Automatic timing annotation of native software for MPSoC simulation

Frédéric Pétrot
Aimen Bouchhima
Patrice Gerin

SLS Group, TIMA Laboratory
Grenoble, France

Objective : Performance evaluation

Three criteria

- As early as possible in the design flow
- Decrease redesign cost
- As fast as possible
  - Enable effective exploration of architectural design space
- As accurate as possible
  - Reliable design decisions
A solution: Native execution platform

- SystemC based
- HW/SW interface models to support the native execution of embedded software
- Different abstraction levels of software
  - HAL
  - OS
  - COM
- Early, fast ... but
- What about accuracy?

Software annotation

- Insert annotations in embedded software code
  - Need to be
    - Accurate
    - Automatic
- Performance is affected by
  - Software itself (programmer, compiler, ISA)
  - Hardware (pipeline, cache, interconnect)
- The two aspects are orthogonal
  - We consider the software side of the problem
Software performance modeling: Example

- Embedded software execute in the context of SystemC processes
- Computation inside a SystemC process takes ZERO time
- Introduce wait() statement in source code
- Problem : Data dependency
  - Solution 1 : coarse grain instrumentation (WCET,...)
  - Solution 2 : finer instrumentation
  - Follow the control flow
  - How to automate?
Source level annotation

```c
for(i=0; i<x; i++)
{
    if(cond[i]>0)
    {
        result+=a[i]*c[i];
        wait(40);
    }
    else
    {
        result+=a[2*i+1];
        wait(25);
    }
    wait(15);
} 
Port = result;
```

The alternative: Binary level instrumentation

```c
for(i=0; i<x; i++)
{
    if(var>0)
    {
        tab[i]=i*coef[i];
    }
    else
    {
        tab[2*i+1] = y[i];
    }
}
gcc
wait(7)
wait(3)
wait(5)
wait(24)
wait(56)
wait(31)
wait(22)
```

Impossible with optimization turned on
Case of Optimizations

Compiler (gcc) is structured into 2 parts

- A front-end: Processor independent
  - Depends on the HLL used:
    - C, C++, ObjectiveC, JAVA
  - Transforms HLL into intermediate representation
  - Processor independent optimizations
- A back-end: Processor specific
  - One backend of each processor
    - X86, ARM, MIPS, Sparc...
  - Transforms IR into machine code
  - Processor specific optimizations

Problem diagnostic
Cross-annotation technique

 GCC is more than 1 million code line
 Use gcc as front-end
   LLVM-GCC
   Just add a LLVM back-end to GCC
 Use the LLVM infrastructure as a back-end
   native back-end
   X back-end
 Problem delayed?

Proposed solution

 Don’t try to find the relation between native and target CFG
 Target CFG as entry to the native back-end
 After being instrumented
   Basic bloc instrumentation pass
   Result : instrumented bytecode
   Debugging still on original source code
 The instrumented target CFG is « compiled » to the native processor
Practically

1. `llvm-gcc -emit-llvm -O2 prog.c -c -o prog.bc`
2. `llc -f -perf -march=X prog.bc`
3. `llc -f prog_inst.bc`
4. `gcc -shared -O2 prog_inst.s -o prog.so`

Example of result...

```
Bubble:
  sub r13, r13, #8
  str r4, [r13, #4]
  str r14, [r13, #0]
  mov r0, #500
  ldr r1, .CPI4_0
  str r0, [r1, #0]
.BB4_1: @bb22.outer
  mov r1, #1
  ldr r2, .CPI4_1
  b .BB4_5 @bb22
...
```

```
Bubble:
  movl $11, (%esp)
  call wait
  subl $28, %esp
  movl %esi, 24(%esp)
  movl %edi, 20(%esp)
  movl %ebx, 16(%esp)
  movl $500, top
  xorl %esi, %esi
  movl %edi, %edi
  subl $500, %edi
  movl $1, %ebx
  movl $sortlist, %ebp
  jmp .BB4_5
...
```
Conclusion

- Fully automatic software instrumentation tool
- Used very early in the design flow
  - Can be combined with abstract architecture dynamic simulation of:
    - Cache
    - Pipeline
  - Speedup: 1000x compared to ISS