Native Simulation of Complex VLIW Instruction Sets Using Static Binary Translation and Hardware-Assisted Virtualization

Mian-Muhammad Hamayun, Frédéric Pétrot and Nicolas Fournel

System Level Synthesis Group,
TIMA Laboratory, CNRS/INP Grenoble/UJF,
46, Avenue Félix Viallet, F-38031 Grenoble, FRANCE

January 25th, 2013
Introduction and Motivation

Introduction – MPSoC Trends

Motivation

- Homogeneous vs. Heterogeneous Multi-Processor System-on-Chip
  - Many General Purpose Processors (GPPs) ⇒ System Level Parallelism
  - Specialized Processing Elements e.g. Digital Signal Processors (DSPs)
  - Very Long Instruction Word (VLIW) ⇒ Instruction Level Parallelism (ILP)

- MPSoC Complexity limits use of Analytical Methods for Design Space Exploration (DSE) and System Validation ⇒ Simulation Systems
Introduction – MPSoC Trends

Motivation

- Homogeneous vs. Heterogeneous Multi-Processor System-on-Chip
  - Many General Purpose Processors (GPPs) ⇒ System Level Parallelism
  - Specialized Processing Elements e.g. Digital Signal Processors (DSPs)
  - Very Long Instruction Word (VLIW) ⇒ Instruction Level Parallelism (ILP)

- MPSoC Complexity limits use of Analytical Methods for Design Space Exploration (DSE) and System Validation ⇒ Simulation Systems
Software Simulation Levels and Native Simulation

Software Simulation Levels

- Interpretation
  - Instruction Set Simulation (ISS).

- Native Simulation
  - Source Level Simulation (SLS).
  - Intermediate Representation Level Simulation (IRLS).

- Binary Level Simulation (BLS)
  - Dynamic Binary Translation (DBT)
  - Static Binary Translation (SBT)

What is Native Simulation?
When software is compiled/translated for host machine and does not require run-time translation or interpretation support. Native software accesses host machine resources (CPU, Memory, ...) directly or at-least has an illusion of direct access.
Software Simulation Levels and Native Simulation

<table>
<thead>
<tr>
<th>Software Simulation Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpretation</td>
</tr>
<tr>
<td>- Instruction Set Simulation (ISS).</td>
</tr>
<tr>
<td>Native Simulation</td>
</tr>
<tr>
<td>- Source Level Simulation (SLS).</td>
</tr>
<tr>
<td>- Intermediate Representation Level Simulation (IRLS).</td>
</tr>
<tr>
<td>Binary Level Simulation (BLS)</td>
</tr>
<tr>
<td>- Dynamic Binary Translation (DBT)</td>
</tr>
<tr>
<td>- Static Binary Translation (SBT)</td>
</tr>
</tbody>
</table>

What is Native Simulation?

- When software is compiled/translated for host machine and *does not* require run-time translation or interpretation support.
- Native software accesses host machine resources (CPU, Memory, ...) directly or at-least has an *illusion* of direct access.
Table of Contents

1 Introduction and Motivation

2 Problem Definition – Simulation of VLIW on Native Machines

3 Proposed Solution – Native Simulation of VLIW ISA using SBT and HAV

4 Experiments and Results

5 Summary and Conclusions
Software Execution in Virtual Machine

- Software executes in target address-space
  - Transparent memory accesses.
- Requires Host-Dependent HAL layer implementation e.g. x86.

Native Processor Wrapper

- Initializes and Runs VM(s) using KVM userspace library and forwards MMIO accesses to SystemC platform.
- Provides semi-hosting facilities e.g. annotations, profiling etc.

Like a Baremetal Machine

- Software executing in Guest Mode cannot see the Host operating system and libraries
  - No Dynamic Translations.
Native Simulation Platform and Compilation Flow – II

Traditional Compilation Flow

- Software is Compiled to IR using Compiler Front-end.
- Target-specific Backend optimizes the IR.
- An annotation pass annotates the Cross-IR ➔ *Equivalent CFG (Control Flow Graph)*

What can we do for VLIW Machines?

- Source Level Simulation?
  - sequential vs. parallel instructions.
- IR Level Simulation?
  - Requires a retargetable compiler e.g. LLVM.
  - Source code may not be available ➔ Binary Translation for Native Simulation?
  - Static translation is a better match i.e. Explicit ILP in VLIW.
  - Generated code could be optimized.
Native Simulation Platform and Compilation Flow – II

**Traditional Compilation Flow**

- Software is Compiled to IR using Compiler Front-end.
- Target-specific Backend optimizes the IR.
- An annotation pass annotates the Cross-IR — *Equivalent CFG (Control Flow Graph)*

**What can we do for VLIW Machines?**

- **Source Level Simulation?**
  - *sequential vs. parallel* instructions.

- **IR Level Simulation?**
  - Requires a *retargetable* compiler e.g. LLVM

- **Source code may not be available**
  - Binary Translation for Native Simulation?
    - Static translation is a better match *i.e.* Explicit ILP in VLIW.
    - Generated code could be optimized.
Superscalar vs. VLIW Processors

VLIW: A Simplified Superscalar
- No Reservation Stations or ROBs
- No Dynamic Scheduling.
  - Static Scheduling
  - Compile-Time ILP Specification

VLIW: Still A Complex Architecture!
- Parallel Instruction Execution (i.e. Execute Packets)
- Data Hazards.
- Complex Pipelines
  - Data + Control Hazards.
Superscalar vs. VLIW Processors

VLIW: A Simplified Superscalar

- No Reservation Stations or ROBs ⇒ No Dynamic Scheduling.
- Static Scheduling ⇒ Compile-Time ILP Specification.

VLIW: Still A Complex Architecture!

- Parallel Instruction Execution i.e. Execute Packets ⇒ Data Hazards.
- Complex Pipelines ⇒ Data + Control Hazards.
VLIW Processors Features and Translation Issues (TI C6x Series)

Key Features – VLIW Processors

- NOP Instructions
- Delay Slots ➔ Out-of-Order Completion.
- No Pipeline Flushing
- Instruction Fetch ➔ Instruction Execution.

Key Issues – Binary Translation

- Data Hazards (RAW, WAR, WAW).
- Control Hazards (Nested Branches).
- Early Termination e.g. Multi-Cycle NOPs.
- Side Effects i.e. Modification of Source Operands.
Address Translation + Indirect Branches

Data, Instruction and I/O Memory Accesses

- Exploit memory virtualization capabilities provided by VMM.
- Transfer I/O accesses to SystemC platform.

Indirect Branch Instructions

- No dynamic translation support ➞ Resort to static translation.
- Provide multi-level translations i.e. Basic Blocks and Execute Packets.

Hand-Written and Self-Modifying Code

- Branch Instructions targeting non-startup instructions in Execute Packets. Usually VLIW compilers do not produce such code ➞ Not Addressed.
- Presence of Pointers and Dynamic Linking. No very common in VLIW binaries ➞ Not Addressed.
Table of Contents

1 Introduction and Motivation

2 Problem Definition – Simulation of VLIW on Native Machines

3 Proposed Solution – Native Simulation of VLIW ISA using SBT and HAV

4 Experiments and Results

5 Summary and Conclusions
A Generic Approach Illustrated using LLVM

Translation Flow – RISC Machines

1. Target-specific instruction decoders.
2. RISC-specific Basic Block construction.
3. Target ISA functional specification in C.
4. Target-independent intermediate code generation.
5. Native compilation using native backend.

Translation Flow – VLIW Machines

Better suited to VLIW i.e. Minimal translation unit is an Execute Packet (Upto 8 Instructions).
A Generic Approach Illustrated using LLVM

Translation Flow – RISC Machines

1. Target-specific instruction decoders.
2. RISC-specific Basic Block construction.
3. Target ISA functional specification in C.
4. Target-independent intermediate code generation.
5. Native compilation using native backend.

Translation Flow – VLIW Machines

- Instruction decoders ➔ VLIW packet decoders.
- VLIW-specific Basic Block construction.
- Better suited to VLIW i.e. Minimal translation unit is an Execute Packet (Upto 8 Instructions).
VLIW Packet Decoding

- Decodes target instructions ➔ Generates in-memory Instruction Objects.
  - Each object contains target-specific details (Predicate, Operands Types, Values, Delay Slots etc.)
  - Each object can generate a Function Call in LLVM-IR (Architecture + Instruction + Operand Types)
- Extract Parallelism from instruction stream ➔ Execute Packets.
- Branch Analysis ➔ Mark statically known Branch Targets.

Basic Block Construction

- Start a New Basic Block for each statically known Branch Target.
- End at Branch Instruction + Execute Packets within its Delay Slot Range.
VLIW ISA Functional Specifications – I

**Target-specific Instruction Behavior in LLVM-IR**

- *When* and *How* to modify the Register, Memory or Control state of CPU.
- We require ISA behavior in LLVM-IR for *Composing* Intermediate Code.

**Target-specific Instruction Behavior in C**

- Defined in 'Simple' C and converted to LLVM-IR using LLVM Compiler Front-End.
- Multiple ISA behavior definitions *i.e.* Exhaustively representing *All* Operand Type combinations ➔ Simple and Easy to Generate.
### An ISA Example: MPYSU Instruction in 'C'

```c
// MPYSU - Multiply Signed 16 LSB and Unsigned 16 LSB.
ReturnStatus_t C62xMPYSU_SC5_UR16_SR32(C62x_DSPState_t * p_state, uint8_t is_cond,
                                        uint8_t be_zero, uint16_t idx_rc, uint32_t constant, uint16_t idx_rb,
                                        uint16_t idx_rd, uint8_t delay, C62x_Result_t * result){
    if(Check_Predicate(p_state, is_cond, be_zero, idx_rc)){
        int16_t ra = C6XSC5_TO_S16(constant);
        uint16_t rb = GET_LSB16(p_state->m_reg[idx_rb]);
        int32_t rd = ra * rb;
        SAVE_REG_RESULT(result, idx_rd, rd);
    }
    return OK;
}
```

### Key Elements

- **Naming Convention**: `C62xMPYSU_SC5_UR16_SR32(...)`
- **Behavior Specification**: `int32_t rd = ra * rb;`
- **Result on Parent’s Stack**: `C62x_Result_t * result` (Life time + Scope)
- **Return Value**: `OK` ➜ Instruction does not require special processing.
  ➜ Early Termination, Wait-for-Interrupt, Error Condition etc..
Intermediate Code Generation

IR Function Composition

- One Entry / Return Basic Block for each Target Basic Block.
- Core and Update Basic Block Pair for each Target Execute Packet.
- Control Flows between Generated IR Basic Blocks.
Intermediate Code Generation

IR Function Composition

- One Entry / Return Basic Block for each Target Basic Block.
- Core and Update Basic Block Pair for each Target Execute Packet.
- Control Flows between Generated IR Basic Blocks.

Core Basic Blocks

C1 Stack Memory Allocation
Instructions for ISA Results.
Intermediate Code Generation

IR Function Composition

- One Entry / Return Basic Block for each Target Basic Block.
- Core and Update Basic Block Pair for each Target Execute Packet.
- Control Flows between Generated IR Basic Blocks.

Core Basic Blocks

C1 Stack Memory Allocation
Instructions for ISA Results.

C2 Calls to ISA Behavior in LLVM-IR.
Intermediate Code Generation

**IR Function Composition**

- One Entry / Return Basic Block for each *Target* Basic Block.
- Core and Update Basic Block Pair for each *Target* Execute Packet.
- Control Flows between Generated IR Basic Blocks.

---

**Core Basic Blocks**

- **C1** Stack Memory Allocation
  Instructions for ISA Results.

- **C2** Calls to ISA Behavior in LLVM-IR.

- **C3** No Delay Slots ➔ Immediate Update
  Delay Slots ➔ Buffered Update;
  *Handles Data Hazards + Side-Effects*
Intermediate Code Generation

IR Function Composition

- One Entry / Return Basic Block for each Target Basic Block.
- Core and Update Basic Block Pair for each Target Execute Packet.
- Control Flows between Generated IR Basic Blocks.

Core Basic Blocks

C1 Stack Memory Allocation Instructions for ISA Results.
C2 Calls to ISA Behavior in LLVM-IR.
C3 No Delay Slots \( \Rightarrow \) Immediate Update Delay Slots \( \Rightarrow \) Buffered Update; Handles Data Hazards + Side-Effects
C4 Instructions for testing ISA Return Values e.g. Early Termination.
Intermediate Code Generation

IR Function Composition

- One Entry / Return Basic Block for each Target Basic Block.
- Core and Update Basic Block Pair for each Target Execute Packet.
- Control Flows between Generated IR Basic Blocks.

Core Basic Blocks

C1 Stack Memory Allocation Instructions for ISA Results.

C2 Calls to ISA Behavior in LLVM-IR.

C3 No Delay Slots ⟷ Immediate Update Delay Slots ⟷ Buffered Update; Handles Data Hazards + Side-Effects

C4 Instructions for testing ISA Return Values e.g. Early Termination.

Update Basic Blocks

U1 Update Processor State Registers including PC, Cycles and Buffered Results.
Intermediate Code Generation

IR Function Composition

- One Entry / Return Basic Block for each Target Basic Block.
- Core and Update Basic Block Pair for each Target Execute Packet.
- Control Flows between Generated IR Basic Blocks.

Core Basic Blocks

C1 Stack Memory Allocation
Instructions for ISA Results.

C2 Calls to ISA Behavior in LLVM-IR.

C3 No Delay Slots ➞ Immediate Update
Delay Slots ➞ Buffered Update;
Handles Data Hazards + Side-Effects

C4 Instructions for testing ISA Return Values e.g. Early Termination.

Update Basic Blocks

U1 Update Processor State
Registers including PC, Cycles and Buffered Results.

U2 If $RPC_T \neq 0$ ➞ Branch Taken;
Pass control to Software Kernel
Handles Nested Branches
Native Memory Accesses using Memory Virtualization

Extended Target Address Space

- All Memory Accesses are in Target Address Space; Thanks to Hardware-Assisted Memory Virtualization.
- Native Binary Size > VLIW Binary Size $\Rightarrow$ Extended Target Address Space.
Native Memory Accesses using Memory Virtualization

**Extended Target Address Space**

- All Memory Accesses are in Target Address Space; Thanks to Hardware-Assisted Memory Virtualization.
- Native Binary Size > VLIW Binary Size \(\Rightarrow\) Extended Target Address Space.

**Simulation Flow**

1. Initialize Platform.
Native Memory Accesses using Memory Virtualization

Extended Target Address Space

- All Memory Accesses are in Target Address Space; Thanks to Hardware-Assisted Memory Virtualization.
- Native Binary Size > VLIW Binary Size ⇒ Extended Target Address Space.

Simulation Flow

1. Initialize Platform.
2. Load Bootstrap Code + Native Binary.
Native Memory Accesses using Memory Virtualization

Extended Target Address Space
- All Memory Accesses are in Target Address Space; Thanks to Hardware-Assisted Memory Virtualization.
- Native Binary Size > VLIW Binary Size \( \Rightarrow \) Extended Target Address Space.

Simulation Flow
1. Initialize Platform.
2. Load Bootstrap Code + Native Binary.
3. Boot KVM CPUs.
Native Memory Accesses using Memory Virtualization

Extended Target Address Space

- All Memory Accesses are in Target Address Space; Thanks to Hardware-Assisted Memory Virtualization.
- Native Binary Size > VLIW Binary Size => Extended Target Address Space.

Simulation Flow

1. Initialize Platform.
2. Load Bootstrap Code + Native Binary.
3. Boot KVM CPUs.
4. Load Target VLIW Binary.
Native Memory Accesses using Memory Virtualization

Extended Target Address Space

- All Memory Accesses are in Target Address Space; Thanks to Hardware-Assisted Memory Virtualization.
- Native Binary Size $>$ VLIW Binary Size $\Rightarrow$ Extended Target Address Space.

Simulation Flow

1. Initialize Platform.
2. Load Bootstrap Code + Native Binary.
3. Boot KVM CPUs.
4. Load Target VLIW Binary.
5. Continue Simulation.
Proposed Solution – Native Simulation of VLIW ISA using SBT and HAV

Multiple Code Generation/Translation Levels

Which Translation Level and Why / Why Not?

- Each Execute Packet ➔ Slower Simulation (Switching in Software Kernel)
Multiple Code Generation/Translation Levels

Which Translation Level and Why / Why Not?

- Each Execute Packet ➔ Slower Simulation (Switching in Software Kernel)
- Basic Blocks Only ➔ Dynamic Translation Support? (Indirect Branches)
Multiple Code Generation/Translation Levels

Which Translation Level and Why / Why Not?

- Each Execute Packet ➔ Slower Simulation (Switching in Software Kernel)
- Basic Blocks Only ➔ Dynamic Translation Support? (Indirect Branches)
- Basic Blocks + Execute Packets ➔ Fast Simulation but Redundant Code.
## Multiple Code Generation/Translation Levels

### Code Generation Modes – Summary

<table>
<thead>
<tr>
<th>Generation Mode</th>
<th>Execute Packets</th>
<th>Basic Blocks</th>
<th>Hybrid (BB+EP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Speed</td>
<td>Slow</td>
<td>Medium</td>
<td>Fast</td>
</tr>
<tr>
<td>Simulator Size</td>
<td>Medium</td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td>Dynamic Translations</td>
<td>Not Required</td>
<td>Required</td>
<td>Not Required</td>
</tr>
<tr>
<td>H/W Synchronization</td>
<td>Per EP</td>
<td>Per EP/BB</td>
<td>Per EP/BB</td>
</tr>
<tr>
<td>Self Modifying Code Support</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
### Code Generation Modes – Summary

<table>
<thead>
<tr>
<th>Generation Mode</th>
<th>Execute Packets</th>
<th>Basic Blocks</th>
<th>Hybrid (BB+EP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Speed</td>
<td>Slow</td>
<td>Medium</td>
<td>Fast</td>
</tr>
<tr>
<td>Simulator Size</td>
<td>Medium</td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td>Dynamic Translations</td>
<td>Not Required</td>
<td>Required</td>
<td>Not Required</td>
</tr>
<tr>
<td>H/W Synchronization</td>
<td>Per EP</td>
<td>Per EP/BB</td>
<td>Per EP/BB</td>
</tr>
<tr>
<td>Self Modifying Code Support</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
## Multiple Code Generation/Translation Levels

### Code Generation Modes – Summary

<table>
<thead>
<tr>
<th>Generation Mode</th>
<th>Execute Packets</th>
<th>Basic Blocks</th>
<th>Hybrid (BB+EP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Speed</td>
<td>Slow</td>
<td>Medium</td>
<td>Fast</td>
</tr>
<tr>
<td>Simulator Size</td>
<td>Medium</td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td>Dynamic Translations</td>
<td>Not Required</td>
<td>Required</td>
<td>Not Required</td>
</tr>
<tr>
<td>H/W Synchronization</td>
<td>Per EP</td>
<td>Per EP/BB</td>
<td>Per EP/BB</td>
</tr>
<tr>
<td>Self Modifying Code Support</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Table of Contents

1 Introduction and Motivation

2 Problem Definition – Simulation of VLIW on Native Machines

3 Proposed Solution – Native Simulation of VLIW ISA using SBT and HAV

4 Experiments and Results

5 Summary and Conclusions
Experimental Setup and Benchmarks

**Test Kernels – Control and Compute Intensive**
- Fibonacci Index – Recursive
- Factorial Index – Recursive
- IDCT Block Decoding

**Benchmark / Reference Simulators**
- TI C6x Full Cycle Accurate simulator (*TI-C6x-FCA*)
- TI C6x Device Functional Cycle Accurate simulator (*TI-C6x-DFCA*)

**Modest Host Machine for Experimentation**
- Pentium(R) Dual-Core CPU E5300 (2.60 GHz, 2M Cache) + 2 GB RAM.
- Linux version 2.6.32-37 32-bit (SMP)
Experimental Results – Fibonacci Index

![Graph showing simulation time vs. Fibonacci Index for different methods]

- ExecutePacket (EP)
- Hybrid (BB+EP)
- Hybrid+LMaps+Opt
- Hybrid+Hash+Opt
- TI-C6x-DFCA
- TI-C6x-FCA

Simulation Time (Seconds) vs. Fibonacci Index

Mian-Muhammad Hamayun (TIMA Laboratory)
Experimental Results
Experimental Results – Summary

Average Speedups/Slowdowns of SBT-Based Simulation

<table>
<thead>
<tr>
<th>Application</th>
<th>C6x-FCA Speedup</th>
<th>C6x-DFCA Speedup</th>
<th>Native Slowdown</th>
<th>DirectHost Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci</td>
<td>159x</td>
<td>39x</td>
<td>90x</td>
<td>101x</td>
</tr>
<tr>
<td>Factorial</td>
<td>132x</td>
<td>33x</td>
<td>205x</td>
<td>220x</td>
</tr>
<tr>
<td>IDCT</td>
<td>129x</td>
<td>31x</td>
<td>133x</td>
<td>141x</td>
</tr>
</tbody>
</table>
Table of Contents

1. Introduction and Motivation

2. Problem Definition – Simulation of VLIW on Native Machines

3. Proposed Solution – Native Simulation of VLIW ISA using SBT and HAV

4. Experiments and Results

5. Summary and Conclusions
### Summary

- A flow for Static Translation of VLIW Binaries to Native Code.
- Functionally Identical to TI Simulators; Verified by Trace Comparison.
- Profits from LLVM Infrastructure Components ➔ Optimized Native Code.

### Limitation and Overhead

- Completely Static ➔ Does not support Basic Block only simulation.
- Hybrid Translation mode ➔ Redundancy in Translated Code.

### Future Directions

- Automatic Generation of VLIW Instruction Decoders and ISA Behavior.
- Performance Estimation of Complex Benchmark Applications.
- Reducing the VLIW Architecture Modeling Overheads in Translated Code.
Thanks for Your Attention!

Questions?