default).
```

```
assign(i, 0).
assign(s, 7).
assign(a, lphi(l1, i, b)).
assign(b, a + s).
```

$\longrightarrow$

```
constProp(assign(b, a + s),
          assign(b, a + 7)).
```

Abstract SSA Graphs

Uncertain values are replaced by approximations, for example:

- masking abstraction

```
removeMixers(assign(X, lphi(_, _, X + Step)), assign(X, unknown)) :-
  occurs(X, Step).
removeMixers(Default, Default).
```

- abstract elements

```
meetOverAllPaths(assign(X, cphi(_, A, B)), assign (X, meet(A, B))).
meetOverAllPaths(Default, Default).
```

Example: Induction Variable Analysis

Need a closed form representation for describing the value of scalar variables at each iteration of a loop nest for:

- data dependence analysis
- propagation of values/ranges after loops
- loop transformations

Representations that we will use:

- chains of recurrences (MCR) for polynomial functions (Bachmann, Zima), algebraic operations (van Engelen).
- lambda expressions for polynomials.

From SSA to Chains of Recurrences

```
fromSSAtoMCR(assign(X, Iphi(_, _, X + Step)), assign(X, unknown)) :-
  occurs(X, Step).
fromSSAtoMCR(assign(X, Iphi(LoopId, Init, X + Step)),
                  assign(X, mcr(LoopId, Init, Step))).
fromSSAtoMCR(Default, Default).
```

Some information is lost, there is no computation: MCR is a subset of SSA. Some examples:

```
?- fromSSAtoMCR(assign(a, Iphi(I1, 2, a + 2)), Q).
Q = assign(a,mcr(I1,2,2))

?- fromSSAtoMCR(assign(a, Iphi(I1, 3, a + mcr(I1, 4, 5))), Q).
Q = assign(a,mcr(I1,3,mcr(I1,4,5)))
```

From SSA to Lambda Functions

```
fromSSAtoLambda(assign(X, Iphi(_, _, X + Step)), assign(X, unknown)) :-
  occurs(X, Step).
fromSSAtoLambda(assign(X, Iphi(LoopId, Init, X + Step)),
                assign(X, Init + Result)) :-
  sumNFirst(LoopId, LoopId, Step, Result).
fromSSAtoLambda(Default, Default).
```

```
sumNFirst(L, N, C * lambda(L, binom(L, K)),
                C * lambda(N, binom(N, ResK))) :-
  integer(C), fold(K + 1, ResK).
```

$$\sum_{L=0}^{N-1} C \cdot \binom{L}{K} = C \cdot \binom{N}{K+1}$$

From SSA to Lambda Functions (Example)

fromSSAtoLambda(assign(a, lphi(l1, 2, a + 2)), Q).
Q = assign(a, 2+2*lambda(l1,binom(l1,1)))

fromSSAtoLambda(assign(a, lphi(l1, 3, a+(4+5*lambda(l1,binom(l1,1))))), Q).
Q = assign(a, 3+(4*lambda(l1,binom(l1,1))+5*lambda(l1,binom(l1,2))))

Semantics of SSA

Semantics of a subset of the SSA given by mappings to polynomial lambda expressions:

- SSA $\longrightarrow$ MCR $\longrightarrow$ Lambda
- SSA $\longrightarrow$ Lambda

We implemented SSA $\longrightarrow$ MCR in the GNU Compiler Collection, and we use the MCR algebra for computations.

However, there is no reason to prefer the MCR over the lambda expressions of polynomials for representing induction variables.

Implementation in GCC

No PROLOG, just plain C.

Analyzer of induction variables is fast and stable: 1 year in production versions of GCC (4.x). Other stable components in production:

- data dependence analysis (Banerjee, gcd, etc.)
- unimodular transformations of loop nests (interchange)
- vectorizer
- value range propagation

Parts of these components work on the results of the induction variable analysis.

Experiments in GCC

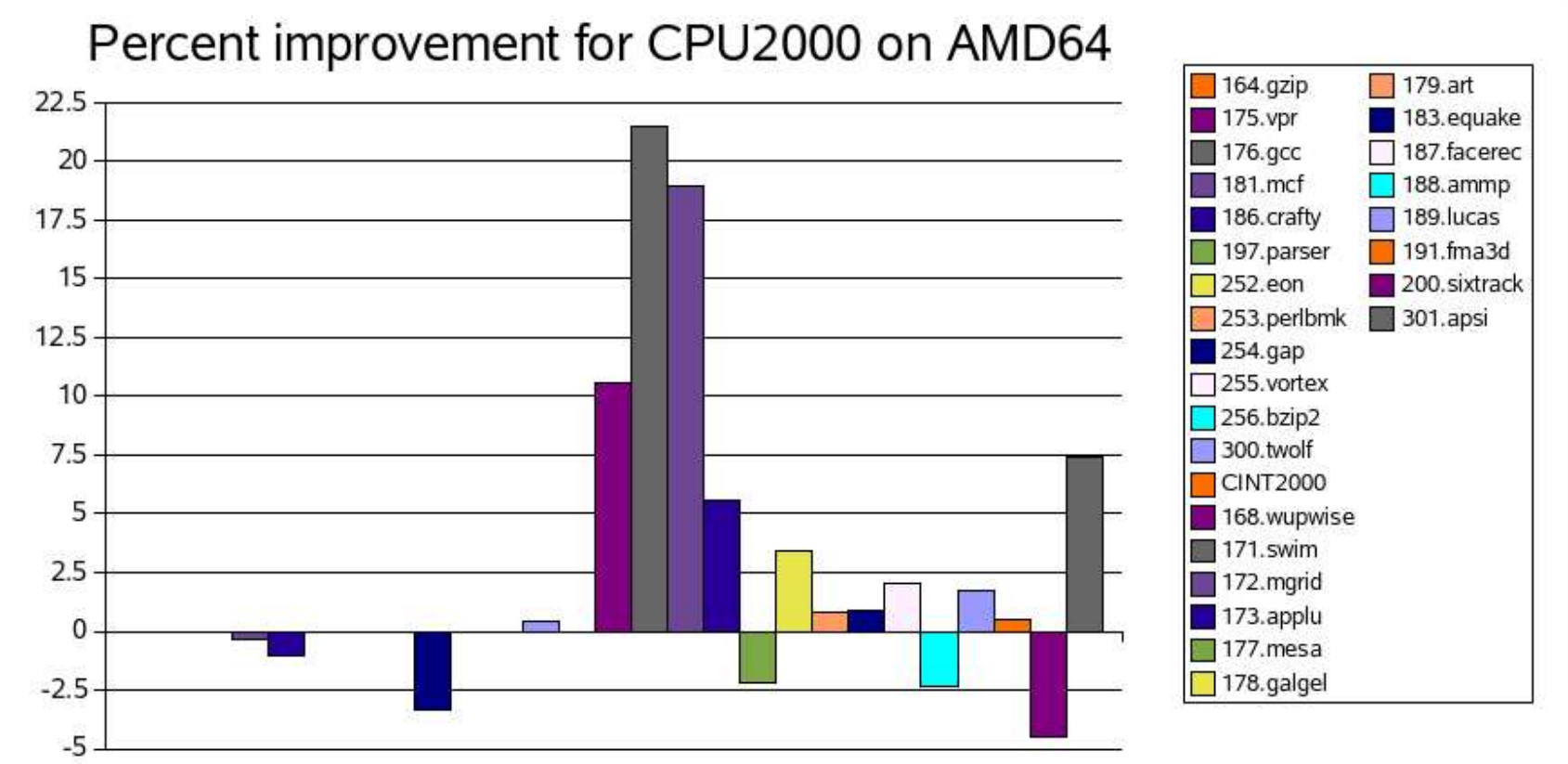
Base and peak compilers:

- 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

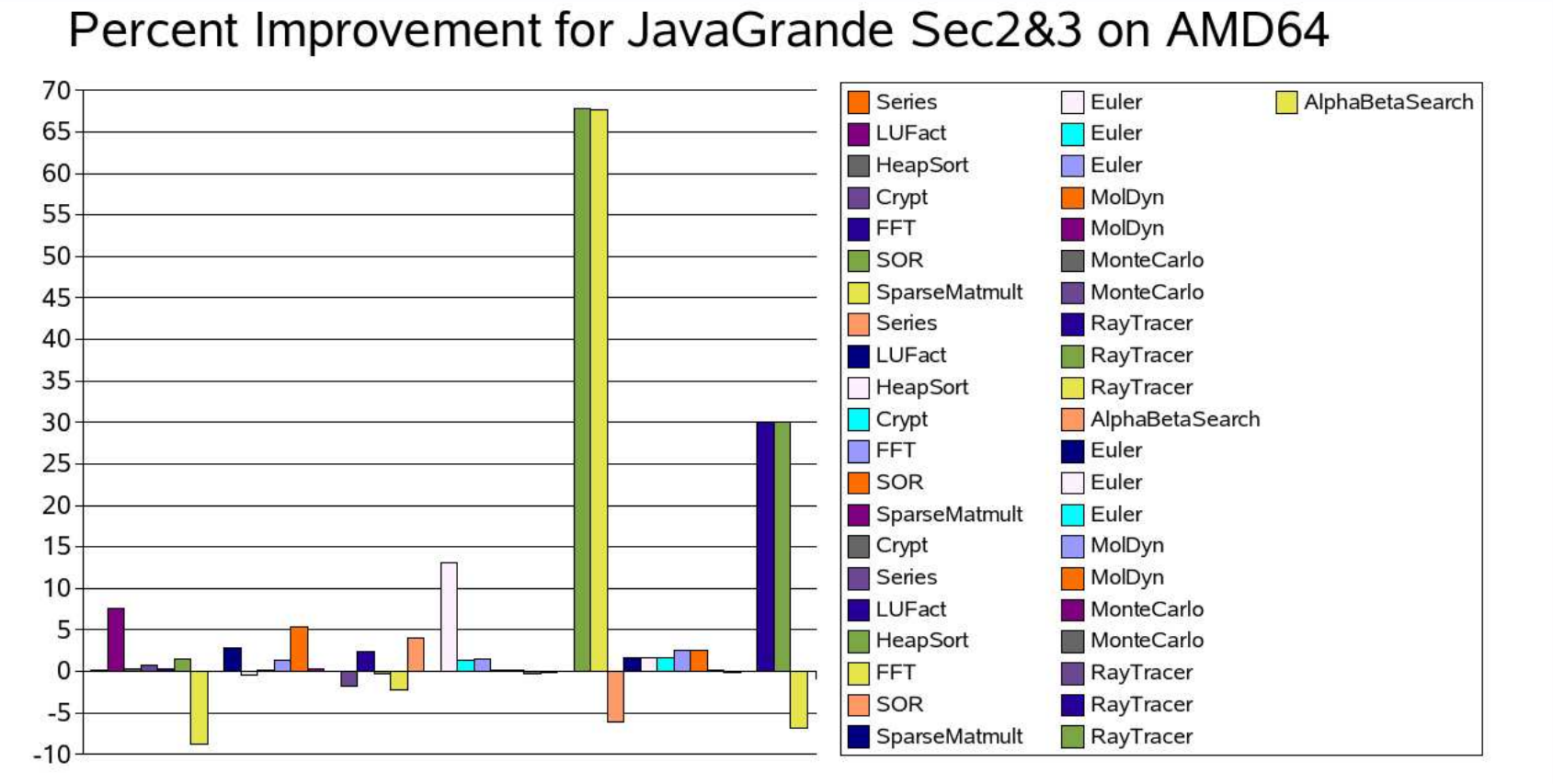
Benchmarks:

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

Experiments

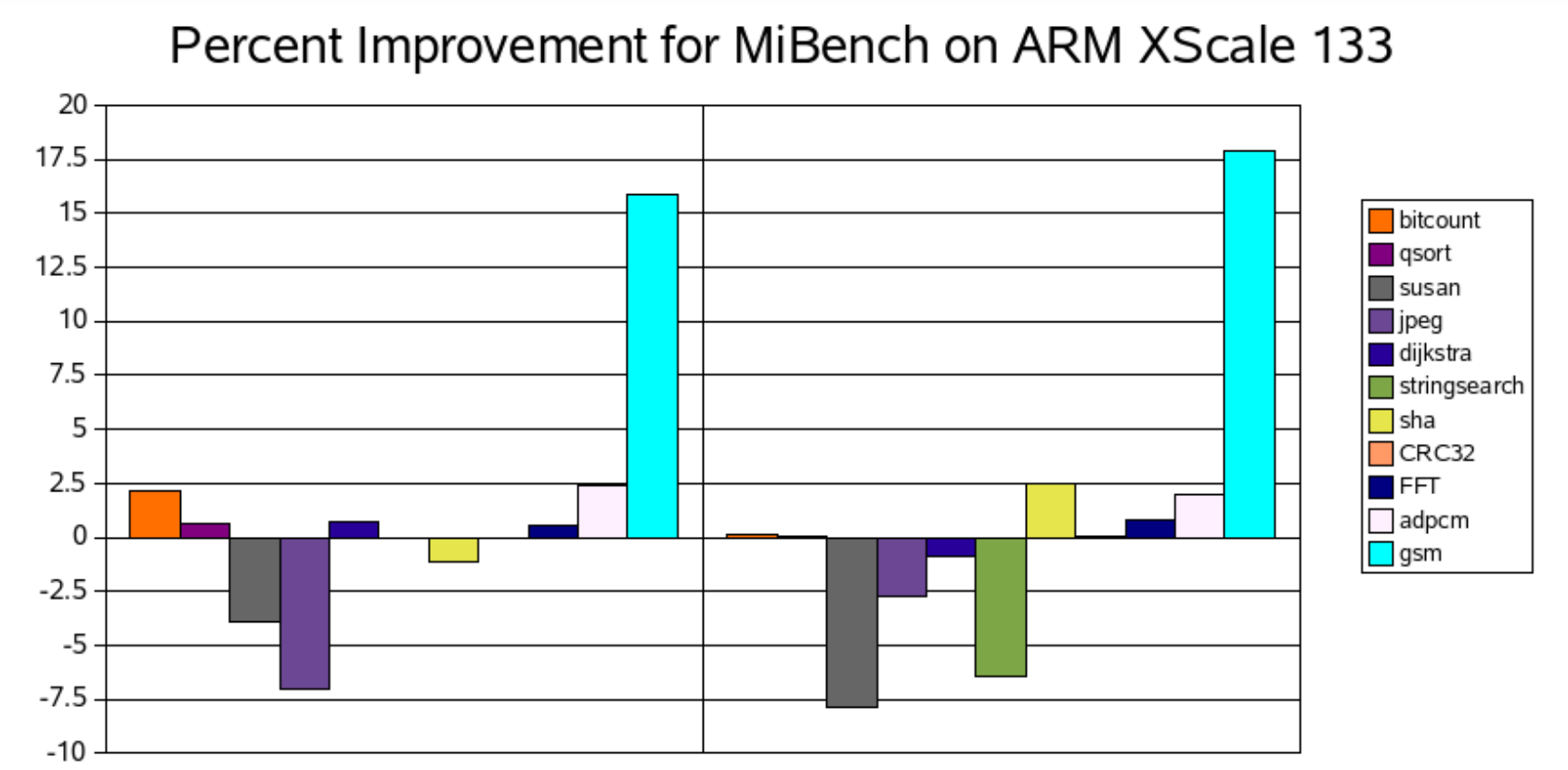


Experiments



- RayTracer: 3871 vs 6497 (pixels/s)
- RayTracer: 3989 vs 5186 (pixels/s)
- Euler: 214166 vs 242101 (gridpoints/s)

Experiments



- Susan1: 110 vs 105 (ms) Susan2: 1313 vs 1210 (ms)
- Stringsearch2: 67 vs 62 (ms)
- Gsm1: 320 vs 370 (ms) Gsm2: 15560 vs 18350 (ms)

Conclusions

- rapid prototyping environment
- transformations of SSA graphs in PROLOG
- abstract SSA graphs
- chains of recurrences = subset of the SSA graphs
- implementation is stable
- loop nest optimization framework is in place

Future work:

- improving the analyzers case by case (missed optimizations)
- data dependences on abstract SSA
- more optimizations (parallelization using OpenMP code gen)
- hybrid (static + dynamic) optimizations

Vision

We hope that the high performance and compilation communities are going to use GCC as their main research and implementation platform in the future.