

# 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

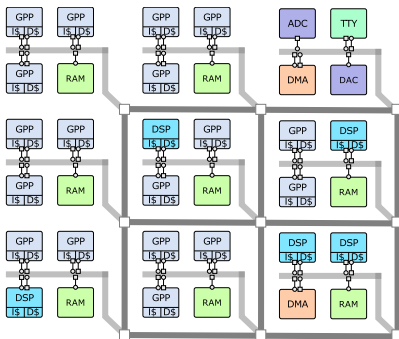
January 25<sup>th</sup>, 2013



# Introduction – MPSoC Trends

## Motivation

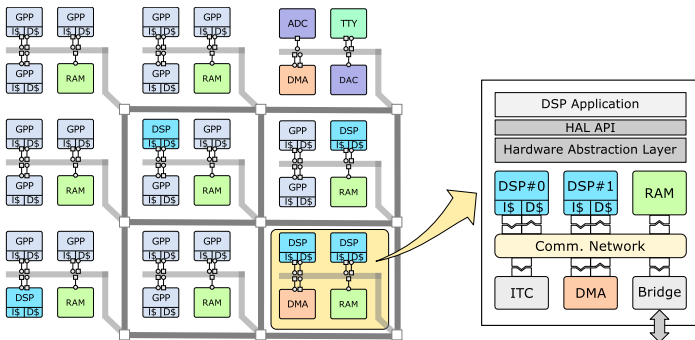
- Homogeneous vs. Heterogeneous Multi-Processor System-on-Chip
  - Many General Purpose Processors (GPPs)  $\Rightarrow$  System Level Parallelism
  - Specialized Processing Elements e.g. Digital Signal Processors (DSPs)
  - Very Long Instruction Word (VLIW)  $\Rightarrow$  Instruction Level Parallelism (ILP)
- MPSoC Complexity limits use of Analytical Methods for Design Space Exploration (DSE) and System Validation  $\Rightarrow$  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)

# 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.

# 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

# Native Simulation Platform and Compilation Flow – I

## Software Execution in Virtual Machine

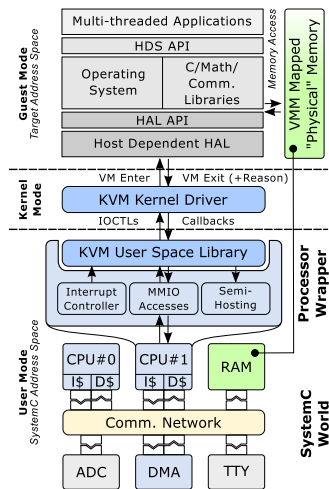
- Software executes in target address-space
  - ➡ Transparent memory accesses.
- Requires Host-Dependent HAL layer implementation e.g. x86.

## 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

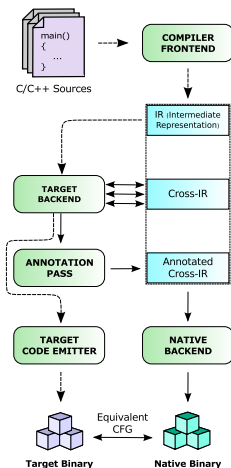
- Software executing in Guest Mode cannot see the Host operating system and libraries
  - ➡ No Dynamic Translations.



## 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)*





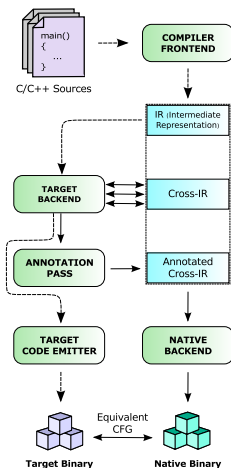
## Native Simulation Platform and Compilation Flow – II

## Traditional Compilation Flow

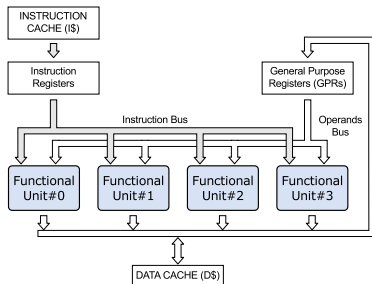
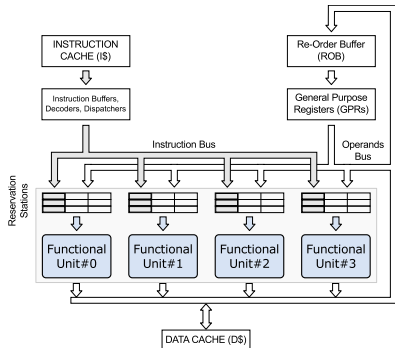
- Software is Compiled to IR using Compiler Front-end.
- Target-specific Backend optimizes the IR.
- An annotation pass annotates the Cross-IR
  - ➡ *Equivalent CFG (Control Flow Graph)*

## What can we do for VLIW Machines ?

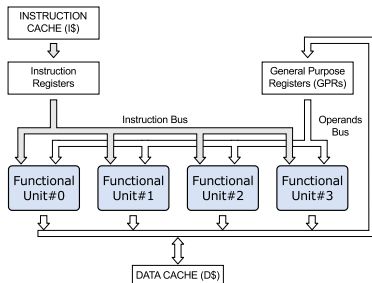
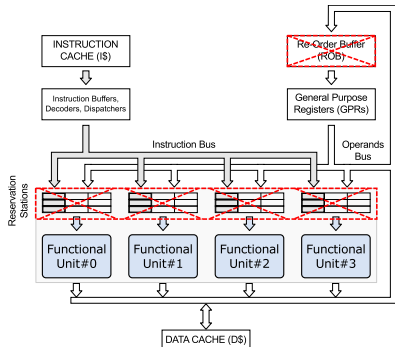
- Source Level Simulation?  
*sequential vs. parallel instructions.*
- IR Level Simulation?  
Requires a *retargetable* compiler e.g. LLVM
- Source code may not be available
  - ➡ Binary Translation for Native Simulation?
    - Static translation is a better match *i.e.* Explicit ILP in VLIW.
    - Generated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



# Superscalar vs. VLIW Processors



## Superscalar vs. VLIW Processors



## VLIW: A Simplified Superscalar

- No Reservation Stations or ROB's  
 ➡ 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.

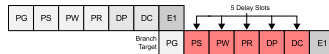
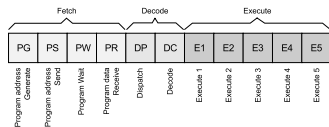
# VLIW Processors Features and Translation Issues (TI C6x Series)

## Key Features – VLIW Processors

- NOP Instructions
- Delay Slots  $\Rightarrow$  Out-of-Order Completion.
- No Pipeline Flushing  
Instruction Fetch  $\Rightarrow$  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.

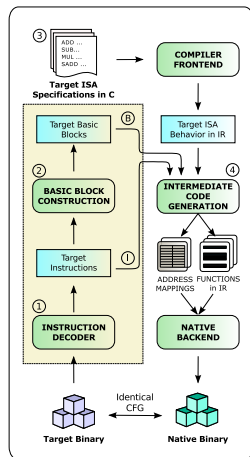
# Table of Contents

- 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.



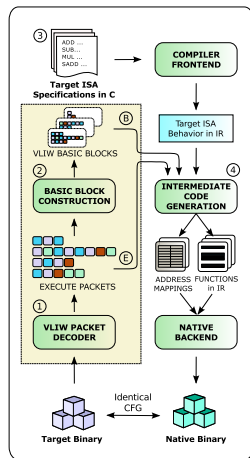
# A Generic Approach Illustrated using LLVM

## 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 Decoder + Target Basic Blocks

## VLIW Packet Decoding

- Decodes target instructions  $\Rightarrow$  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  $\Rightarrow$  *Execute Packets*.
- Branch Analysis  $\Rightarrow$  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  $\Rightarrow$  Simple and Easy to Generate.

## VLIW ISA Functional Specifications – II

## An ISA Example: MPYSU Instruction in 'C'

```

// MPYSU - Multiply Signed 16 LSB and Unsigned 16 LSB.      1
ReturnStatus_t                                             2
C62xMPYSU_SC5_UR16_SR32(C62x_DSPState_t * p_state, uint8_t is_cond,      3
    uint8_t be_zero, uint16_t idx_rc, uint32_t constant, uint16_t idx_rb,  4
    uint16_t idx_rd, uint8_t delay, C62x_Result_t * result){           5
    if(Check_Predicate(p_state, is_cond, be_zero, idx_rc))             6
    {                                                                    7
        int16_t  ra = C6XSC5_T0_S16(constant);                          8
        uint16_t rb = GET_LSB16(p_state->m_reg[idx_rb]);                9
        int32_t  rd = ra * rb;                                         10
                                                                    11
        SAVE_REG_RESULT(result, idx_rd, rd);                           12
    }                                                                    13
    return OK;                                                         14
}                                                                        15

```

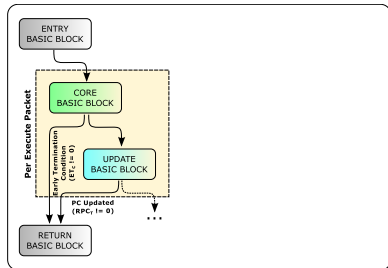
## Key Elements

- 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.



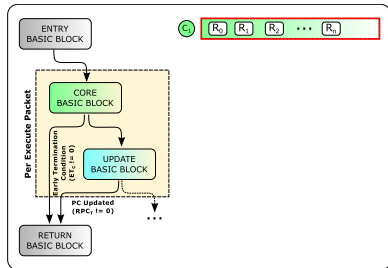
# Intermediate Code Generation

## 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.



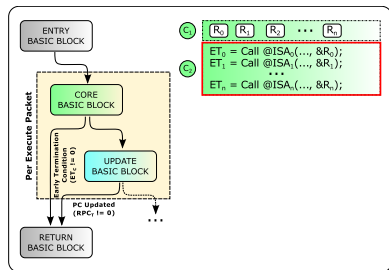
# Intermediate Code Generation

## 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.



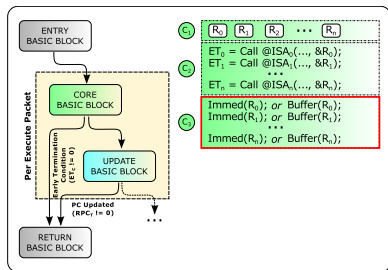
# Intermediate Code Generation

## 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  $\Rightarrow$  Immediate Update  
Delay Slots  $\Rightarrow$  Buffered Update;  
*Handles Data Hazards + Side-Effects*



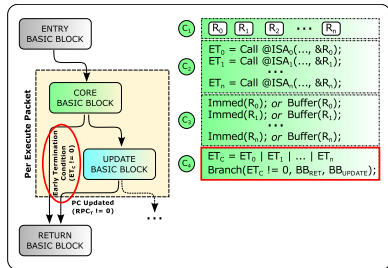
# Intermediate Code Generation

## 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.





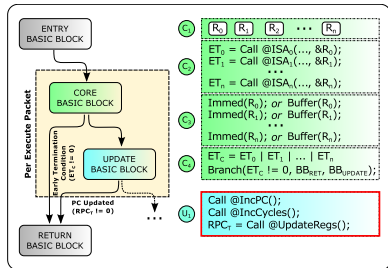
# Intermediate Code Generation

## 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  $\Rightarrow$  Immediate Update  
Delay Slots  $\Rightarrow$  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.

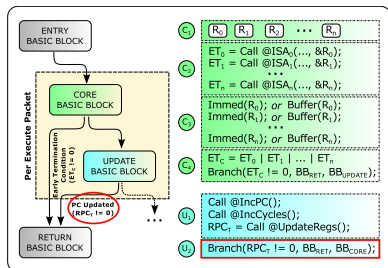
# Intermediate Code Generation

## 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  $\Rightarrow$  Immediate Update  
Delay Slots  $\Rightarrow$  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  $RPC_T \neq 0 \Rightarrow$  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.

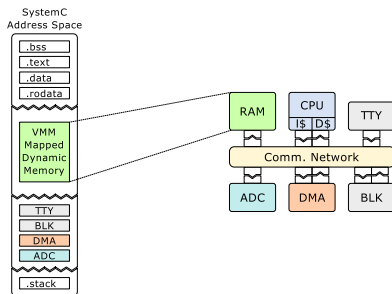
# Native Memory Accesses using Memory Virtualization

## 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.



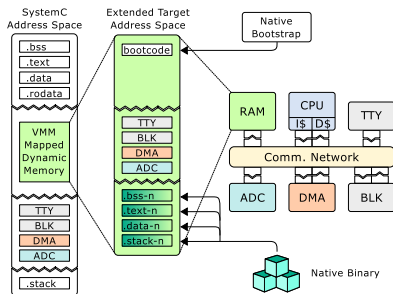
# Native Memory Accesses using Memory Virtualization

## Extended Target Address Space

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

## Simulation Flow

- 1 Initialize Platform.
- 2 Load Bootstrap Code + Native Binary.



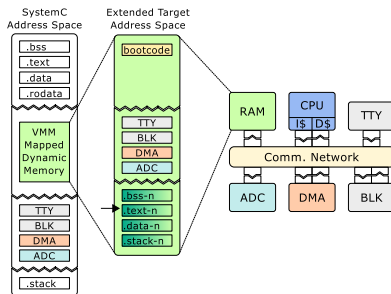
# Native Memory Accesses using Memory Virtualization

## 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.



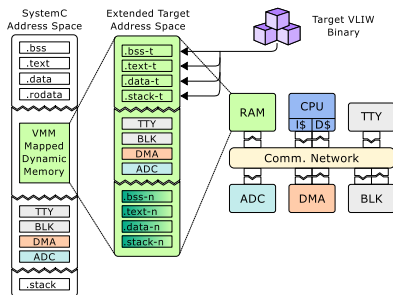
# Native Memory Accesses using Memory Virtualization

## Extended Target Address Space

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

## Simulation Flow

- 1 Initialize Platform.
- 2 Load Bootstrap Code + Native Binary.
- 3 Boot KVM CPUs.
- 4 Load Target VLIW Binary.



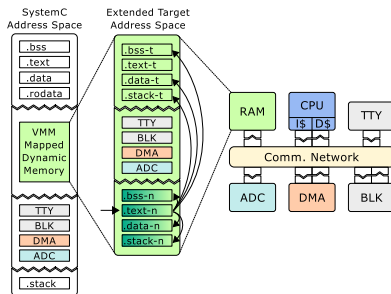
# Native Memory Accesses using Memory Virtualization

## 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.

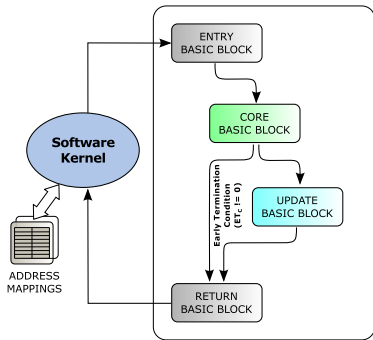




# Multiple Code Generation/Translation Levels

## Which Translation Level and Why / Why Not ?

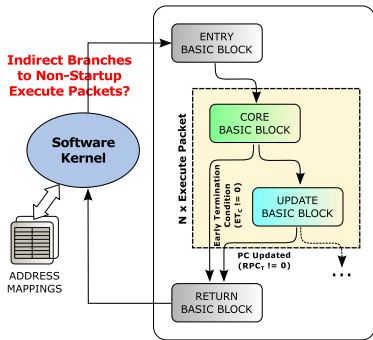
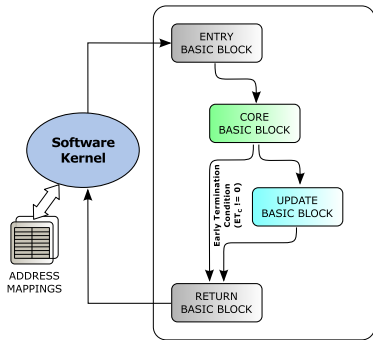
- Each Execute Packet  $\Rightarrow$  Slower Simulation (Switching in Software Kernel)



## Multiple Code Generation/Translation Levels

## Which Translation Level and Why / Why Not ?

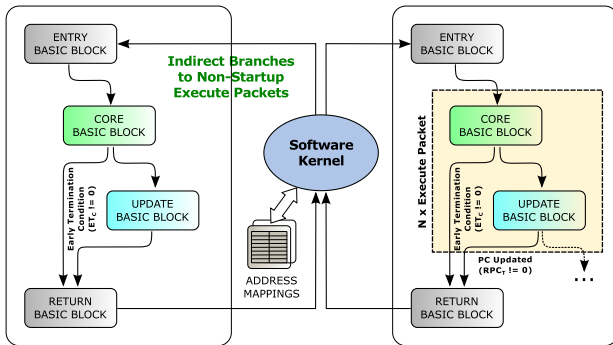
- Each Execute Packet  $\Rightarrow$  Slower Simulation (Switching in Software Kernel)
- Basic Blocks Only  $\Rightarrow$  Dynamic Translation Support ? (Indirect Branches)



# Multiple Code Generation/Translation Levels

## Which Translation Level and Why / Why Not ?

- Each Execute Packet  $\Rightarrow$  Slower Simulation (Switching in Software Kernel)
- Basic Blocks Only  $\Rightarrow$  Dynamic Translation Support ? (Indirect Branches)
- Basic Blocks + Execute Packets  $\Rightarrow$  Fast Simulation but Redundant Code.



## Multiple Code Generation/Translation Levels

## Code Generation Modes – Summary

Generation Mode	Execute Packets	Basic Blocks	Hybrid (BB+EP)
Simulation Speed	<b>Slow</b>	Medium	Fast
Simulator Size	<b>Medium</b>	Small	Large
Dynamic Translations	<b>Not Required</b>	Required	Not Required
H/W Synchronization	<b>Per EP</b>	Per EP/BB	Per EP/BB
Self Modifying Code Support	<b>No</b>	Yes	No

# Multiple Code Generation/Translation Levels

## Code Generation Modes – Summary

Generation Mode	Execute Packets	Basic Blocks	Hybrid (BB+EP)
Simulation Speed	Slow	<b>Medium</b>	Fast
Simulator Size	Medium	<b>Small</b>	Large
Dynamic Translations	Not Required	<b>Required</b>	Not Required
H/W Synchronization	Per EP	<b>Per EP/BB</b>	Per EP/BB
Self Modifying Code Support	No	<b>Yes</b>	No

# Multiple Code Generation/Translation Levels

## Code Generation Modes – Summary

Generation Mode	Execute Packets	Basic Blocks	Hybrid (BB+EP)
Simulation Speed	Slow	Medium	<b>Fast</b>
Simulator Size	Medium	Small	<b>Large</b>
Dynamic Translations	Not Required	Required	<b>Not Required</b>
H/W Synchronization	Per EP	Per EP/BB	<b>Per EP/BB</b>
Self Modifying Code Support	No	Yes	<b>No</b>

# 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

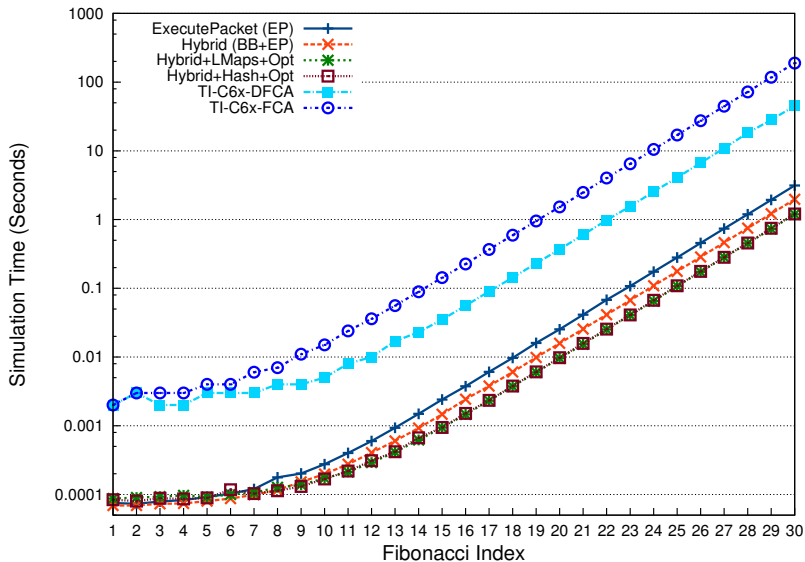
- TI C6x Full Cycle Accurate simulator (*TI-C6x-FCA*)
- TI C6x Device Functional Cycle Accurate simulator (*TI-C6x-DFCA*)

## Modest Host Machine for Experimentation

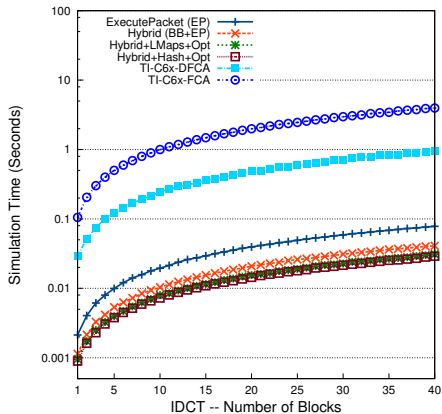
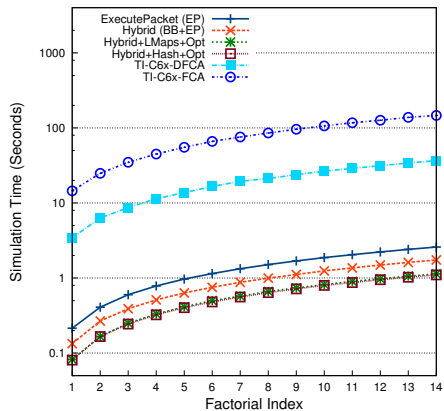
- Pentium(R) Dual-Core CPU E5300 (2.60 GHz, 2M Cache) + 2 GB RAM.
- Linux version 2.6.32-37 32-bit (SMP)



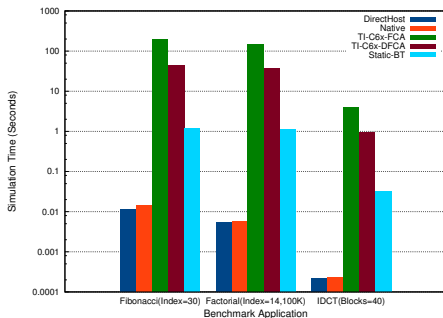
## Experimental Results – Fibonacci Index



# Experimental Results



# Experimental Results – Summary



## Average Speedups/Slowdowns of SBT-Based Simulation

Application	C6x-FCA Speedup	C6x-DFCA Speedup	Native Slowdown	DirectHost Slowdown
Fibonacci	159x	39x	90x	101x
Factorial	132x	33x	205x	220x
IDCT	129x	31x	133x	141x

# 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**

# Final Remarks

## 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 ?