Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation

Aimen Bouchhima, Patrice Gerin and Frédéric Pétrot

System-Level Synthesis Group
TIMA Laboratory
46, Av Félix Viallet, 38031 Grenoble, France

january 21st 2009
Multi-Processors System-On-Chip

The Trends

Software-centric architectures
- Exploit parallelism at application task level
- Benefit from software flexibility

Multiple Processors per SW node
- Achieve easily usable computational power

Validation and debug
- System level architecture exploration: SW deployment, communication implementation

Focus of this work: Software Node

Hardware: The processor subsystem
Software: The layered software stack
## The Trends

**Software-centric architectures**
- Exploit parallelism at application task level
- Benefit from software flexibility

**Multiple Processors per SW node**
- Achieve easily usable computational power

---

<table>
<thead>
<tr>
<th>SW</th>
<th>SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-Communication</td>
<td></td>
</tr>
<tr>
<td>HW</td>
<td>HW</td>
</tr>
</tbody>
</table>
The Trends

Software-centric architectures
- Exploit parallelism at application task level
- Benefit from software flexibility

Multiple Processors per SW node
- Achieve easily usable computational power
The Trends

Software-centric architectures
- Exploit parallelism at application task level
- Benefit from software flexibility

Multiple Processors per SW node
- Achieve easily usable computational power

Overriding challenges
- Validation and debug
- System level architecture exploration: SW deployment, communication implementation

Focus of this work: Software Node

Hardware: The processor subsystem

Software: The layered software stack
The Trends

Software-centric architectures
- Exploit parallelism at application task level
- Benefit from software flexibility

Multiple Processors per SW node
- Achieve easily usable computational power

Overriding challenges
- Validation and debug
- System level architecture exploration: SW deployment, communication implementation
Introduction

Multi-Processors System-On-Chip

The Trends

Software-centric architectures
- Exploit parallelism at application task level
- Benefit from software flexibility

Multiple Processors per SW node
- Achieve easily usable computational power

Overriding challenges

- Validation and debug
- System level architecture exploration: SW deployment, communication implementation

Focus of this work: Software Node

- Hardware: The processor subsystem
**The Trends**

Software-centric architectures
- Exploit parallelism at application task level
- Benefit from software flexibility

Multiple Processors per SW node
- Achieve easily usable computational power

**Overriding challenges**

- Validation and debug
- System level architecture exploration: SW deployment, communication implementation

**Focus of this work: Software Node**

- Hardware: The processor subsystem
- Software: The layered software stack
### MPSOC Abstraction levels

#### Classical approaches

**Cycle Accurate co-simulation environment**
- Cross compiled embedded software
- Interpreted and executed by ISSs
- Accurate but slow

**TLM based co-simulation environment**
- Abstraction of the hardware in TLM
- Software still interpreted by ISSs

---

**Native HW/SW co-simulation approaches**

1. **By the host machine:**
   - i.e., the processor running the simulation

2. **On a simulation model of the hardware dependant part**

---

**High Level Application**

- HDS API
- Operating System
- Com libs
- C/Math libs

**Hardware Abstraction Layer**

- CPU0
  - $I$ $D$
- CPU1
  - $I$ $D$
- ITC
- Intra-Communication
- MEM
  - $I$ $D$
- CPU2
  - $I$ $D$
- DMA
- HW
Classical approaches

Cycle Accurate co-simulation environment
- Cross compiled embedded software
- Interpreted and executed by ISSs
- Accurate but slow

TLM based co-simulation environment
- Abstraction of the hardware in TLM
- Software still interpreted by ISSs

Native HW/SW co-simulation approaches

- Software is executed:
  1. By the host machine: *i.e.* the processor running the simulation

High Level Application

Natively executed by the host machine

Hardware Abstraction Layer

CPU0 $I$ $D$

CPU1 $I$ $D$

ITC

Intra-Communication

MEM

CPU2 $I$ $D$

DMA

HW
MPSOC Abstraction levels

Classical approaches

Cycle Accurate co-simulation environment
- Cross compiled embedded software
- Interpreted and executed by ISSs
- Accurate but slow

TLM based co-simulation environment
- Abstraction of the hardware in TLM
- Software still interpreted by ISSs

Native HW/SW co-simulation approaches

Software is executed:
1. By the host machine: *i.e.* the processor running the simulation
2. On a simulation model of the hardware dependant part

High Level Application

Natively executed by the host machine

Hardware Dependent Simulation Model:
- HAL layer
- Processor Subsystem
Introduction

MPSOC Abstraction levels

Classical approaches

- Cycle Accurate co-simulation environment
  - Cross compiled embedded software
  - Interpreted and executed by ISSs
  - Accurate but slow

TLM based co-simulation environment

- Abstraction of the hardware in TLM
- Software still interpreted by ISSs

Native HW/SW co-simulation approaches

- Software is executed:
  1. By the host machine:
     *i.e.* the processor running the simulation
  2. On a simulation model of the hardware dependant part

- Considerable speedup
- Functional validation of the whole system

High Level Application

Natively executed by the host machine

System

Hardware Dependent Simulation Model:

- HAL layer
- Processor Subsystem

Patrice Gerin (TIMA Laboratory)
Introduction

Problem definition

** Few or no timing information 
- Software executes atomically in zero time 
- Allows only functional validation 
- Annotations must be introduced in software code to enable time modeling 

** Performance of software depends on two orthogonal factors 
- The software itself depends on 
  - Sequence and type of executed instructions 
  - The executed control flow graph 
- The underlying hardware depends on 
  - Caches, access latencies, 
  - Other processors, ... 

- In this work we focus on the software source of dependency. 
- The hardware aspects have been addressed in previous works [1,2] 

Objectives & Contributions

Objectives: Bring native execution closer to target execution
- Provide information of the executed target instructions in native execution
- That reflects closely:
  - The execution flow on the target processor
  - The performance of the instruction execution on the target processor

Contributions: A compiler based annotation technique
- Specific to native simulation approaches
- Fully automated and accurate
Outline

1. Introduction
2. Basic Concepts
3. Proposed Approach
4. Experimentations
5. Conclusions and Perspectives
Execution time approach

- Follow the execution control flow of the target program
- Annotate at basic block level
Basic Concepts

Execution time approach

- Follow the execution control flow of the target program
- Annotate at basic block level

Basic concepts

- A software source code

```c
x = (y!=0) ? 23 : 1234567;
```
Basic Concepts And Challenges

Execution time approach
- Follow the execution control flow of the target program
- Annotate at basic block level

Basic concepts
1. A software source code
2. The target object CFG (ARM)

```
x = (y!=0) ? 23 : 1234567;
```

```
ARM

cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #-16]
b .L4
mov r3, #1228800
add r3, r3, #5760
add r3, r3, #7
str r3, [fp, #-16]
```
Basic Concepts

Basic Concepts And Challenges

Execution time approach
- Follow the execution control flow of the target program
- Annotate at basic block level

Basic concepts
1. A software source code
2. The target object CFG (ARM)
3. The host object CFG (x86)
   Not relevant for estimation, \( x86 \neq ARM \)

\[
x = (y\neq 0) \ ? 23 : 1234567;
\]

\[
\begin{align*}
\text{ARM} & : \\
\text{cmp r3, #0} & \\
\text{beq .L2} & \\
\text{mov r2, #23} & \\
\text{str r2, [fp, #-16]} & \\
\text{b .L4} & \\
\text{mov r3, #1228800} & \\
\text{add r3, r3, #5760} & \\
\text{add r3, r3, #7} & \\
\text{str r3, [fp, #-16]} & \\
\text{ARM} & \\
\end{align*}
\]

\[
\begin{align*}
\text{x86} & : \\
\text{testl %eax, %eax} & \\
\text{je .L2} & \\
\text{movl 23, x} & \\
\text{jmp .L4} & \\
\text{x86} & \\
\text{movl 1234567, x} & \\
\end{align*}
\]
Basic Concepts

Basic Concepts And Challenges

Execution time approach
- Follow the execution control flow of the target program
- Annotate at basic block level

Basic concepts
1. A software source code
2. The target object CFG (ARM)
3. The host object CFG (x86)
   - Not relevant for estimation, x86 ≠ ARM
     - Annotation function call inserted in each basic blocks
     - Function argument identifies a corresponding basic block in the target CFG

```
x = (y!=0) ? 23 : 1234567;
```

```
1
x86
movl $23, x
jmp   .L4
movl $1234567, x
testl %eax, %eax
je    .L2
2
ncmp r3, #0
beq ...  $1, (%esp)
call  annotate
movl  $1234567, x
movl  $1, (%esp)
call  annotate
testl %eax, %eax
je    .L2
```

```
ARM
```

```
1
```

```
2
```

```
3
```

Patrice Gerin (TIMA Laboratory)
ASP-DAC’09
**Execution time approach**

- Follow the execution control flow of the target program
- Annotate at basic block level

**Basic concepts**

1. A software source code
2. The target object CFG (ARM)
3. The host object CFG (x86) Not relevant for estimation, x86 ≠ ARM
   - Annotation function call inserted in each basic blocks
   - Function argument identifies a corresponding basic block in the target CFG
4. Assumes a one-to-one mapping between the two CFGs: generally not the case
Outline

1. Introduction
2. Basic Concepts
3. Proposed Approach
4. Experimentations
5. Conclusions and Perspectives
A compiler based cross annotation

Main idea: Use the compiler intermediate representation IR

1. Host independent (before the host processor back-end)
2. Independent from the high level language (C,C++,etc)
3. The IR already contains the CFG related informations

Cross IR concept

- Extend the IR throughout the back-end
- Keep track of processor specific CFG transformations
Proposed Approach

A compiler based cross annotation

Main idea: Use the compiler intermediate representation IR

1. Host independent (before the host processor back-end)
2. Independent from the high level language (C, C++, etc)
3. The IR already contains the CFG related informations

Cross IR concept

- Extend the IR throughout the back-end
- Keep track of processor specific CFG transformations
Proposed Approach

A compiler based cross annotation

Main idea: Use the compiler intermediate representation IR

1. Host independent (before the host processor back-end)
2. Independent from the high level language (C, C++, etc)
3. The IR already contains the CFG related informations

Cross IR concept

- Extend the IR throughout the back-end
- Keep track of processor specific CFG transformations
Proposed Approach

A compiler based cross annotation

Main idea: Use the compiler intermediate representation IR

1. Host independent (before the host processor back-end)
2. Independent from the high level language (C, C++, etc)
3. The IR already contains the CFG related informations

Cross IR concept

- Extend the IR throughout the back-end
- Keep track of processor specific CFG transformations
Proposed Approach

Cross IR Construction

Typical case of CFG transformation

1. A complex IR instruction *e.g.* Set On Condition
2. Converted in a diamond-like structure for target processor with no support of such instructions
3. The Cross IR is modified to reflect the same diamond-like structure

Native and Target CGF are isomorphic
Proposed Approach

Cross IR Construction

Typical case of CFG transformation

1. A complex IR instruction e.g. Set On Condition
2. Converted in a diamond-like structure for target processor with no support of such instructions
3. The Cross IR is modified to reflect the same diamond-like structure

Native and Target CGF are isomorphic
Typical case of CFG transformation

1. A complex IR instruction e.g. Set On Condition
2. Converted in a diamond-like structure for target processor with no support of such instructions
3. The Cross IR is modified to reflect the same diamond-like structure

Native and Target CFG are isomorphic
Proposed Approach

Cross IR Annotation

For each cross-IR basic blocks:
1. Analyze statically the corresponding target basic block i.e. number/type of instructions, estimated number of cycles
2. Store informations (memory, file,...) and identify the basic block
3. Annotation call insertion with basic block identifier as only one argument

```
movl  $23, x
jmp   .L4
movl  $1234567, x
testl %eax, %eax
je    .L2
cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #-16]
b .L4
mov r3, #1228800
add r3, r3, #5760
add r3, r3, #7
str r3, [fp, #-16]
```

Target CFG

```
cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #-16]
b .L4
mov r3, #1228800
add r3, r3, #5760
add r3, r3, #7
str r3, [fp, #-16]
```

Native CFG

```
testl %eax, %eax
je    .L2
movl $23, x
jmp   .L4
movl $1234567, x
```

Patrice Gerin (TIMA Laboratory)
For each cross-IR basic blocks:

1. Analyze statically the corresponding target basic block i.e. number/type of instructions, estimated number of cycles
2. Store informations (memory, file,...) and identify the basic block
3. Annotation call insertion with basic block identifier as only one argument

```assembly
movl $23, x
jmp .L4
movl $1234567, x
testl %eax, %eax
je .L2
cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #-16]
b .L4
mov r3, #1228800
add r3, r3, #5760
add r3, r3, #7
str r3, [fp, #-16]
```

Target CFG

Native CFG

```assembly
testl %eax, %eax
je .L2
movl $23, x
jmp .L4
movl $1234567, x
```
For each cross-IR basic blocks:

1. Analyze *statically* the corresponding target basic block
   *i.e.* number/type of instructions, estimated number of cycles

```
cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #-16]
b .L4
mov r3, #1228800
add r3, r3, #5760
add r3, r3, #7
str r3, [fp, #-16]
```

```
testl %eax, %eax
je .L2
cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #16]
```

```
movl  $23, x
jmp   .L4
movl  $1234567, x
testl %eax, %eax
je    .L2
```

```
cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #-16]
```

```
BB_ID 1
- 2 instructions
- 3 cycles
- 5.8 µJoule
- ...
```
Proposed Approach

Cross IR Annotation

For each cross-IR basic blocks:

1. Analyze *statically* the corresponding target basic block
   *i.e.* number/type of instructions, estimated number of cycles
2. Store informations (memory, file,...) and identify the basic block

Target CFG

```
cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #-16]
b .L4
mov r3, #1228800
add r3, r3, #5760
add r3, r3, #7
str r3, [fp, #-16]
```

Native CFG

```
testl %eax, %eax
je    .L2
cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #]
b .L4
mov r3, #1228800
add r3, r3, #5760
add r3, r3, #7
str r3, [fp, #-16]
```
Proposed Approach

Cross IR Annotation

For each cross-IR basic blocks:

1. **Analyze statically** the corresponding target basic block 
   *i.e. number/type of instructions, estimated number of cycles*

2. **Store informations** (memory, file,...) and identify the basic block

3. **Annotation call insertion** with basic block identifier as only one argument

```
movl $23, x
jmp   .L4
movl $1234567, x
movl $1, (%esp)
call  annotate
testl %eax, %eax
je    .L2

cmp r3, #0
beq .L2
mov r2, #23
str r2, [fp, #-16]
b .L4

mov r3, #1228800
add r3, r3, #5760
add r3, r3, #7
str r3, [fp, #-16]
```
For each cross-IR basic blocks:

1. **Analyze statically** the corresponding target basic block
   *i.e. number/type of instructions, estimated number of cycles*

2. Store informations (memory, file,...) and identify the basic block

3. Annotation call insertion with basic block identifier as only one argument

**Proposed Approach**

- **Analyze**
  - `cmp r3, #0`  
  - `beq .L2`
  - `mov r2, #23`  
  - `str r2, [fp, #-16]`
  - `b .L4`
  - `mov r3, #1228800`
  - `add r3, r3, #5760`
  - `add r3, r3, #7`
  - `str r3, [fp, #-16]`

- **Store**
  - `movl $2, (%esp)`
  - `call annotate`

- **Annotate**
  - `testl %eax, %eax`
  - `je .L2`
  - `cmp r3, #0`
  - `beq .L2`
  - `mov r2, #23`
  - `str r2, [fp, #-16]`

**Target CFG**

- `cmp r3, #0`
- `beq .L2`
- `mov r2, #23`
- `str r2, [fp, #-16]`
- `b .L4`
- `mov r3, #1228800`
- `add r3, r3, #5760`
- `add r3, r3, #7`
- `str r3, [fp, #-16]`

**Native CFG**

- `movl $1, (%esp)`
- `call annotate`
- `testl %eax, %eax`
- `je .L2`
- `movl $2, (%esp)`
- `call annotate`
- `movl $23, x`
- `jmp .L4`
- `movl $3, (%esp)`
- `call annotate`
- `movl $1234567, x`
The Low Level Virtual Machine is

- An open source compiler infrastructure
- An intermediate representation

Architecture organization

- middle-end: transformation and optimization
- front-end: a port of GCC to the LLVM ISA
- back-end: processor specific Machine-LLVM representation
The Low Level Virtual Machine is

- An open source compiler infrastructure
- An intermediate representation

Architecture organization

- middle-end: transformation and optimization
- front-end: a port of GCC to the LLVM ISA
- back-end: processor specific Machine-LLVM representation
The Low Level Virtual Machine is
- An open source compiler infrastructure
- An intermediate representation

Architecture organization
- middle-end: transformation and optimization
- front-end: a port of GCC to the LLVM ISA
- back-end: processor specific Machine-LLVM representation
Proposed Approach

LLVM back-end extension

LLVM CFG maintained during back-end
- Transformations in the target CFG are reflected to the LLVM CFG until the last pass.

Annotation pass
- Analysis and annotation take place at the end of the back-end

Output
- The annotated bytecode can be recompiled using the host machine back-end to obtain the native annotated code

Patrice Gerin  (TIMA Laboratory)  
ASPDAC'09  
January 21st 2009  
14 / 26
Proposed Approach

LLVM back-end extension

LLVM CFG maintained during back-end
- Transformations in the target CFG are reflected to the LLVM CFG until the last pass.

Annotation pass
- Analysis and annotation take place at the end of the back-end

Output
- The annotated bytecode can be recompiled using the host machine back-end to obtain the native annotated code
Proposed Approach

LLVM back-end extension

**LLVM CFG maintained during back-end**
- Transformations in the target CFG are reflected to the LLVM CFG until the last pass.

**Annotation pass**
- Analysis and annotation take place at the end of the back-end

**Output**
- The annotated bytecode can be recompiled using the host machine back-end to obtain the *native annotated code*
Proposed Approach

LLVM back-end extension

### LLVM CFG maintained during back-end
- Transformations in the target CFG are reflected to the LLVM CFG until the last pass.

### Annotation pass
- Analysis and annotation take place at the end of the back-end

### Output
- The annotated bytecode can be recompiled using the host machine back-end to obtain the native annotated code
Approach Limitations

Limitations

- Processor specific implementation in assembly language
  - Hand optimized performance critical algorithms
  - Compilers back-end *builtin* functions
- Binary object format libraries not handled by this approach
  - Code provided by third-party
  - Non *Open-Source* code

Possible solution

- Decompilation approaches
  - Convert target assembly into compiler IR
  - Annotate the obtained IR according to the target code
  - Generate host machine code
Outline

1 Introduction

2 Basic Concepts

3 Proposed Approach

4 Experimentations

5 Conclusions and Perspectives
Experimentations Context

Software part

- Application: Multithread version of Motion-JPEG
- Operating System: DNA OS, with SMP support and POSIX pthread library
- C library: Newlib

Hardware part

- Symmetric Multi-Processor architecture

High Level Application

- Operating System
- Com libs
- C/Math libs

Hardware Abstraction Layer

- HDS API

Intra-Communication

- CPU0
- CPU1
- CPU2
- ITC
- MEM
- DMA

HW

Patrice Gerin (TIMA Laboratory)
Software part

- Application: Multithread version of Motion-JPEG
- Operating System: DNA OS, with SMP support and POSIX pthread library
- C library: Newlib
### Software part

- **Application:** Multithread version of Motion-JPEG
- **Operating System:** DNA OS, with SMP support and POSIX pthread library
- **C library:** Newlib

### Experimentations Context

#### Motion-JPEG

**HDS API**

<table>
<thead>
<tr>
<th>DNA OS</th>
<th>POSIX pthread</th>
<th>C/Math libs</th>
</tr>
</thead>
</table>

**Hardware Abstraction Layer**

- **CPU0**
  - $I$ | $D$
- **CPU1**
  - $I$ | $D$
- **CPU2**
  - $I$ | $D$
- **ITC**
- **MEM**
- **DMA**
- **HW**

**Intra-Communication**

- **buffer.push**
- **synchronize**

```c
void annotate(uintptr_t id)
{
    buffer.push((basicblock_t*)id)
    if(buffer.full())
        synchronize();
}
```

**Basic block data base**

- **BB info**
  - Nb instructions
  - Nb load
  - Nb store
  - Nb cycles

---

Patrice Gerin (TIMA Laboratory)  
ASP-DAC’09  
January 21st 2009  
17 / 26
**Software part**

- **Application**: Multithread version of Motion-JPEG
- **Operating System**: DNA OS, with SMP support and POSIX pthread library
- **C library**: Newlib

**Diagram**

- **Motion-JPEG**
  - HDS API
  - DNA OS
  - POSIX pthread
  - Newlib

- **Hardware Abstraction Layer**
  - CPU0 $I$ $D$
  - CPU1 $I$ $D$
  - CPU2 $I$ $D$
  - MEM
  - DMA
  - ITC
  - HW

- **Intra-Communication**

- **Note**
  - *Motion-JPEG*
  - *DNA OS*
  - *POSIX*
  - * pthread*
  - *Newlib*
Experimentations Context

Software part
- Application: Multithread version of Motion-JPEG
- Operating System: DNA OS, with SMP support and POSIX pthread library
- C library: Newlib

Hardware part
- Symmetric Multi-Processor architecture
An MPSOC native co-simulation environment: Software part
Experimentations

Integration of the Proposed Technique in a Simulation Flow

An MPSOC native co-simulation environment: Software part

- Hardware independent part of the software is annotated using `llvm-gcc`
  - For arm: `llvm-gcc -g -Zmllvm"-annotate=arm" -c main.c -o main.o`
  - For sparc: `llvm-gcc -g -Zmllvm"-annotate=sparc" -c main.c -o main.o`

[Diagram of hardware and software components]
Integration of the Proposed Technique in a Simulation Flow

An MPSOC native co-simulation environment: Software part

1. Hardware independent part of the software is annotated using \textit{llvm-gcc}
   - For arm: \texttt{llvm-gcc -g -Zmllvm -annotate=arm} -c main.c -o main.o
   - For sparc: \texttt{llvm-gcc -g -Zmllvm -annotate=sparc} -c main.c -o main.o

2. Build a dynamic library of the software parts containing:
   - Undefined \textit{annotate} function calls, automatically inserted during compilation
   - Basic blocks information directly stored in the library binary image
   - ID argument corresponds to a basic block information structure address

\begin{verbatim}
extern void annotate(uintptr_t id);
\end{verbatim}
An MPSOC *native co-simulation environment:* Hardware part

Experimentations
Integration of the Proposed Technique in a Simulation Flow

An MPSOC *native co-simulation environment:* Hardware part

```
extern void annotate(uintptr_t id);
```

Basic block data base

```
BB info
- Nb instructions
- Nb load
- Nb store
- Nb cycles
```
An MPSOC *native co-simulation environment*: Hardware part

- Processor Sub-System and HAL layer are modeled using SystemC
  - Allow validation of the OS and middleware implementation
  - Reflect low level details of a real architecture

### Diagram

```
Motion-JPEG
```

```
extern void annotate(uintptr_t id);
```

```
app.so
```

```
HDS API
```

```
DNA
```

```
POSIX
```

```
pthread
```

```
Newlib
```

```
Basic block data base
```

```
- Nb instructions
- Nb load
- Nb store
- Nb cycles
```

Hardware Dependent Simulation Model:
- HAL layer
- Processor Subsystem
Experimentations

Integration of the Proposed Technique in a Simulation Flow

An MPSOC native co-simulation environment: Hardware part

1. Processor Sub-System and HAL layer are modeled using SystemC
   - Allow validation of the OS and middleware implementation
   - Reflect low level details of a real architecture

2. The `annotate` function is implemented in the SystemC model
   - Called at each basic block execution
   - ID are buffered and computed only when needed to speed-up the simulation
   - Basic block information is computed to consume simulation time

```c
extern void annotate(uintptr_t id);

void annotate(uintptr_t id) {
    buffer.push((basicblock_t*)id);
    if (buffer.full())
        synchronize();
}

void synchronize() {
    ... for(i=0; i<BUF_SIZE; i++)
        time += buffer[i].nb_cycles;
    wait(time);
}
```

Hardware Dependent Simulation Model:
- HAL layer
- Processor Subsystem
Objective: Assess only the annotation accuracy

- Ability to reflect the CFG of the target software execution
- Should not take into account the underlying HW model
  ⇒ Use the number of instruction metric

Estimate the number of executed instructions

- On a relevant function:
  - Need a function with a large dynamicity
  - Variable Length Decoder (VLD) function of the jpeg decoder

Does not provide any performance estimation

- Number of instruction ≠ execution time
Experimentations

Experimentation Results

Number of executed instruction for each VLD function call

- Cycle accurate bit accurate (caba) provide the reference count
- Less than 3% of error due to not annotated code
- The SystemC model of the HAL software layer
- The error is negative or zero when the code is fully annotated

![Graph showing number of executed instructions and error percentage for VLD function calls](image)
Experimentation Results

Simulation Speed-up compared to CABA execution model

- Very dependent on:
  - Execution time computation
  - trace dump, software profiling, ...
  - The underlying HW model

- From $x100$ with timing estimation and execution time software profiling
- To $x1000$ speed-up factor with only execution time estimation.
Outline

1 Introduction

2 Basic Concepts

3 Proposed Approach

4 Experimentations

5 Conclusions and Perspectives
A compiled-based approach

- Automatic annotation of embedded software
- Accurate in term of program control flow execution
- The annotation process is clearly separated from the performance estimation
- Performance estimation depend on
  - Informations associated with the basic blocks
  - The underlying hardware architecture

Main benefits

- Adapted to high level hardware/software cosimulation approaches
- Not restricted to a particular compiler
## Improving analysis of basic blocks

- Increase accuracy
  - Pipeline effect
  - Instructions dependencies
  - *e.g.* WCET at a BB granularity
- Different information
  - Power consumption

## Tools are needed

- To interpret simulation results
- Annotation technique used to profile target software executed on the host machine
  "Cross profiling"
Conclusions and Perspectives

Perspectives & Future Work

Improving analysis of basic blocks

- Increase accuracy
  - Pipeline effect
  - Instructions dependencies
  - e.g. WCET at a BB granularity
- Different information
  - Power consumption

Tools are needed

- To interpret simulation results
- Annotation technique used to profile target software executed on the host machine “Cross profiling”
Conclusions and Perspectives

Perspectives & Future Work

Improving analysis of basic blocks

- Increase accuracy
  - Pipeline effect
  - Instructions dependencies
  - E.g. WCET at a BB granularity
- Different information
  - Power consumption

Tools are needed

- To interpret simulation results
- Annotation technique used to profile target software executed on the host machine "Cross profiling"
Questions

Patrice.Gerin@imag.fr

System-Level Synthesis Group
TIMA Laboratory
46, Av Félix Viallet, 38031 Grenoble, France