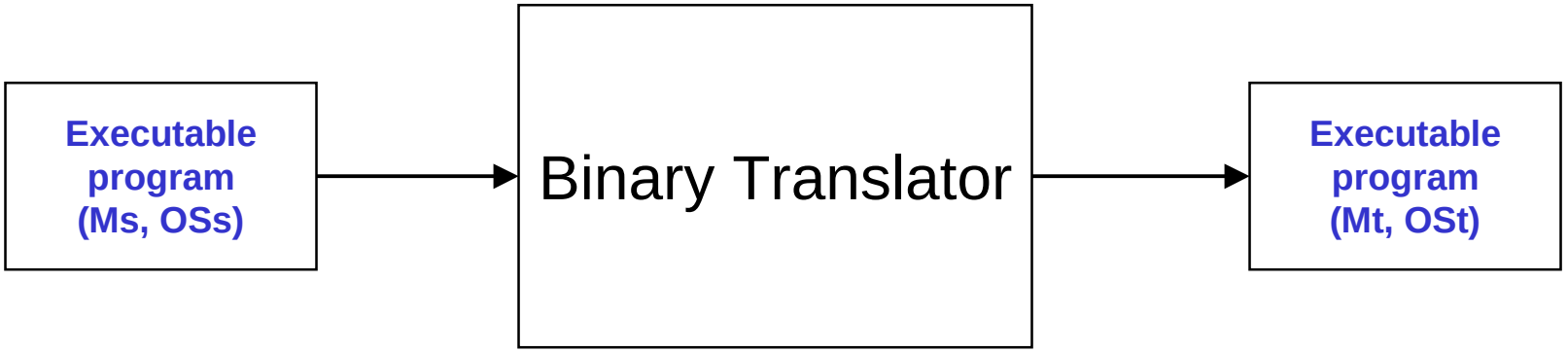
Retargetable Binary Translation

Cristina Cifuentes

Sun Microsystems Labs

The binary translation process



Overview

- Uses of binary translation
- Retargetability within UQBT
- Example
- Results
- Other uses of machine specifications
- Future areas of impact
- Conclusions

Uses of binary translation

- Migrate off legacy platforms (late 1980s)
  - HP (HP3000 → PA-RISC)
  - Tandem's Accelerator (TNS/CISC → TNS/RISC)
  - Digital's VEST (VAX → Alpha)
- Run competitor's binaries (1990s)
  - AT&T's Flashport (Mac 68K → PowerPC)
  - Digital's FX!32 (x86/WinNT → Alpha/WinNT)

Uses of Binary Translation

- Recently
  - IBM's Daisy (PowerPC → VLIW)
  - Transmeta's code morphing (x86 → Crusoe)
  - HP's Aries (PA-RISC → IA-64)
  - Compaq's Tandem (TNS/CISC → Alpha)

Binary Translation

- Related areas
  - Emulation
    - Since the 60s
    - Wabi, Sun Microsystems (94)
  - Simulation
    - Shade, Sun Microsystems (94)
    - SimOS and Embra, Stanford (95)
  - Dynamic reoptimization
    - HP Labs' Dynamo (00)
    - Compaq's Wiggins/Redstone (00)
  - Instrumentation
    - Universitat Politècnica de Catalunya's Dixie (99)
  - …

The UQBT Project

Goals

- To improve understanding of binary code manipulation
- To understand what knowledge of machines and OS's is needed to perform binary translation
- To formally specify that knowledge in description languages

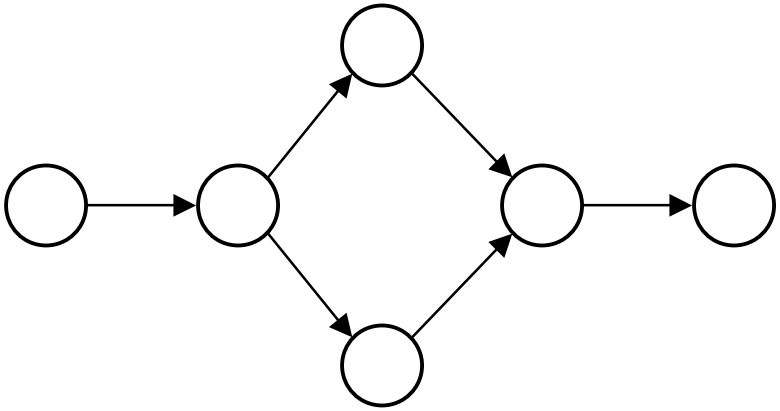
Goals

- To understand how to implement machine-dependent analyses on the intermediate representations
- To understand which analyses can be made machine-independent, and how
- To provide a framework for experimental binary translation

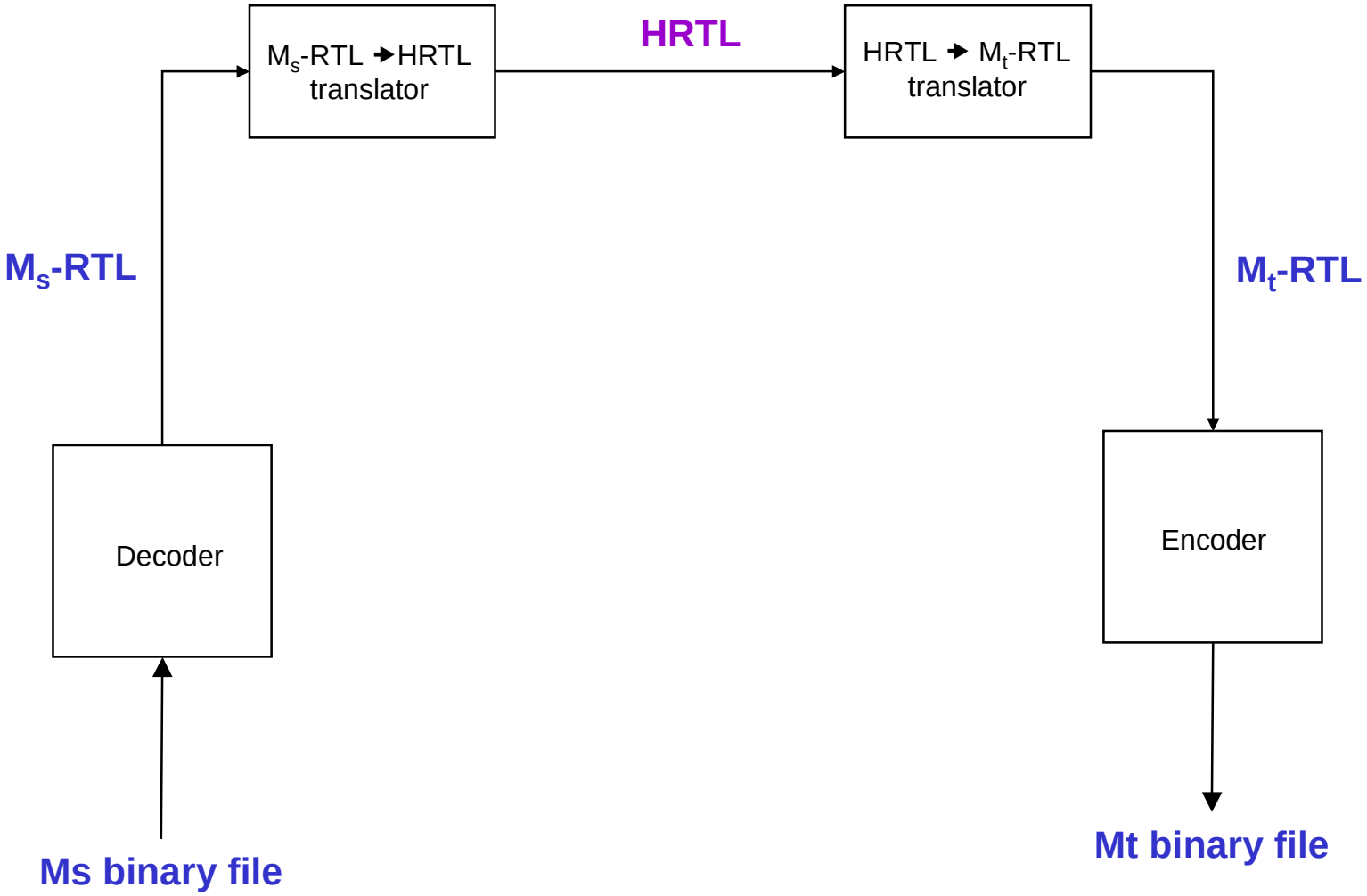
Intermediate representations used in the UQBT framework

Intermediate representations

- Control flow graphs



Translation block diagram and intermediate representations



Intermediate representations

- Register transfer lists (RTLs)
  - low-level instruction representation
  - represents effects of the machine instructions
  - locations
    - infinite number of registers r[x]
    - memory m[x]
  - instructions
    - assignment

Intermediate representations

- Higher-level register transfer language (HRTL)
  - locations
    - infinite number of registers r[x]
    - infinite number of variables v*x*
    - memory m[x]
  - instructions
    - assignment, conditional branch, unconditional branch, call, return
    - arithmetic, logical and swapN expression

RTL example

```
value = fib (number);
```

```
ld [%fp-20],%o0
call 0x10a9c
nop
st %o0,[%fp-24]
```

Sparc-RTL

```
*32* r[8] := m[r[30] - 20]   // load parameter
*32* r[15]:= 0x010b40        // call fib()
*32* %pc := %npc
*32* %npc := 0x010a9c
*32* m[r[30] - 24] := r[8]   // store return value
```

RTL example

```
value = fib (number);
```

```
movl 0xfffffffc(%ebp),%eax
pushl %eax
call 0x8048960
addl $0x4,%esp
movl %eax,0xfffffff8(%ebp)
```

Pentium-RTL

```
*32* r[24] := m[r[29] - 4]  // load parameter
*32* r[28] := r[28] - 4     // put it on the stack
*32* m[r[28]] := r[24]
*32* r[28] := r[28] - 4     // call fib()
*32* m[r[28]] := 0x80489d0
*32* %pc := 0x8048960
*32* r[28] := r[28] + 8     // fix stack frame
*32* r[24] := r[24]
*32* m[r[29] - 8] := r[24]  // assign return value
```

HRTL example

```
value = fib (number);
```

HRTL (Sparc)

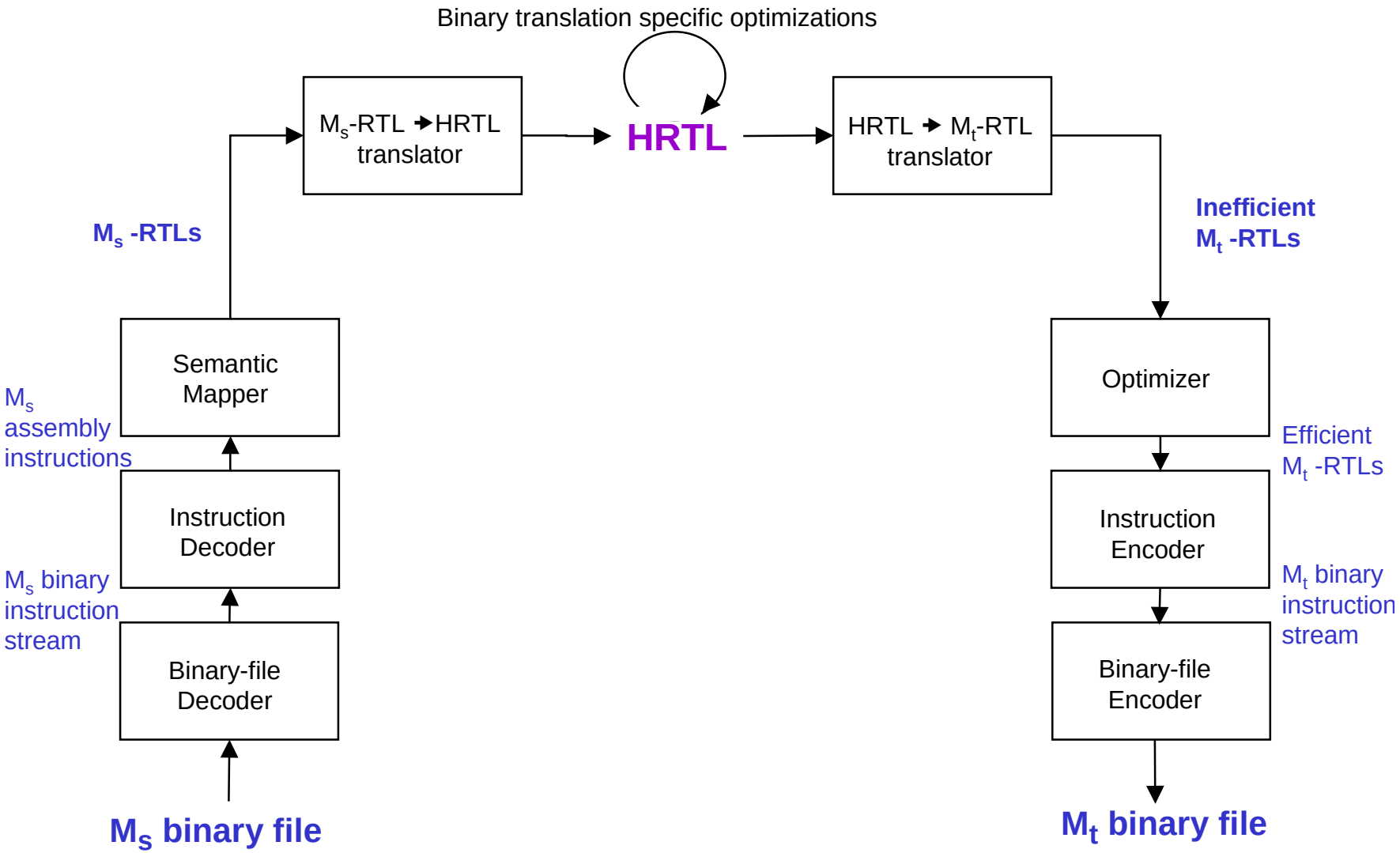
```
v0 := m[%afp + 100]
v0 := CALL fib (v0)
m[%afp + 96] = v0
```

HRTL (Pentium)

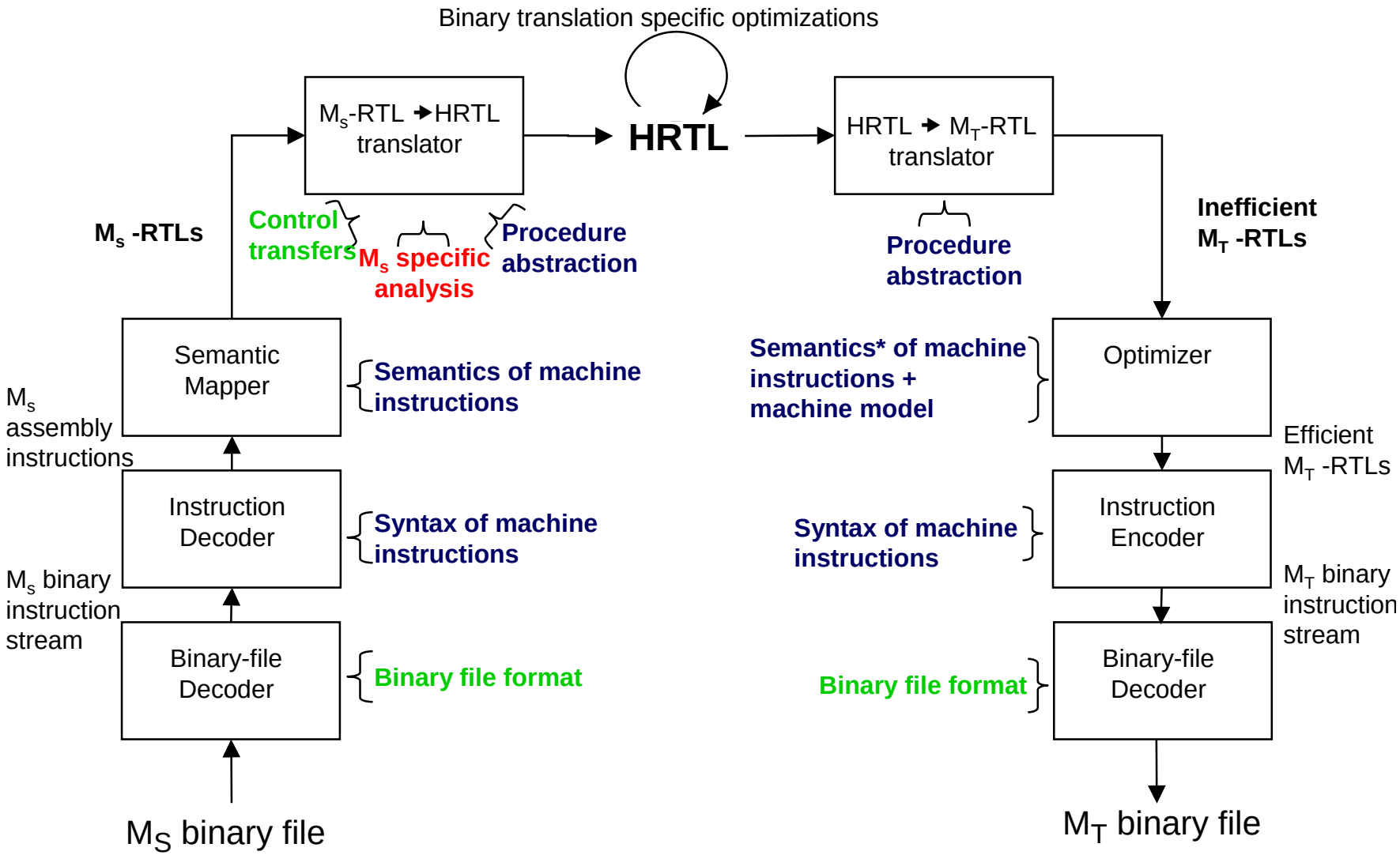
```
v3 := m[%afp + 4]
v4 := v3
v3 := CALL fib (v4)
m[%afp] := v3
```

Retargetability within UQBT

Binary translation process



Retargetable binary translation



Description languages and APIs

- Machine-oriented
  - SLED (specification language for encoding and decoding)
  - SSL (semantic specification language)
  - CTL (control transfer API)
- OS-oriented
  - BFF (binary-file format API)
  - PAL (procedural abstraction language)

Abstracting away from machine details (Ms-RTL to HRTL)

- Machine independent analyses
  - HL control transfer instructions
  - Recovery of actual and formal parameters
  - Recovery of function return values
  - Removal of stack frame and stack pointer (%afp)

- Machine dependent analyses
  - Sparc: removal of delayed branches
  - x86: removal of stack-based nature of float code
  - mc68k: removal of data pointer (%agp)
  - …

Calling conventions - Sparc architecture ABI

```
first ()
{  ...
   second();
   ...
}
```

Function prologue

```
second:
   save  %sp,-80,%sp
```

Function epilogue

```
second:
   jmpl  %i7+8,%g0
   restore  %l4,0,%o0
```

Pgs 3-13 and 3-20, Sparc architecture ABI supplement

Recovery of calling conventions with PAL

```
# Prologues and epilogues for callees and callers
CALLER_PROLOGUE std_call addr IS
    call__ (addr)
CALLEE_PROLOGUE new_reg_win simm13 IS
    SAVE (%sp, imode(simm13), %sp)
CALLEE_EPILOGUE std_ret IS
    ret();
    restore_()
```

Parameter passing - Sparc architecture ABI

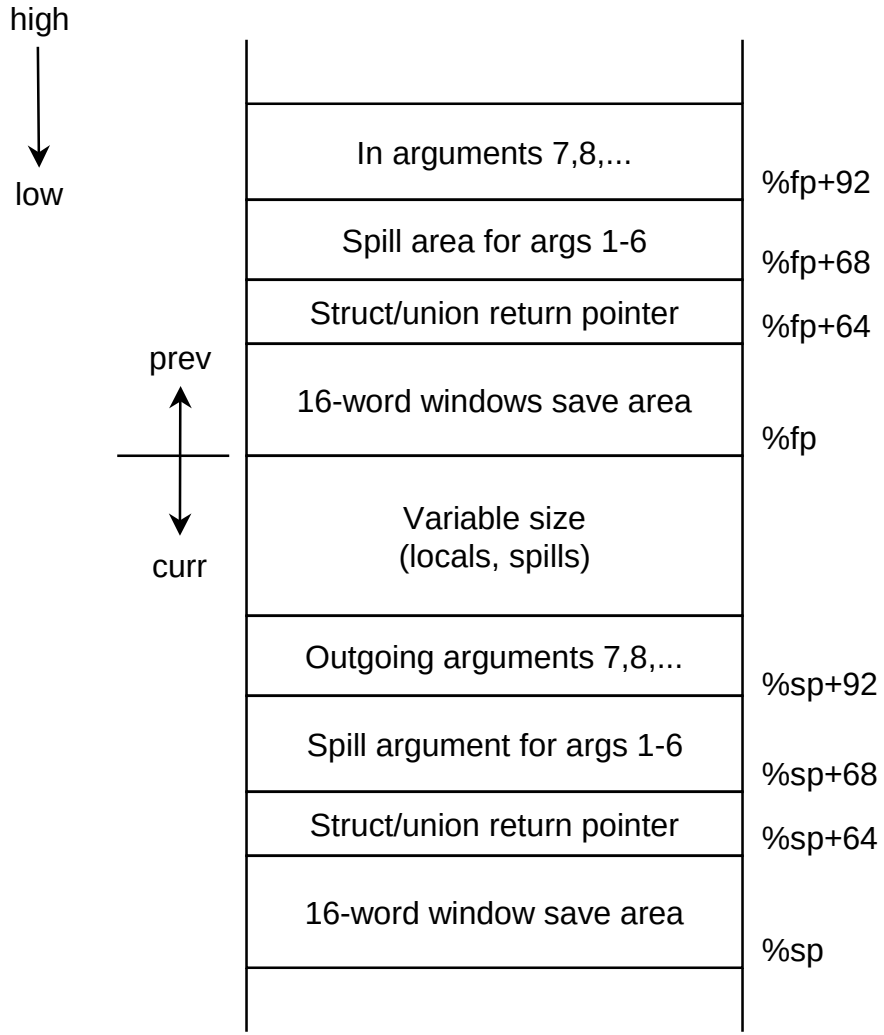
Call	Argument	Caller	Callee
	1	%o0	%i0
	2	%o1	%i1
	3	%o2	%i2
g(1,2,3,	4	%o3	%i3
4,5,6,7,	5	%o4	%i4
(void*)0);	6	%o5	%i5
	7	%sp+92	%fp+92
	(void*)0	%sp+96	%fp+96

Fig 3-19, Sparc architecture ABI supplement

Recovery of parameter passing locations with PAL

```
# Locations used for parameter passing
INCOMING PARAMETERS
    new_reg_win
    {
        AGGREGATE -> [%afp - simm13 + 64]
        REGISTERS -> %i0 %i1 %i2 %i3 %i4 %i5
        STACK     -> BASE = [%afp - simm13 + 92]
                     OFFSET = 4
    }

OUTGOING PARAMETERS
    AGGREGATE -> [%afp + 64]
    REGISTERS -> %o0 %o1 %o2 %o3 %o4 %o5
    STACK     -> BASE = [%afp + 92]
                 OFFSET = 4
```

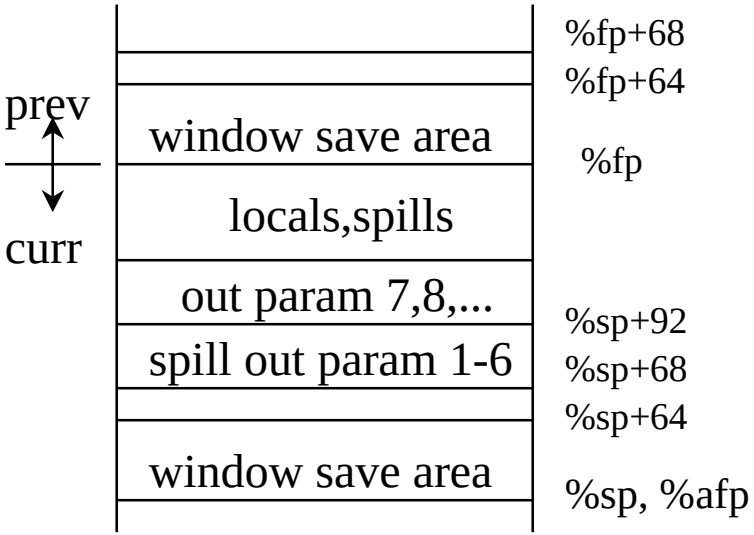
Stack frame - Sparc architecture ABI



Recovery of locals with PAL

- Abstract frame pointer (%afp)

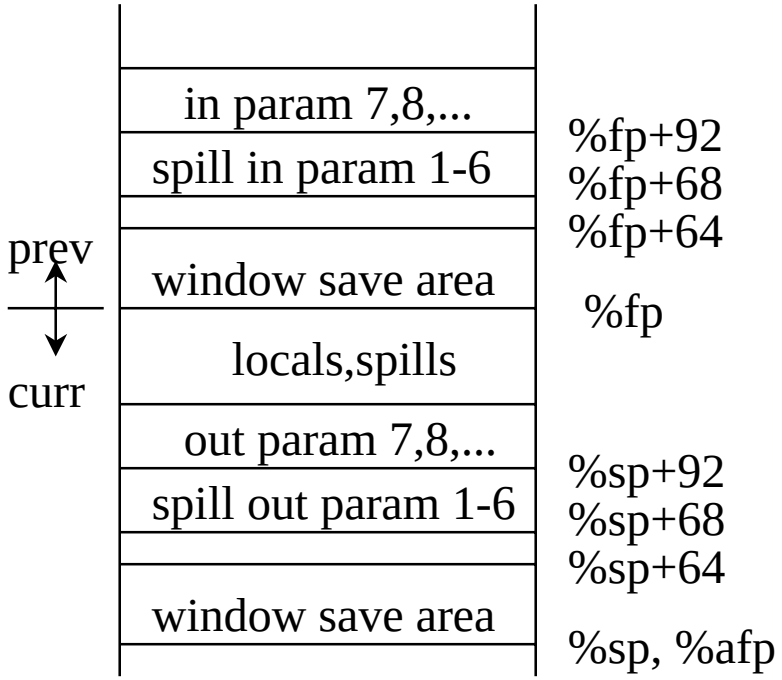
```
CALLEE_PROLOGUE new_reg_win N IS
   SAVE (%sp, imode(N), %sp)

FRAME ABSTRACTION
   INIT = %sp
   new_reg_win
   {
      %fp -> %afp - N
   }
```



Recovery of locals with PAL

- Access to parent frame via locals

```
PARENT STACK
  new_reg_win
  {
    %afp - N + 68 TO
    %afp - N + 88
    STEP 4
  }
```

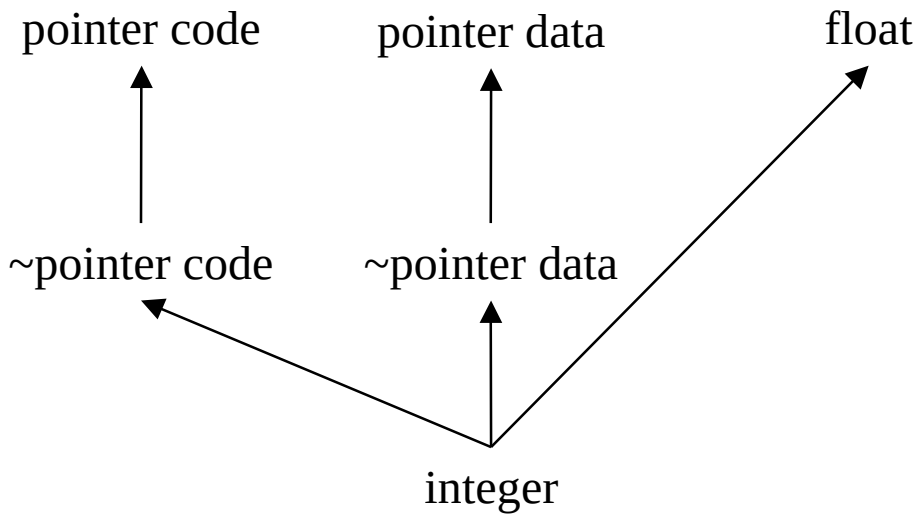


Recovery of parameters analysis

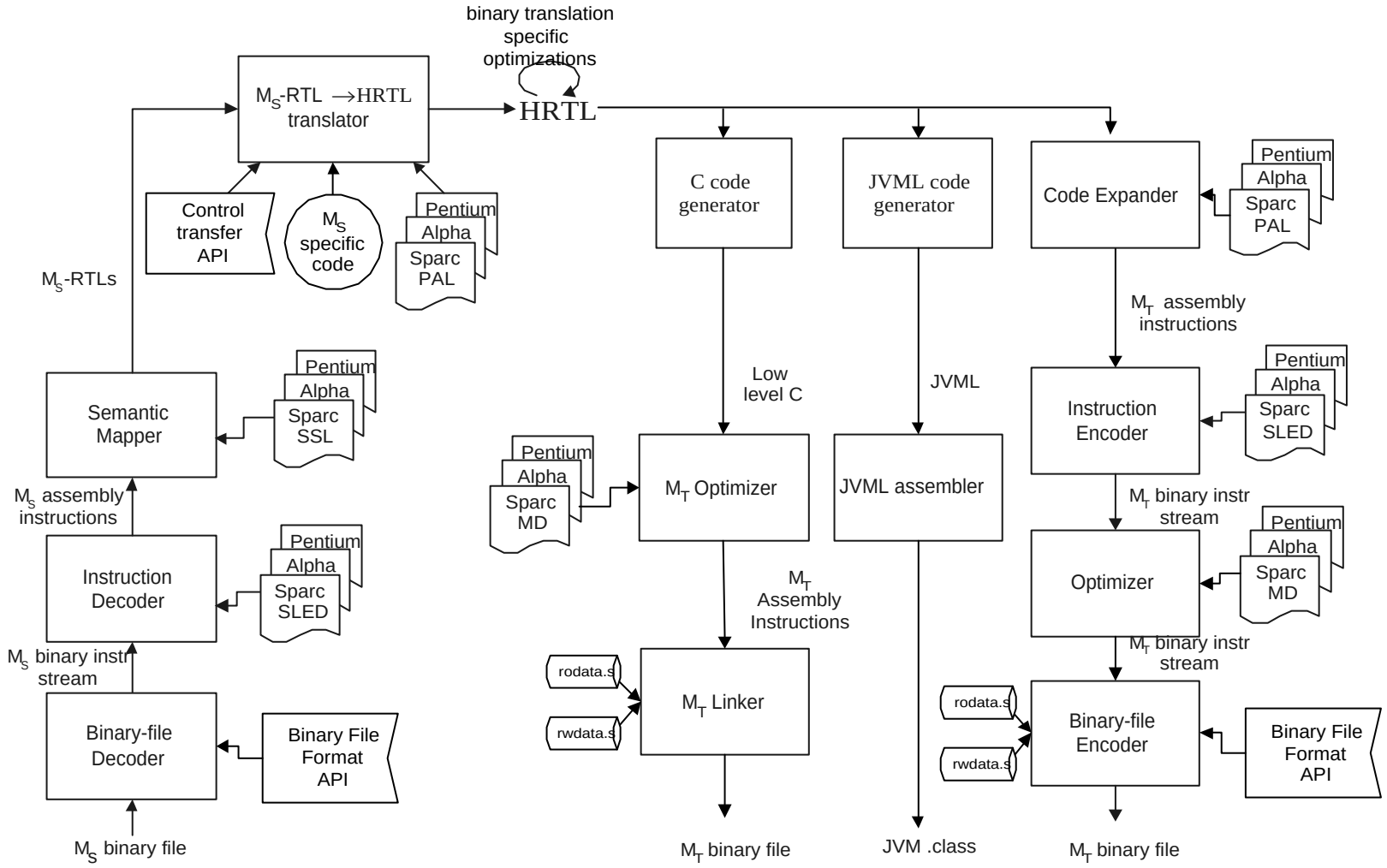
```
main:
    ...                          ReachingParamLocs = {%o0,%o1}
    ...
call gcd
    %o3 = %o0                    Use(%o0)
    ...
    return

gcd:                             LiveInParamLocs  = {%i0,%i1}
    ...
    return                       LiveOutRetLocs  = {%i0}
```

Recovery of parameters and return values

- Type recovery
- Lattice of low-level types:
  - integer
  - pointer to data
  - pointer to code
  - float
- Attributes
  - size
  - sign



The new UQBT framework



Example

Fibonacci C source

```
int fib (int x)
{
  if (x > 1)
      return (fib(x - 1) + fib(x - 2));
  else return (x);
}
```

Fibonacci – Pentium assembly

```
08048798:       55                      push   %ebp
08048799:       89 e5                   mov    %esp,%ebp
0804879b:       53                      push   %ebx
0804879c:       83 7d 08 01             cmpl   $0x1,0x8(%ebp)
080487a0:       7e 2e                   jle    80487d0 <fib+0x38>
080487a2:       8b 45 08                mov    0x8(%ebp),%eax          # fib(x - 1)
080487a5:       48                      dec    %eax
080487a6:       50                      push   %eax
080487a7:       e8 ec ff ff ff          call   8048798 <fib>
080487ac:       83 c4 04                add    $0x4,%esp
080487af:       89 c3                   mov    %eax,%ebx
080487b1:       8b 45 08                mov    0x8(%ebp),%eax
080487b4:       83 c0 fe                add    $0xfffffffe,%eax
080487b7:       50                      push   %eax
080487b8:       e8 db ff ff ff          call   8048798 <fib>
080487bd:       83 c4 04                add    $0x4,%esp
080487c0:       89 c0                   mov    %eax,%eax
080487c2:       8d 14 18                lea    (%eax,%ebx,1),%edx
080487c5:       89 d0                   mov    %edx,%eax
080487c7:       eb 0f                   jmp    80487d8 <fib+0x40>
080487c9:       8d 76 00                lea    0x0(%esi),%esi
080487cc:       eb 0a                   jmp    80487d8 <fib+0x40>
080487ce:       8d 36                   lea    (%esi),%esi
080487d0:       8b 45 08                mov    0x8(%ebp),%eax
080487d3:       eb 03                   jmp    80487d8 <fib+0x40>
080487d5:       8d 76 00                lea    0x0(%esi),%esi
080487d8:       8b 5d fc                mov    0xfffffffc(%ebp),%ebx
080487db:       c9                      leave
080487dc:       c3                      ret
080487dd:       8d 76 00                lea    0x0(%esi),%esi
```

Fibonacci – Pentium-RTL code

```
08048798 *32* r[28] := r[28] - 4
         *32* m[r[28]] := r[29]
08048799 *32* r[29] := r[28]
0804879b *32* r[28] := r[28] - 4
         *32* m[r[28]] := r[29]
...
080487a0 *32* %pc := (((%NF ^ %OF) | %ZF) = 1) ? 80487d0 : %pc
080487a2 *32* r[24] := m[r[29] + 8]                                    # fib(x - 1)
080487a5 *32* r[tmp1] := r[24]
         *32* r[24] := r[24] - 1
          *1* %CF := (~(r[tmp1]@[31:31]) and (1@[31:31]))
                     or ((r[24]@[31:31]) and (~(r[tmp1]@[31:31]) or (1@[31:31])))
          *1* %OF := ((r[tmp1]@[31:31]) and ~(1@[31:31]) and ~(r[24]@[31:31]))
                     or (~(r[tmp1]@[31:31]) and (1@[31:31]) and (r[24]@[31:31]))
          %1* %NF := r[24]@[31:31]
          *1* %ZF := r[24] = 0 ? 1 : 0
080487a6 *32* r[28] := r[28] - 4
         *32* m[r[28]] := r[24]
080487a7 *32* r[28] := r[28] - 4
         *32* m[r[28]] := %pc
         *32* %pc := fib
080487ac *32* r[28] := r[28] + 4
080487af *32* r[27] := r[24]
080487b1 *32* r[24] := m[r[29] + 8]
...
080487d8 *32* r[27] := m[r[29] - 4]
080487db *32* r[28] := r[29]
         *32* r[29] := m[r[28]]
         *32* r[28] := r[28] + 4
080487dc *32* %pc := m[r[28]]
         *32* r[28] := r[28] + 4
```

Fibonacci – HRTL code

```
High level RTLs for procedure fib(v0)
Twoway BB (0x3f96a0):
0804879c *32* v4 := 1
         *32* v3 := v0
080487a0  JCOND (v3 <= v4) 80487d0

Call BB (0x3ee9f0):
080487a2 *32* v1 := v0                # fib(x - 1)
080487a5 *32* r[tmp1] := v1
         *32* v1 := v1 - 1
080487a6 *32* v2 := v1
080487a7  v1 := CALL fib(v2)

Call BB (0x3ed5f0):
080487af *32* r[27] := v1
080487b1 *32* v1 := v0
080487b4 *32* r[tmp1] := v1
         *32* v1 := v1 - 2
080487b7 *32* v2 := v1
080487b8  v1 := CALL fib(v2)

Oneway BB (0x3ec5f8):
080487c0 *32* v1 := v1
080487c2 *32* r[26] := v1 + (r[27] * 1)
080487c5 *32* v1 := r[26]
080487c7  JUMP 80487d8

L1: Oneway BB (0x3f7ab0):
080487d0 *32* v1 := v0
080487d3  JUMP 80487d8

L2: Ret BB (0x3ea400):
080487d8  RET     Return location is v1 (type int32)
```

Fibonacci – Low-level C code

```
#include "uqbt.h"
int32 fib(int32 v0) {
char _locals[4];
        int32 v1;  int32 v2;
        int32 v3;  int32 v4;
        int32 r26, r27;  int32 tmp1;

        /* 8048798 */
        v4=1;
        v3=v0;
        tmp1=(v0)-(1);
        if ((v3)<=(v4)) goto L1;
        v1=v0;
        tmp1=v1;
        v1=(v1)-(1);
        v2=v1;
        v1=fib(v2);
        r27=v1;
        v1=v0;
        tmp1=v1;
        v1=(v1)-(2);
        v2=v1;
        v1=fib(v2);
        v1=v1;
        r26=(v1)+( *(unsigned int32*)&r27*1);
        v1=r26;
        goto L2;
L1:             /* 80487d0 */
        v1=v0;
        goto L2;
L2:             /* 80487d8 */
        return v1;
}
```

Fibonacci – Low-level C code

```
#include "uqbt.h"
int32 main(int32 v0, int32 v1) {
char _locals[8];
        int32 v2;
        int32 v3;
        int32 v4;
        int32 v5;
        /* 80487e0 */

        v5=134514872;
        printf(v5);
        v2=(_locals)+(4);
        v5=v2;
        v4=134514887;
        scanf(v4,v5);
        v2= *((int32*)((_locals)+(4)));
        v5=v2;
        v2=fib(v5);
        v2=v2;
         *((int32*)(_locals))=v2;
        v2= *((int32*)(_locals));
        v5=v2;
        v2= *((int32*)((_locals)+(4)));
        v4=v2;
        v3=134514890;
        v2=printf(v3,v4,v5);
        v2=(v2)^(v2);
        goto L1;
L1:             /* 8048830 */
        return v2;
}
```

Fibonacci – Sparc assembly output

```
8051928:  9d e3 bf 88      save        %sp, -120, %sp
805192c:  80 a6 20 01      cmp         %i0, 1
8051930:  04 80 00 08      ble         0x8051950
8051934:  01 00 00 00      nop
8051938:  7f ff ff fc      call        fib
805193c:  90 06 3f ff      add         %i0, -1, %o0
8051940:  a0 10 00 08      mov         %o0, %l0
8051944:  7f ff ff f9      call        fib
8051948:  90 06 3f fe      add         %i0, -2, %o0
805194c:  b0 02 00 10      add         %o0, %l0, %i0
8051950:  81 c7 e0 08      ret
8051954:  81 e8 00 00      restore
```

Architectural differences – Pentium vs Sparc

- Instruction set
  - CISC vs RISC
- Register file
  - Sparc has register windows
  - x86 has overlapping registers (eg. al,ax,eax)
- Control transfers
  - Sparc has delayed branches
- Endianness
  - x86: little endian, Sparc: big endian
- Condition codes
  - x86 uses them extensively

Translators and Preliminary Results

Static translators instantiated from the UQBT framework

- uqbtss (sparc to sparc)
- uqbtsp (sparc to pentium)
- uqbtps (pentium to sparc)
- uqbtpp (pentium to pentium)

Experimental:

- uqbtsj (sparc to JVM bytecode)
- uqbtsm (sparc to majc)
- uqbtpa (mc68k to arm)
- uqbths (pa-risc to sparc)

Results (Sparc to Sparc)

	Generated Sparc Code		Native Sparc Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) O0	11.5	13.1	26.9	
Fibo(40) O4	11.4	13.2		14.9
Sieve(3000) O0	12.6	12.6	18.8	
Sieve(3000) O4	13.5	12.6		15.7
Mbanner(500K) O0	36.8	27.3	40.2	
Mbanner(500K) O4	19.5	18.6		14.6
Compress (14000000 e 2231)	214.0	183.0		158.0*
Go (50 21)	119.0	124.0		113.0*

Results on an idle Sun Ultra5, 400MHz w/2Mb cache, 512Mb RAM , Solaris 2.8
Source programs compiled with gcc 2.95.2.1 at O0 and O4 on a Sparc processor
Walkabout optimized code by gcc 2.95.2.1 and cc 5.0 on a Sparc processor
gcc-O4, ccf == cc –xarch=v8 –xchip=ultra –fast –xO4 -xdepend
Native code compiled with gcc 2.95.2.1 -O0 and -O4 on a Sparc processor,
* compiled with ccf options

Results (Pentium to Sparc)

	Generated Sparc Code		Native Sparc Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) O0	11.6	13.0	26.9	
Fibo(40) O4	11.6	13.0		14.9
Sieve(3000) O0	38.5	28.3	18.8	
Sieve(3000) O4	53.8	49.2		15.7
Sieve(3000) O0 (32b)	18.1	12.6	18.8	
Sieve(3000) O4 (32b)	17.6	12.6		15.7
Mbanner(500K) O0	190.0	124.0	40.2	
Mbanner(500K) O4	12.1	8.9		14.6

Source programs compiled with gcc 2.8.1 at O0 and O4 on a Pentium processor
Results on an idle Sun Ultra5, 400MHz w/2Mb cache, 512Mb RAM , Solaris 2.8
Walkabout optimized code by gcc 2.95.2.1 and cc 5.0 on a Sparc processor
gcc-O4, cc –xarch=v8 –xchip=ultra –fast –xO4 -xdepend
Native code compiled with gcc 2.95.2.1 -O0 and -O4 on a Sparc processor

Results (Pentium to Pentium)

	Generated Machine Code		Native Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) sec	25.8	24.5	28.6	25.9
bytes	16,496	7,268	16,144	16,152
Sieve(3000)	18.6	17.1	18.9	18.6
bytes	16,228	6,536	15,964	15,944
Mbanner(500K)	48.7	46.5	80.5	44.8
bytes	25,664	16,016	21,524	25,436

Results on a Pentium MMX, 250MHz w/128 RAM , Solaris 2.6 OS
Source programs compiled with gcc 2.8.1-O4 on a Pentium processor
Optimizer code by gcc 2.8.1 and cc 4.2 on a Pentium processor
Native code compiled with gcc 2.8.1 -O0 and -O4 on a Pentium processor

Results (Sparc to Pentium)

	Generated Machine Code		Native Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) sec	27.7	28.5	28.6	25.9
bytes	16,512	7,292	16,144	24,564
Sieve(3000)	17.8	17.4	18.9	18.6
bytes	16,244	6,548	15,964	15,944
Mbanner(500K)	42.5	n/a	80.5	44.8
bytes	22,240		21,524	25,436

Results on a Pentium MMX, 250MHz w/128 RAM , Solaris 2.6 OS
Source programs compiled with gcc 2.8.1-O4 on a Sparc processor
Optimizer code by gcc 2.8.1 and cc 4.2 on a Pentium processor
Native code compiled with gcc 2.8.1 -O0 and -O4 on a Pentium processor

Effort in writing a new translator

- Reuse specs or write new specs ~1.5 KLOC
- Decoding module (partially generated), integrate special patterns ~2 KLOC
- Reuse analysis and transformations ~24 KLOC
- Add any new optimizations

- Experienced: 3 months (mc68k)
- Novice: 6-9 months (PA-RISC)

Problems with static translation

- Find all code
- Self-modifying code
- Self-referential code
- Precise exceptions
- System code
- Volatile locations (threads)
- Time dependent code
- Transparency

Walkabout Project

Dynamic

- 90% of program execution time is spent in 10% of the code
- On-demand translation at runtime
  - Compile 10% of the code
  - Better cache usage is possible
  - Dynamic optimization opportunities
  - Overcomes limitations of static translation

Dynamic

- Virtual machine (VM)
- Interpreter/emulator
  - Traces
  - Hot/seed instruction
- Compile hot path
  - Simple code generation
  - Translation cache
- Optimization

Dynamic

- Interpretation vs naïve code generation
- Finding hot/seed instruction
  - Counters on edges until threshold is met
  - Hardware counters/sampling
- Finding hot path
  - Based on collected trace information
  - Next pass over the loop
  - Hw support?

Dynamic

- Types of optimizations
  - Code layout + branch removal
  - Constant propagation
  - Value specialization
  - Dead code elimination
  - Instruction scheduling
- Advantages
  - Across functions, library calls and system calls

Dynamic

- Translation cache management
  - Flush all
  - LRU
  - Generational collection

Problems with dynamic translation

- Initial startup overhead
- Performance of generated code
- Time dependent code
- Harder to debug

Future Areas of Impact

Areas of impact

- Reoptimize binaries
- Migrate legacy apps
- Support simpler hardware with backwards compatibility

Dynamic vs Static

Conclusions

- Machine-dependent vs machine-independent separation
- Machine and OS descriptions
- Good translation at low cost is feasible

More information

http://www.csee.uq.edu.au/csm/uqbt.html

cristina.cifuentes@sun.com