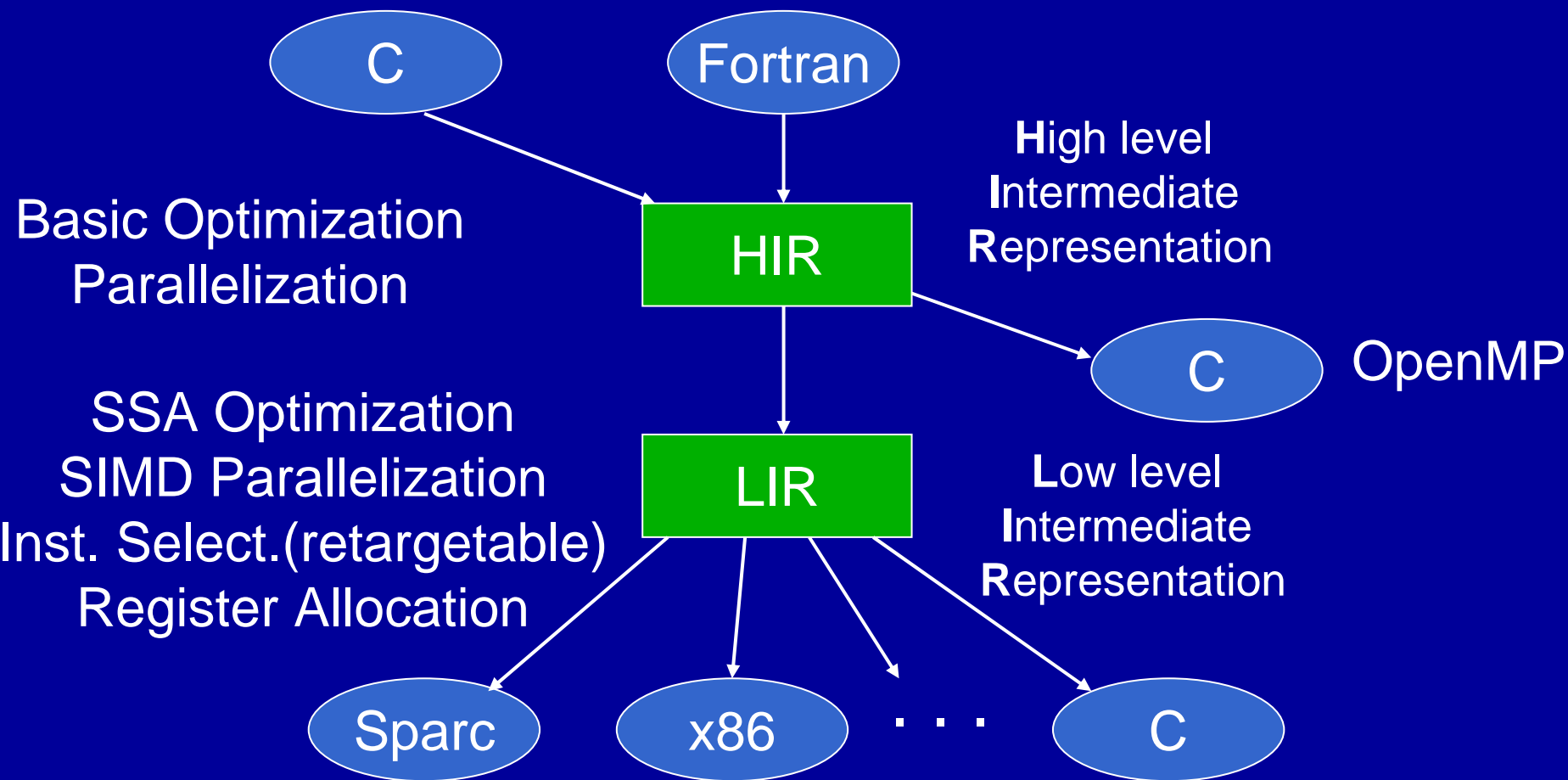


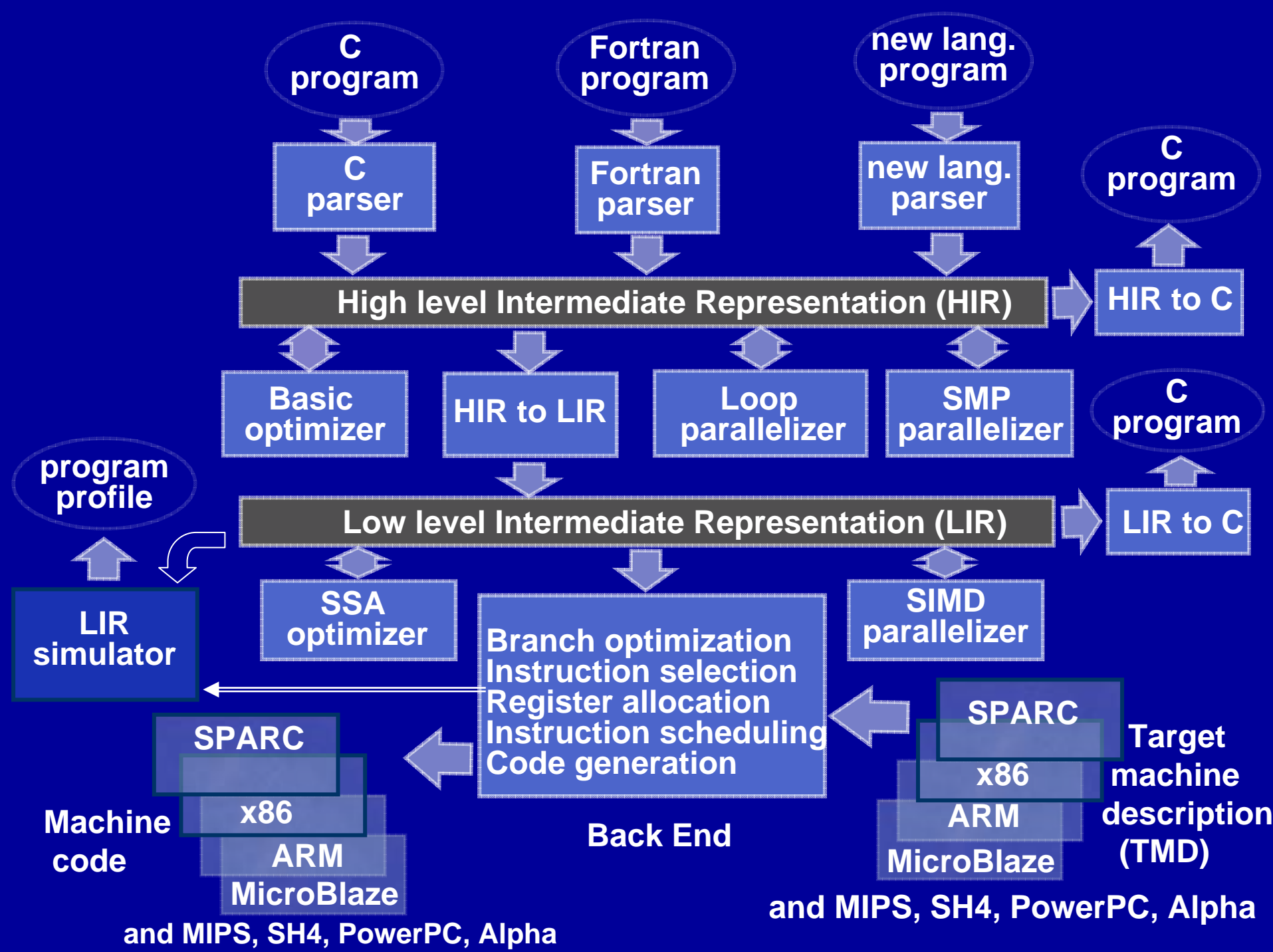
COINS: COmpiler INfra-Structure

Funded by
the Ministry of Education, Culture,
Sports, Science and Technology
2000 ~ 2004

The COINS System



Written in Java from scratch



COINS's features

- HIR (High level Intermediate Representation)
- LIR (Low level Intermediate Representation)
- Parsers (source program --> HIR)
 - C, Fortran, Java(planed)
- Optimizers for HIR/LIR
 - data flow based (HIR/LIR)
 - SSA based (LIR)
- Parallelizers for HIR/LIR
 - HIR --> OpenMP
 - SIMD parallelization (LIR)
- Code generators (LIR --> machine code)
 - retargetable code generator
 - Sparc, Intel x86

Project Members

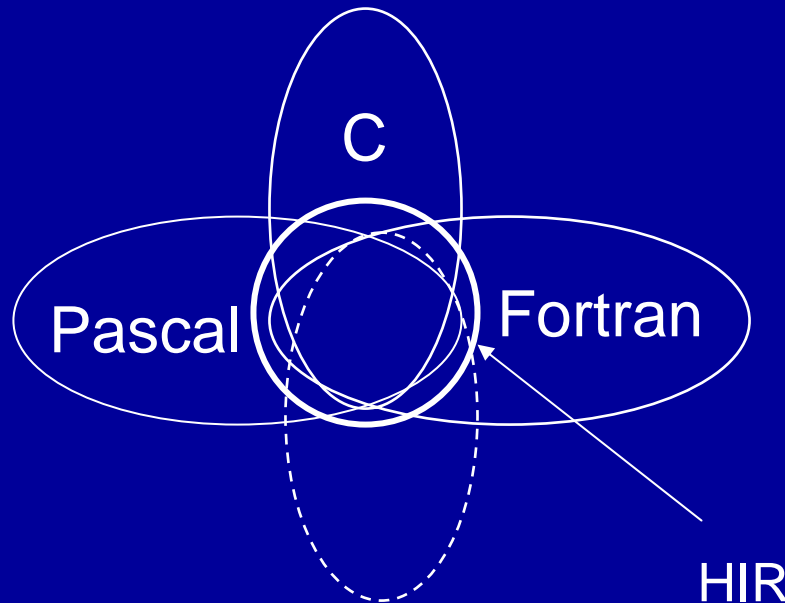
Ikuo Nakata (Hosei Univ.): Leader
Tan Watanabe (The Univ. of Electro-Communications)
Masataka Sassa (Tokyo Institute of Technology)
Tetsuro Fujise (Mitsubishi Research Institute, Inc.)
Nobuhisa Fujinami (SONY Corporation)
Toshitsugu Yuba (The Univ. of Electro-Communications)
Mitsugu Suzuki (The Univ. of Electro-Communications)
Seika Abe (The Univ. of Tokyo)
Toshihiro Nishioka (Mitsubishi Research Institute, Inc.)
Kyoko Iwasawa (Takushoku Univ.)
Sigeru Chiba (Tokyo Institute of Technology)
Kouichirou Mori (LSI Japan)
Takeaki Fukuoka (Kanrikogaku Kenkyusho, Ltd)
Munehiro Takimoto (Tokyo Univ. of Science)
Shinichiro Yamamoto (Aichi Prefectural Univ.)

Design principles of HIR (High level Intermediate Representation)

Abstract the basic features of languages currently prevailed.

Localize language-dependent information and machine –dependent information.

Generalize the basic features to cover many languages currently used or that will appear in near future.



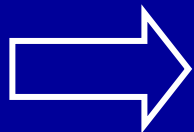
Design principles of LIR (Low level Intermediate Representation)

Rigorous semantic specifications (by denotational semantics)
-> Avoid troubles caused by ambiguity of specifications.

Organize as a language

LIR = gcc RTC + Module + Functional interface

in production use description of data and linkage clear interface



Modules of compilers for many languages and machines can be built by reading the LIR specifications without looking in its implementation.

Intermediate Representation (IR) and Symbol table

IR class structure

```
HIR
|- Program
|- SubpDefinition
|- Stmt
|   |- AssignStmt
|   |- IfStmt
|   |- LoopStmt
|   |   |- ForStmt
|   |   |- WhileStmt
|   :
|   :
|- Exp
|   |- ConstNode
|   |- SymNode
|   |   |- VarNode
|   |   |- SubpNode
|   |   :
|- SubscriptedExp
|- PointedExp
|- QualifiedExp
:   . . .
```

```
LIR
|- LIRNode
|   |- LConst
|   |- Register
|   |- Memory
|   |- LLabel
|   :
|   :
|- LIRTree
:   |- Set
|- Call
|- Jump
:   . . .
```

refer



Symbol class structure

```
Sym
|- Var
|   |- Param
|   |- Elem
|- Subp
|- Type
|   |- BaseType
|   |- VectorType
|   |- PointerType
|   |- StructType
|   :
|   :
|- Const
|- Label
|- Reg
```

Expansion for new language or new machine:

Add subclass (and factory method to create subclass objects).

Add IR access methods.

Revise SourceLanguage.java, MachineParam.java.

Expansion for new parallelization or optimization:

Add information to IR nodes (attachInf()).

Add information local to some phase (setWork (), getWork ()).

Add flags (setFlag (), getFlag()).

HIR image

Input program

```
for (i=0; i<10; i=i+1) {  
  a[i]=i;  
  ...  
}
```

HIR generation in syntax analyzer

(access methods correspond with source language features)

```
Stmt lLoopBody = h.blockStmt();  
forStmt (  
  h.assignStmt(  
    h.varNode(s_i), h.constNode(0)), ...  
  h.exp(HIR.OP_CMP_LT, varNode(s_i),  
    h.constNode(10)), ...,  
  lLoopBody, ...,  
  h.assignStmt(h.varNode(s_i),  
    h.exp(HIR.OP_ADD, h.varNode(s_i),  
    h.constNode(1))),  
  ... );  
lLoopBody.addLastStmt(h.assignStmt(  
  h.subscriptedExp(h.varNode(s_a),  
    h.varNode(s_i)), ...));  
...
```

HIR (abstract syntax tree with attributes)

```
(for  
  (assign <var i int> <const 0 int>)  
  (cmpLT <var i int> <const 10 int>)  
  (block  
    (assign  
      (subs <var a <VECT 10 int>> <var i int>)  
      <var i int>)  
    ....  
  )  
  (assign  
    <var i int>  
    (add <var i int> <const i int> ) ) )
```

Symbol table

```
s_i "i" var int  
s_a "a" var <VECT 10 0 int>  
....
```

Loop information

```
Parent loop  
Child loops  
Brother loops  
Induction variable i  
Basic block list
```

LIR image

Source program

```
for (i=0; i<10; i=i+1) {  
    a[i]=i; ...}
```

Part of LIR syntax

LIR

```
(set (mem (static (var i))) (const 0))  
(labeldef _lab5)  
(jumpc (tstlt (mem (static (var i)))  
              (const 10))  
        (list (label _lab6)  
              (label _lab4))))  
(labeldef _lab6)  
(set (mem (add (static (var a))  
              (mul (mem (static (var i)))  
                  (const 4))))  
      (mem (static (var i))))  
...
```

```
<Lprog> ::= ( {<Lmod>} )  
<Lmod> ::= ( MODULE <String> <GlobalAlist>  
{<Ldata>|<Lfunc>} )  
<GlobalAlist> ::= ( ALIST {<GlobalEnt>} )  
<GlobalEnt> ::= ( <String> <<RegEnt> | <StaticEnt> > )  
<LocalEnt> ::= ( <String> <<RegEnt> | <FrameEnt> > )  
<Ldata> ::= ( DATA <String> { <DataSeq> | <ZeroSeq> |  
<SpaceSeq> } )  
<Lfunc> ::= ( FUNCTION <String> <LocalAlist> <Lseq> )  
<Lseq> ::= ( {<Lexp>} )  
<Lexp> ::= <TypedExp> | <UnTypedExp>  
<TypedExp> ::= <AtomicTypedExp> | <NonAtomicTypedExp>  
<AtomicTypedExp> ::= <ConstExp> | <AddrExp> | <RegExp>  
<NonTypedAtomicExp> ::= <PureExp> | <MemExp> |  
<SetExp>  
...
```

Represent procedures as a sequence of subtrees corresponding to set, jump, call, etc. where, each leaf represents a variable, constant, or register.

Before instruction selection: memory accesses and operations are represented in abstract form so as optimum instruction sequence can be selected.

After instruction selection: each instruction corresponds to a machine instruction in LIR grammar.

Not only instructions but also data and interfaces are described so that LIR can be transferred in text form.

HIR and flow analysis data

Source program

```
a=1;
if (a==0)
  c=a;
else
  c=a+2;
```

HIR

```
(assign
 <var a>
 <const 1>)
(if
 (cmpEq
  <var a>
  <const 0>)
 (labeledSt
  <labelDef _lab3>
  (assign
   <var c>
   <var a>))
 (labeledSt
  <labelDef _lab4>
  (assign
   <var c>
   (add
    <var a>
    <const 2>))))
 (labeledSt
  <labelDef _lab5>))
```

B1
defined a
availOut a
used a

B2
defined c
availIn a
availOut a,c
used a

B3
defined c
availIn a
availOut a,c
used a

B4
availIn a,c

Control flow is represented by a directed graph of basic blocks. Data flow information is attached to basic blocks. There are mutual linkages between HIR and flow data. Do analysis again if program is changed.

→ Control flow
↔ Information link

Optimization in Static Single Assignment (SSA) Form

```
1: a = x + y
2: a = a + 3
3: b = x + y
```

(a) Source program

SSA
translation

```
1: a1 = x0 + y0
2: a2 = a1 + 3
3: b1 = x0 + y0
```

(b) SSA form

Optimization in SSA form (common
subexpression elimination)

```
1: a1 = x0 + y0
2: a2 = a1 + 3
3: b1 = a1
```

(c) After SSA form optimization

SSA back
translation

```
1: a1 = x0 + y0
2: a2 = a1 + 3
3: b1 = a1
```

(d) Optimized normal form

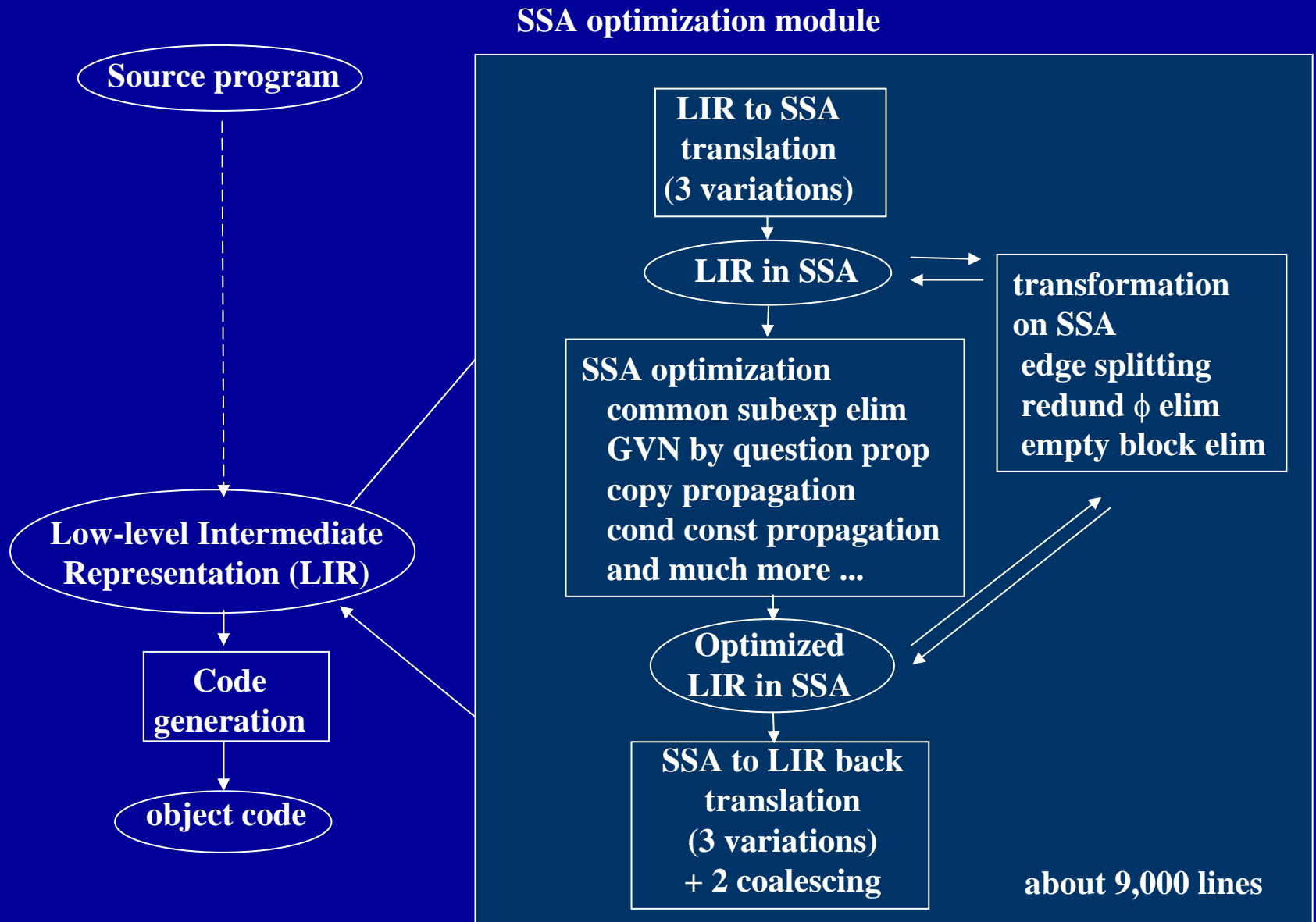
SSA form is a new internal representation where indices are attached to variables so that their definitions become unique.

Suited for clear handling of dataflow analysis and optimization in compilers.

Outline of SSA module

- Translation into and back from SSA form on Low-level Intermediate Representation (LIR)
 - SSA translation: Use dominance frontier
 - SSA back translation: Sreedhar et al.'s method
- Several optimization on SSA form:
 - dead code elimination, copy propagation, common subexpression elimination, global value numbering based on efficient query propagation, conditional constant propagation, loop invariant code motion, operator strength reduction for induction variable and linear function test replacement, empty block elimination ...
- Useful transformation for SSA form optimization
 - critical edge removal, loop transformation ...
- Each variation, optimization and transformation can be made selectively by specifying options

SSA optimization module in COINS



Example of SSA Optimization

source program (from Appel's Book)

```
int main () {  
    ...  
    i = 1;  
    j = 1;  
    k = 0;  
    while (k < 100) {  
        if (j < 20) {  
            j = i;  
            k = k + 1;  
        } else {  
            j = k;  
            k = k + 2;  
        }  
    }  
    printf ("%d\n", j);  
}
```

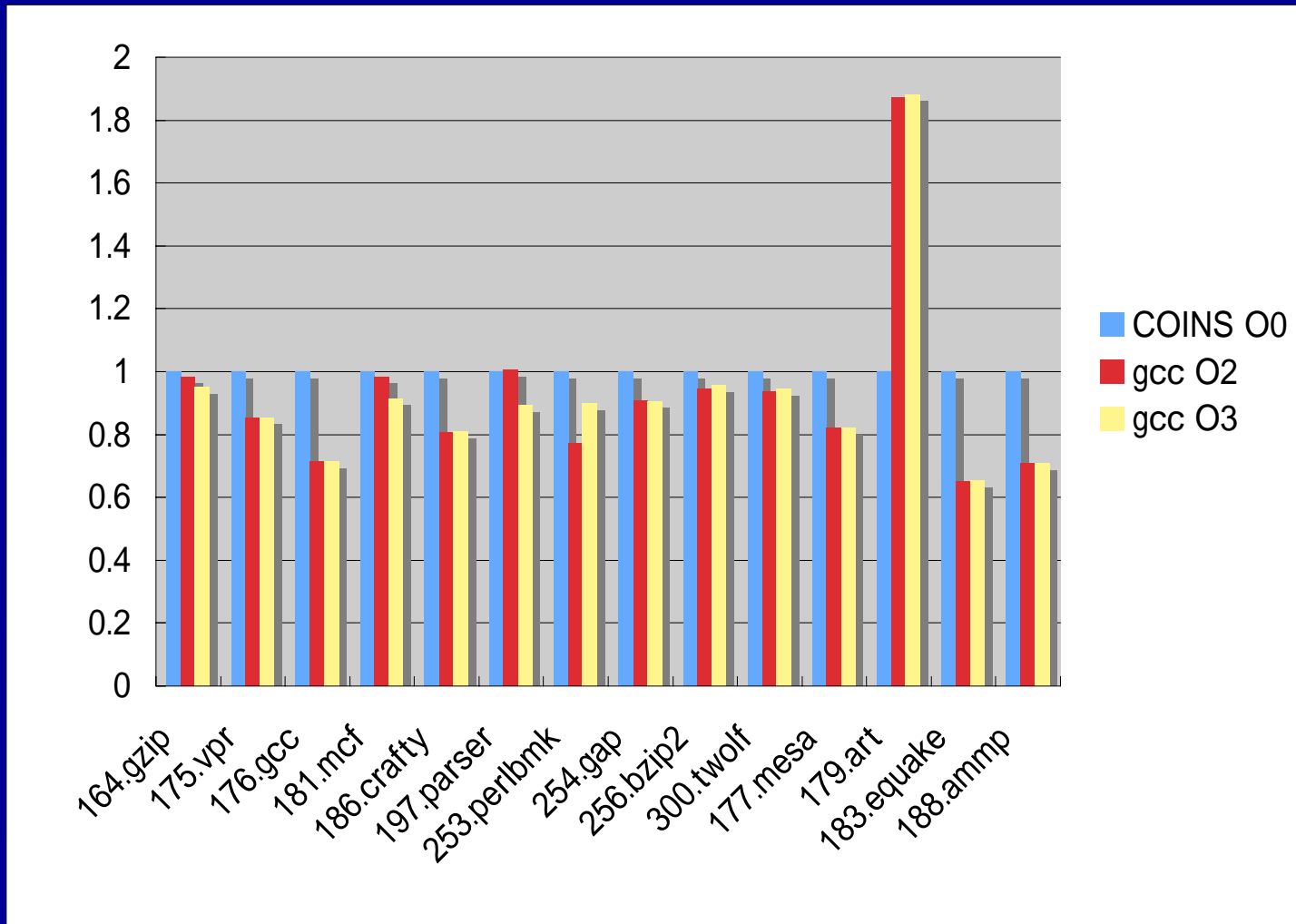


result of SSA optimization

```
int main(void) {  
    ...  
    ar3_2 = 0;  
_lab3:  
    if(ar3_2>=100) goto _lab8;  
    ar3_2 = ar3_2 + 1;  
    goto _lab3;  
_lab8:  
    printf ("%d\n", 1);  
}
```

translated from LIR to C
by the option 'lir2c'

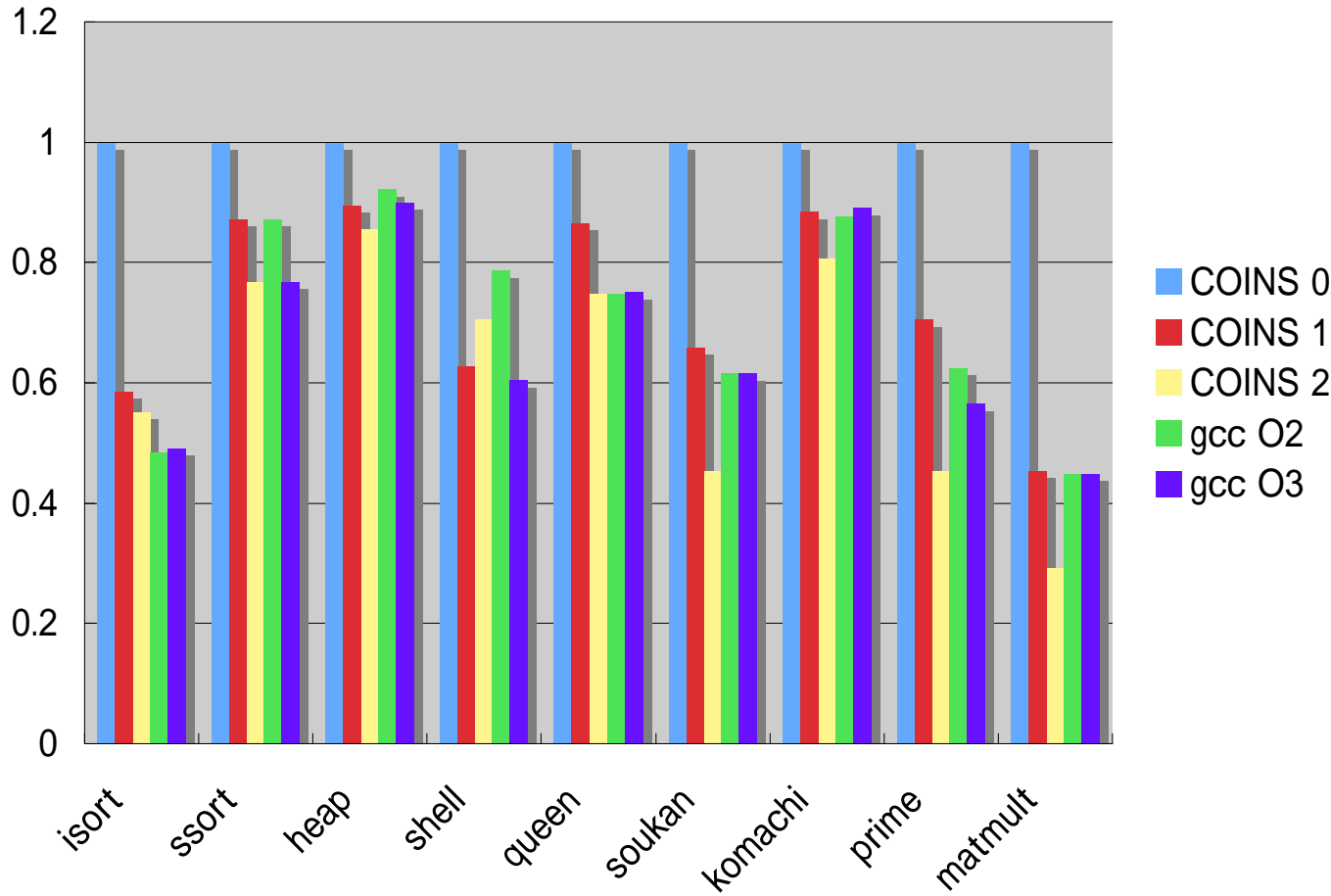
Execution time ratio (SPEC2000, x86)



On Epson
Pro-1000
Pentium 4
1.8 GHz
256 MB

Execution time ratio compared to COINS O0
(no optimization option)

Execution time ratio (small programs, Sparc)



isort: insertion sort
ssort: selection sort
heap: heap sort
shell: shell sort
queen: 8 queens
soukan: correlation coefficient
komachi: number decomposition
prime: prime numbers
matmult: matrix multiply

on Sun Blade 1000
2 x UltraSPARC-III
750 MHz, 1 GB

COINS: 0 = no optimization option

1 = ssa: prun/divex/cse/cstp/hli/osr/hli/cstp/cyp/reqp/cstp/rpe/dce/srd3

2 = ssa+instr. schedule + register promotion + pipelining

gcc O2, O3: gcc with -O2, -O3 option

Retargetable Code Generator

LIR

Control Flow Analysis

Branch Optimization

Instruction Selection

Register Allocation
(Graph Coloring)

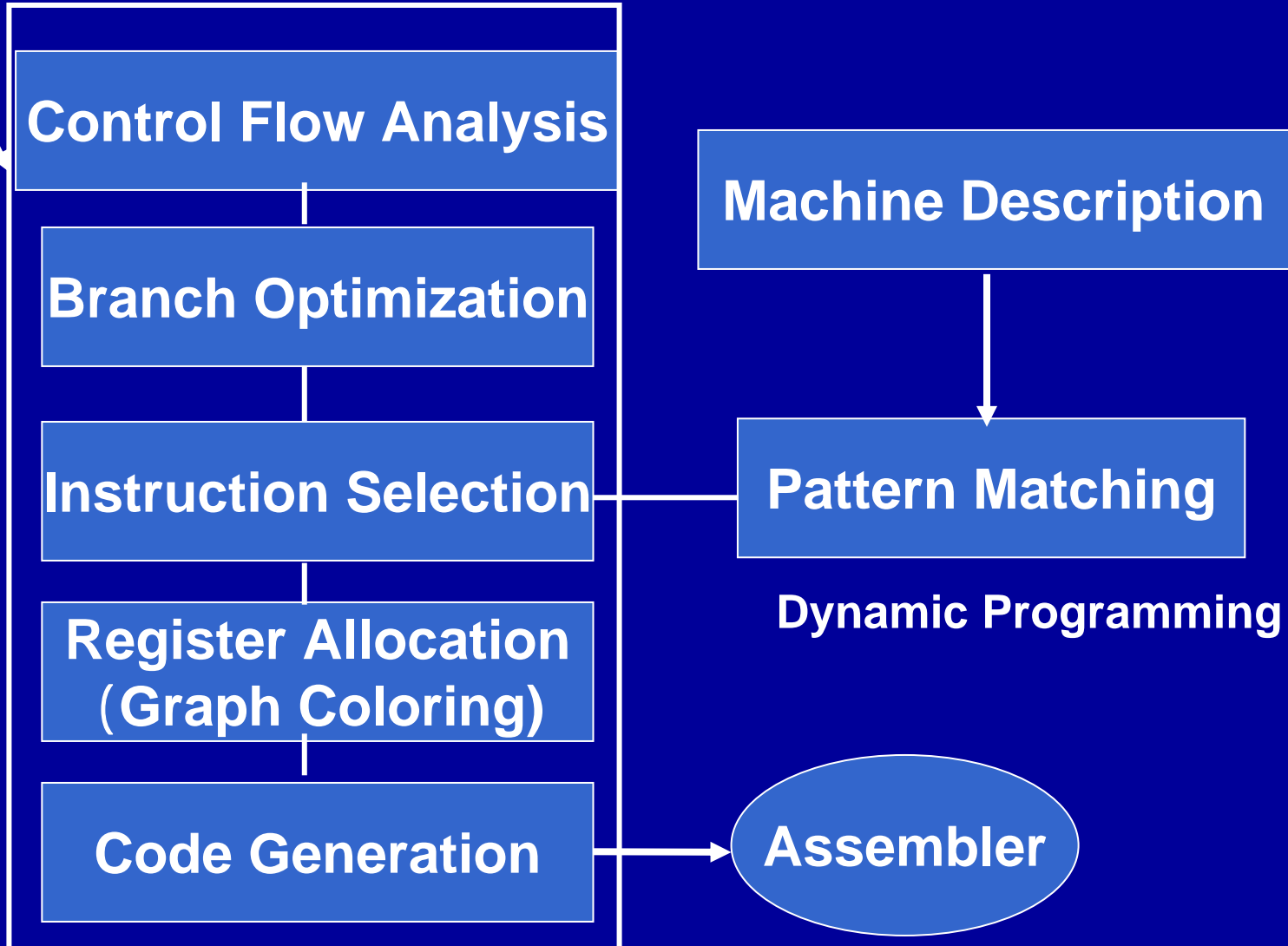
Code Generation

Machine Description

Pattern Matching

Dynamic Programming

Assembler



Machine Description

Register Definition *%i0 - %i5, %o0 - %o5, %l0 - %l7*

```
(def *reg-I32* ( (foreach @io (i o)
                 (foreach @n (0 1 2 3 4 5) (REG I32 "%@io@n"))))
  int 32bits (foreach @n (0 1 2 3 4 5 6 7) (REG I32 "%l@n")) ))
```

Instruction Description *LIR Sparc*

```
(foreach (@op @code) ((ADD add) (SUB sub)
                      (BAND and) (BOR or) (BXOR xor))
  (defcode @code (SET I32 reg (@op I32 reg rc))
  (asm `(@code, $1, $2, $0))      rc: reg or const
  (cost 1)))      cost of this instruction
(foreach (@n @l) ((2 1) (4 2) (8 3) (16 4) (32 5)) ;; mult by shift
  (defcode mul-sll@l (SET I32 reg (MUL I32 reg (INTCONST I32 @n)))
  (asm `(sll , $1 (con @l) , $0))      con @l = 1, 2, 3, 4, or 5
  (cost 1)))
```

Example of Code Generation

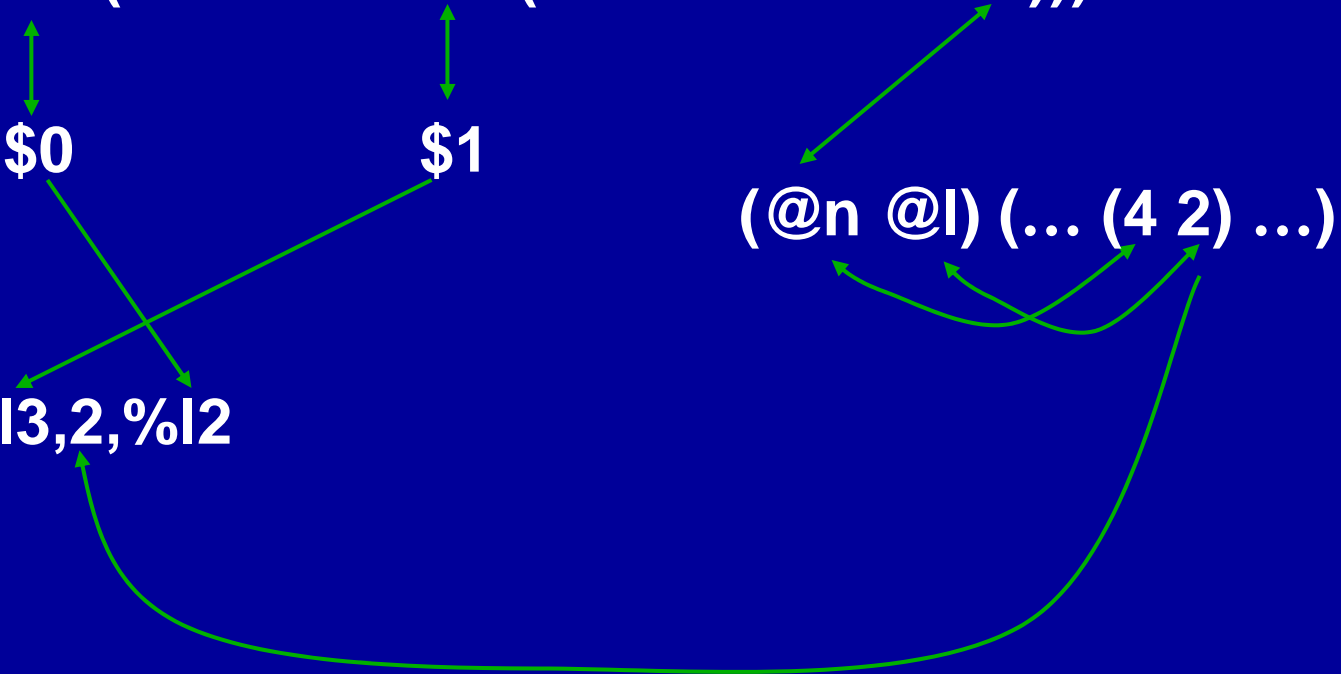
```
(foreach (@n @I) ((2 1) (4 2) (8 3) (16 4) (32 5))  
  (defcode mul-sll@I (SET I32 reg (MUL I32 reg (INTCONST I32 @n)))  
    (asm `(sll ,$1 (con @I) ,$0)  
      (cost 1)))
```

LIR: (SET:I32 %I2 (MUL:I32 %I3 (INTCONST:I32 4)))

\$0 \$1

(@n @I) (... (4 2) ...)

Sparc: sll %I3,2,%I2



Examples of retargeting

Machine	Coded Lines	Months	Note
SPARC	1952	-	not available
x86	2533	-	not available
MIPS	2207	3	nonexperienced student
SH4	3596	6	nonexperienced student
ARM	3052	6	nonexperienced
ARM-Thumb	1980	3	nonexperienced
MicroBlaze	1383	2	experienced
Power PC	5018	6	nonexperienced student
Alpha	1216	2	nonexperienced student

Parallelization

- Generation of parallelized object code for do-all loops
 - Executable even in environment without OS.
- Basic Parallelization (HIR --> OpenMP)
 - Loop Parallelization
- SMP (Symmetric Multi-Processor) Parallelization (HIR --> OpenMP)
 - Coarse-grain Parallel Processing
- SIMD (Single Instruction Multiple Data stream) Parallelization (LIR --> SIMD inst.)

Basic Parallelization

Example of translation (C --> HIR --> OpenMP)

C program

Generated OpenMP program

```
int i, j, n, m;  
float a[100], b[100], s;  
.
```

```
j=0 ;  
for (i=n ; i<m; i=i+1) {  
  s = a[i]+b[j] ;  
  if (s>0) a[i] = s*j ;  
  j = j+2;  
}
```

```
j=0 ;  
for (i=0 ; i<m; i=i+1) {  
  a[j] = a[i]+s ;  
  if (b[j]>0) s = 1./b[k];  
  j = j+2;  
}
```

insertion of
OpenMP
directives

Normalization
/elimination of
induction
variables

explanation
of causes
preventing
parallelization

```
int i, j, n, m, $i;  
float a[100], b[100], s;  
.  
j=0;
```

```
#pragma omp parallel for  
private(i,j) lastprivate(s)
```

```
for ($i=0 ; $i<m-n-1; $i=$i+1) {  
  i=$i+n;  
  j=$i*2;
```

```
  s = a[i]+b[j] ;  
  if (s>0) a[i] = s*j ;  
}
```

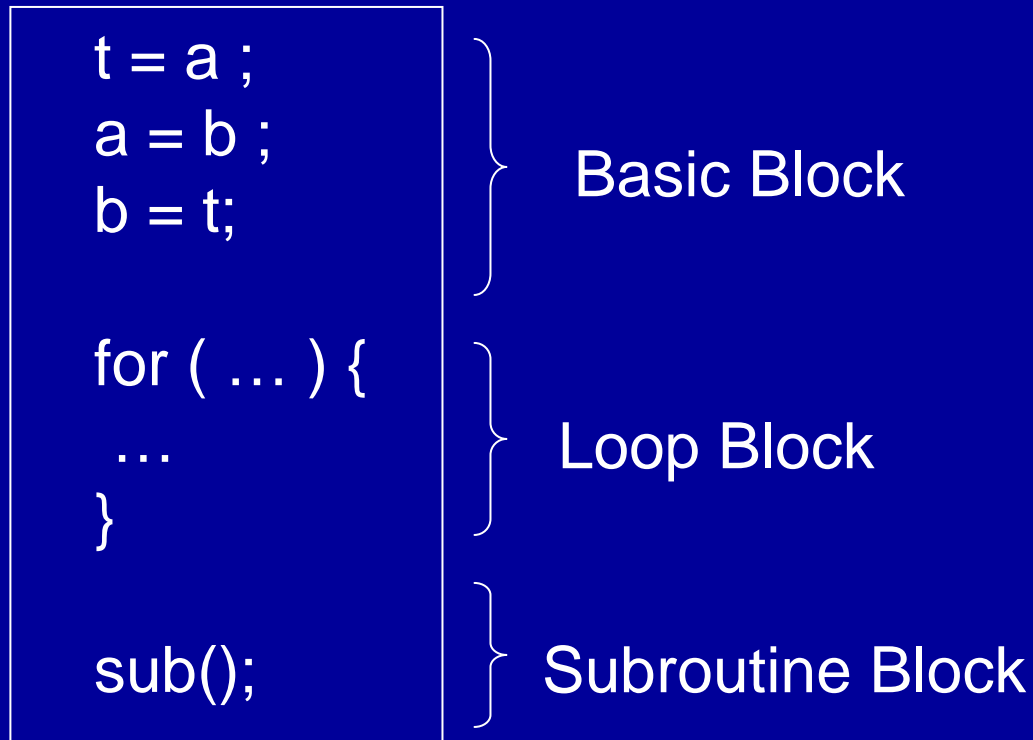
```
j=0 ;  
//loop parallelization impossible  
//loop carried data dependency of a[] and s
```

```
for (i=0 ; i<m; i=i+1) {  
  a[j] = a[i]+s ;  
  if (b[j]>0) s = 1/b[j] ;  
  j = j+2;  
}
```

SMP Parallelization

- Decompose sequential program to coarse-grain tasks, then generate **coarse-grain task graph**
- Generate parallel program with dynamic scheduling code and OpenMP directives

Example of Coarse-grain Tasks



Each Block is a Coarse-grain Task

SIMD Parallelization

- Precise and detailed description of each machine instruction in SIR (extended LIR)
 - Intel x86 1,307,399B
 - PowerPC G4 847,967B
 - SPARC v.8 32 104,177B
 - SPARC v.9 64 254,062B
- Parallelization Algorithms
 - extended peephole optimization for SIMD instruction
 - if-conversion by logical operation
 - special pattern matching for SIMD instruction
 - data size inference
- Example
 - three times faster object code (compared with the result of existing optimized compiler)

Example of SIMD Parallelization

C program

```
#define ABSDIF(x, y) ((x <= y) ? y-x : x-y)

int error(char xx[16][16], char yy[16][16]){
char *across;
char *cacross;
int diff=0;
int y;
for (y=0;y<16;y++) {
    across = &xx[y][0];
    cacross = &yy[y][0];
    diff += ABSDIF(across[0], cacross[0]);
    diff += ABSDIF(across[1], cacross[1]);
    diff += ABSDIF(across[2], cacross[2]);
    diff += ABSDIF(across[3], cacross[3]);
    diff += ABSDIF(across[4], cacross[4]);
    diff += ABSDIF(across[5], cacross[5]);
    diff += ABSDIF(across[6], cacross[6]);
    diff += ABSDIF(across[7], cacross[7]);
    diff += ABSDIF(across[8], cacross[8]);
    diff += ABSDIF(across[9], cacross[9]);
    diff += ABSDIF(across[10], cacross[10]);
    diff += ABSDIF(across[11], cacross[11]);
    diff += ABSDIF(across[12], cacross[12]);
    diff += ABSDIF(across[13], cacross[13]);
    diff += ABSDIF(across[14], cacross[14]);
    diff += ABSDIF(across[15], cacross[15]);
};
return diff;
}
```



Sparc program

```
save %sp, -8, %sp
mov (0), %l0
mov (0), %l1
_lab5:  cmp %l1, (16)
        bl _lab6
        nop
        ba _lab4
        nop
_lab6:  mov %i0, %l2
        smul %l1, (16), %l3
        add %l2, %l3, %l4
        mov %i1, %l5
        add %l5, %l3, %l6
        ldd [%l4], %f0
        ldd [%l6], %f2
        st %l0, [%i6-(4)]
        st %g0, [%i6-(8)]
        ldd [%i6-(8)], %f4
        pdist %f0, %f2, %f4
        ldd [%l4+(8)], %f0
        ldd [%l6+(8)], %f2
        pdist %f0, %f2, %f4
        std %f4, [%i6-(8)]
        ld [%i6-(4)], %l0
        add %l1, (1), %l1
        ba _lab5
        nop
_lab4:  mov %l0,%i0
        ret
        restore
```

pdist: SIMD instruction

8 parallel additions of absolute differences

PDIST instruction (1/4) in SIR

```
(DEFINST ("pdist ?1f, ?2f, ?3f" "pdist ?1f, ?2f, ?3f")
(PARALLEL
  (SET (SUBREG I32 (REG F64 (HOLE 3 FREG)) 0)
  (ADD I32 (ADD I32 (ADD I32 (ADD I32 (ADD I32 (ADD I32 (ADD I32
  (ADD I32
  (SUBREG I32 (REG F64 (HOLE 3 FREG)) 0)
  (BOR I32
    (BAND I32
      (SUB I32 (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 2 FREG)) 0))
        (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 1 FREG)) 0)))
      (TSTLES I32 (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 1 FREG)) 0))
        (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 2 FREG)) 0))))))
    (BAND I32
      (SUB I32 (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 1 FREG)) 0))
        (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 2 FREG)) 0)))
      (BNOT I32
        (TSTLES I32 (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 1 FREG)) 0))
          (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 2 FREG)) 0))))))
  (BOR I32
    (BAND I32
      (SUB I32 (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 2 FREG)) 1))
        (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 1 FREG)) 1)))
      (TSTLES I32 (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 1 FREG)) 1))
        (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 2 FREG)) 1))))
    (BAND I32
      (SUB I32 (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 1 FREG)) 1))
        (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 2 FREG)) 1)))
      (BNOT I32
        (TSTLES I32 (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 1 FREG)) 1))
          (CONVZX I32 (SUBREG I8 (REG F64 (HOLE 2 FREG)) 1))))))
  .....
  .....
```

Execution Time

(10^7 calls of error())

GCC -O0	223.46	sec	
GCC -O2	44.89		
COINS	281.71		(with old backend)
COINS	65.15		(with new backend)
COINS-SIMD	13.68		
COINS-SIMD	9.31		(if data are aligned)

version: GCC-3.3, COINS-0.10.1

Sun Ultra 60 (2 X UltraSPARC-II 450MHz)

Memory 1GB

URL

<http://www.coins-project.org/international/index.html>

Abstract

Overview(pdf)

Compilation examples

Sub Projects

- Base

- SSA optimization

- SIMD parallelisation

How to use Coins

Developing new compiler with Coins

Research organization

Members

Download Coins