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

Mian-Muhammad Hamayun, Frédéric Pétrot and Nicolas Fournel
TIMA Laboratory, CNRS/INP Grenoble/UJF
46 Avenue Félix Viallet, F-38031 Grenoble, France
firstname.lastname@imag.fr

Abstract— We introduce a static binary translation flow in native simulation context for cross-compiled VLIW executables. This approach is interesting in situations where either the source code is not available or the target platform is not supported by any retargetable compilation framework, which is usually the case for VLIW processors. The generated simulators execute on a Hardware-Assisted Virtualization (HAV) based native platform. We have implemented this approach for a TI C6x series processor and our simulation results show a speed-up of around two orders of magnitude compared to the cycle accurate simulators.

I. Introduction

Digital Signal Processors **(DSPs)** are one of the key type of processing units found in *Multiple Processors System-on-Chip* **(MPSoC)** architectures, along with *General Purpose Processors* **(GPPs)** and *Micro-Controller Units* **(MCUs)**. Integrating multiple and special-purpose processing elements helps in reducing overall cost of a given system design, while increasing the system complexity and design effort [5]. *Very Long Instruction Word* **(VLIW)** is a special class of processor architecture and many DSPs implement it. Simulators for these heterogeneous MPSoCs are needed for concurrent hardware/-software development, early availability, easy deployment and scalability studies.

Instruction Set Simulators **(ISSs)** are the most mature and commonly used processor virtualization technology. They execute the cross-compiled target binaries (Operating System + Applications) and mimic the behavior of target processors by using either instruction accurate interpretation [1] or *Dynamic Binary Translation* **(DBT)**. The former approach suffers from low simulation speed while the latter requires relatively high development cost. We refer readers to [11] for a detailed overview and comparison of software interpretation techniques. To achieve higher simulation speed, native simulation approaches [2, 8] have been proposed to overcome the interpretation/translation cost, by compiling the embedded software to the host binary format and executing it directly on the host processors. In order to access the modeled hardware resources, the software must use a specific API, either at operating system level [7] or hardware abstraction layer level [6].

Usually the functional part of native simulation schemes does not depend on the *Instruction Set Architecture* **(ISA)** of target processors, which makes them attractive from development point of view. On the other hand this makes the simulation subtly different from the target *Control Flow Graph* **(CFG)** perspective, as target specific optimizations cannot be easily applied on the native software. In the absence of VLIW architecture support in retargetable compilers, this becomes impossible and we have to consider binary translation for modeling such architectures.

II. Related Work

This work builds on *Hardware Assisted Virtualization* **(HAV)** based event-driven simulation platform [13] that uses natively compiled and un-modified software binaries to simulate a complete software stack, as shown in Fig. 1(a). This platform profits from the processor and memory virtualization capabilities provided by the *Virtual Machine Monitors* **(VMMs)** e.g. *Kernel Virtual Machine* **(KVM)**, and integrates it with the SystemC environment. The processor models interact with KVM User Space Library which interfaces with the KVM Kernel Driver using *I/O Controls* and *Callbacks*. The software executes in Guest Mode, using Target Address Space and exits to Kernel Mode for fulfilling I/O and Semi-hosting requests. The simulated memory is accessed *transparently* in Guest Mode and is shared between S/W and H/W models.

Native simulation platforms [6], primarily focus on source compiled and instrumented code for rendering faster simulations. But they incur serious limitations, due to the absence of complex architecture support in the retargetable compilation frameworks such as *Low Level Virtual Machine* **(LLVM)** [9], which does not support VLIW architectures, as of today. Examples of such architectures include C6000 series (Texas Instruments), SHARC processors (Analog Devices) and TriMedia DSPs (NXP Semiconductors). In native contexts such a compiler framework is employed for instrumenting the generated code, in order to accurately model the target processor.

One key feature of VLIW architecture is the compiler-based static scheduling of instructions i.e. the *Instruction Level Parallelism* **(ILP)** is provided by the compilation system and the processor executes them in exactly the same order. As the execution-order is known in advance, we can use a static translation scheme similar to [4], to translate, analyze and optimize the generated code. In addition we can avoid the decoding overhead at simulation run-time that can be quite significant for complex architectures.

Binary translation systems usually divide the translation process into multiple steps, introducing an *Intermediate Representation* **(IR)** to become resourceable and retargetable. The IR, such as *HRTL* in [4], is used to raise the abstraction level and is independent of both target and host architectures. The choice of IR becomes important, especially in cases where one intends to exploit some already available optimization facilities, such as provided by compilation frameworks. Static transla-

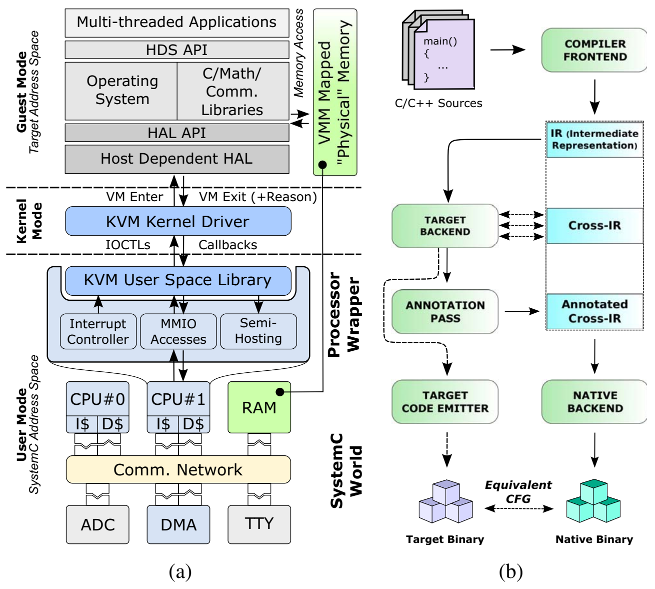


Fig. 1. (a) Native Simulation using Hardware-Assisted Virtualization and (b) Native Binary Generation Flow using Source Compilation.

tors have a few limitations, due to the lack of runtime information such as dynamic code flow, unknown register and memory contents, requiring some *fallback* mechanism at runtime. To cater for these problems many compiled and hybrid solutions have been proposed such as [3] and [12], but most of them are implemented for simple scalar architectures only and do not address system simulation. Few instances exist where simulation of complex processor architectures have been addressed as in [10, 14]. We address the simulation of VLIW processors by combining an HAV based native platform with retargetable compiler based static translation scheme. The primary interest of such a system would be the *Design Space Exploration* (**DSE**) of systems containing complex processors.

III. Problem Definition

Native simulation is a promising technology in early stages of DSE, given that a retargetable compilation framework that supports both target and host processors is available. Software is initially compiled for the target processor, instead of emitting target binary code, an annotation pass is introduced to gather and annotate the target-independent intermediate representation. This annotated IR is then compiled for the host machine using native compiler backend. Fig. 1(b), taken from [6] shows this compilation and annotation flow. Native binaries *reflect* the execution of target binaries, as they are considered to have an *equivalent* CFG. Software annotations provide target processor timings and are used to synchronize with SystemC component models.

In the absence of a retargetable compilation framework, native simulation degrades to functional validation, as target-specific annotations are not available. Moreover, in the absence of software sources, even functional validation becomes impossible using the traditional native techniques. As discussed in section II, VLIW architectures are usually not supported by retargetable compilers, instead vendor-specific toolchains are used to generate optimized binary code. This aspect further limits the generation of native binaries that have *equivalent* CFG, even if a VLIW specific backend is available.

Program 1 An Example VLIW Code (TI C6x)

```
    [!B1] ADD     B1,4,B3          // CPU Cycle          1
||        STW     B6,*B5++[1]                            2
||         MV     B3,A4                                  3
|| [B0]     B     0x010100         // 1                  4
        MPYSU     B6,A4,B6                               5
||       MPYU     A4,B4,A3         // 2                  6
          SUB     1,B0,B0                                7
||        SHR     B1,0x1,A3        // 3                  8
          NOP     5                // 4,5,6 [7,8]        9
          ADD     4,A10,B4         // 9                  10
```

Considering the limitations above, its interesting to see if generation of native code is possible using static translation principle. We concentrate on static translation, in order to avoid any run-time translation overheads and apply available optimizations during the code generation. Following sections highlight the problem from simulation platform, VLIW architecture and methodology point of views.

A. HAV Based Native Simulation Specifics

Native simulation requires that simulated software and hardware models share a uniform view of system memory. For example when software programs a DMA device to copy a software allocated buffer into a hardware device, consistency of addresses must be assured. The existing native simulation platform shown in Fig. 1(a), provides the capability to simulate software in target address space and access simulated memory transparently. In this virtualized memory context, run-time support from host operating system is not available and the software executes as if running on a baremetal machine, limiting the possibility of dynamic translations.

B. VLIW Processor Architecture Characteristics

We consider *Texas Instruments* (**TI**) C6x processors for our case study as they represent VLIW architecture and exhibit most problematic features found in such processors. On C6x processors, multiple *RISC-like* instructions are bundled together and execute in parallel in a single CPU cycle. These are known as *Execute Packets*, an can contain up-to eight instructions. In a given packet, a register can be the source and destination operand of multiple instructions simultaneously, resulting in *Write After Read* (**WAR**) data hazards. To avoid these hazards, we must preserve values of source operands during the execution of such packets. Program 1 shows an example of C6x instruction set, containing 10 instructions grouped into 5 execute packets. The || symbol in front of an instruction specifies parallelism, such as before instructions 2 through 4 indicating that these instructions belong to first packet and will be executed in parallel to instruction 1. Additionally, we can see a data hazard between instructions 1 and 3 for register *B3*. Most of the instructions consume one CPU cycle, nevertheless quite a few instructions require additional cycles i.e. *delay slots* as referred in TI's literature. VLIW compilers can schedule instructions within these delay slots resulting in *Write After Write* (**WAW**) data hazards. We must delay the writeback of destination operands in these multi-cycle instructions. Instruction 8 (*SHR*) uses register *A3* as destination, which is also a destination operand of instruction 6 (*MPYU*) but lies within its delay slot range, so the result of instruction 6 should be written *after* the execution of instruction 8. *Read After Write* (**RAW**) data hazards appear during compilation and VLIW compilers handle them by inserting NOP (No Operation) instructions. For proper simulation of VLIW binaries, these NOP instructions should also be *handled*, by advancing the pipeline stages of previously issued instructions.

In C6x processors, the execution pipeline is never flushed i.e. once an instruction enters the pipeline it will complete its execution. This feature dictates that a multi-cycle instruction issued before a branch, will finish its execution even after the branch has been taken. Multi-cycle NOP instructions or multi-nops are used to fill these delay slots. Multi-nop instructions could terminate earlier than their specified number of CPU cycles. Such situations usually arise when multi-nops are issued after branch instructions. Instruction 4 is a conditional branch, which *if taken* will force the *Early Termination* of instruction 9 after 6th CPU cycle, otherwise the multi-nop will complete its execution and instruction 10 will execute in the 9th CPU cycle.

Branch instructions can also be scheduled inside the delay slots of earlier branch instruction(s). It is also possible that all such branches are taken, in such cases, the control flow should finally reach the target address of last *taken* branch instruction. VLIW architectures also support different addressing modes to access data memories. C6x processors support Linear and Circular addressing modes for address calculation, depending on the contents of *AMR* register. Side effect updates of source registers are also possible during address calculation, in addition to the destination operand(s). Instruction 2 shows a side effect update for register *B5*. These types of instructions produce multiple outputs, usually with different delay slots.

C. Static Binary Translation Specifics

Address translation is one of the key problems of native simulation and previous studies have tried to merge the target and native address spaces [6] to create a unique and uniform address-space. But these solutions have some limitations, such as the need for dynamic linking in software binaries and modifications to the hardware platform components. Using static translation, we *cannot* transform target to native addresses, as there is no relationship between them, unless we exploit an address translation layer that could provide this support.

Static branch targets can be easily identified and analyzed during translation, as compared to the indirect branch instructions. There are two principle options for handling indirect branch instructions i.e. either provide dynamic translation support or do the static translation of all execute packets in addition to the higher level aggregations e.g. basic blocks. We will elaborate on this issue in Section IV(F). In certain cases, VLIW binaries can contain hand-written assembly code, with branch instructions targeting the middle of execute packets, for optimization or code-size reasons. Usually VLIW compilers *cannot* produce this type of binary code, so we do not handle this case in our case study. Self-Modifying Code is another issue, which usually emerges due to the presence of pointers and dynamic linking. Usually the VLIW binaries do not contain such code segments, as most of the functionality is processing intensive and instructions are rarely modified during execution, so we do not handle it.

IV. Proposed Solution

Our solution to the simulation for VLIW processors, relies on HAV based native platform as discussed in Section II. This platform provides the capability to access simulated memory in target address space, without requiring any changes to hardware models and software stack. We propose a binary translation flow for RISC and VLIW processors as shown in Figs. 2(a)

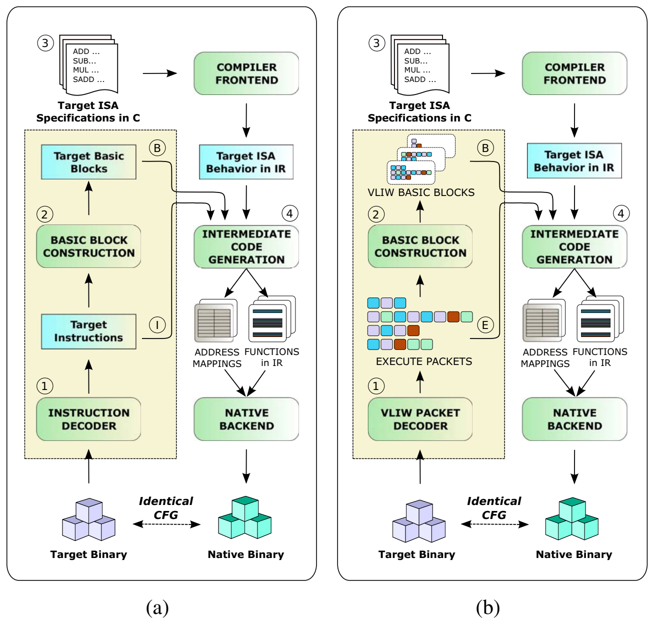


Fig. 2. (a) Static Translation of Cross-compiled RISC and (b) VLIW Binaries

and 2(b) respectively, which is used to generate native binaries from target binaries. This flow requires four key components, labeled as ①, ②, ③ and ④ in Figs. 2(a) and 2(b). We describe these components in detail below.

A. Instruction/VLIW Packet Decoder

The target-specific instruction decoder ①, decodes the instructions and their operands and creates in-memory objects for later use in translation. The VLIW Packet Decoder in Fig. 2(b), in addition to decoding also extracts parallelism from the instructions and creates objects representing Execute Packets. It also performs branch analysis by marking all of the statically known *branch target* instructions. This information is used as an additional criteria for basic block construction.

B. Basic Block Construction

Target basic blocks are formed by starting at either a statically known branch target instruction or when a previous target basic block must terminate due to the presence of a branch instruction. When branch instructions do not have delay slots as is usually the case in RISC machines, basic blocks are terminated immediately. Contrarily in VLIW machines a branch instruction normally requires some delay slots, so we have to include execute packets that lie within the these delay slots as they *will be* executed even-if the branch is *taken*. Besides basic block construction, this module also collects information about *direct* branch instructions, which is subsequently used for generating local address mappings as described in Section IV(E).

C. Target ISA Specifications

The target-ISA specifications ③ are defined in 'C' language and are used to generate ISA definitions in IR using the compiler front-end. Each target instruction has a specific behavior, that is *how* and *when* it modifies the register, memory or control state of the processor. Once the ISA is available in IR, it can be used to *compose* IR code for the target basic blocks. Each target instruction is represented with multiple definitions in IR, exhaustively representing all operand combinations. This strategy simplifies the ISA specification process, at the cost of specification size, which becomes irrelevant once specification generation process is automated.

D. Intermediate Code Generation

The intermediate code generation component ④ is based on LLVM framework, which provides classes and methods for generating functions, basic blocks, control flow and data processing instructions. Target specific algorithms are encapsulated in decoded instruction objects whereas the target independent algorithms are generic and ask the decoded objects for specific services, such as ISA call generation where a function call is placed in a given IR basic block by considering the number and operand types of the decoded instruction. This *separation of concerns* brings target independence to our design flow. Fig. 3(a) shows the structure of a generated function, containing four types of IR basic blocks, i.e. *Entry*, *Return*, *Core* and *Update* types. Each generated function contains one instance of Entry and Return basic blocks, whereas the Core and Update blocks depend on the number of execute packets in the target basic block, as highlighted in Fig. 3(a).

Each core basic block contains four groups of instructions, shown as C_1, C_2, C_3 and C_4. In group C_1, we generate memory allocation instructions for saving *Results* of ISA executions. The number of memory allocations depend on the size of an execute packet and are shown by R_0, R_1, ..., R_n in Fig. 3(a). These variables are allocated on the stack of generated function, thus limiting their visibility and lifetime to this function. Subsequently during code optimization, we can analyze these variables for their scope and remove unnecessary ones by algebraic simplifications. For group C_2, we generate the actual ISA calls and pass the addresses of corresponding stack variables. Each ISA computes and stores its result(s) in the callers stack memory variables. An ISA call also returns a status value shown as ET_0, ET_1, ..., ET_n, which if *non-zero* indicates an *Early Termination* request from the ISA call. In group C_3, we generate either *immediate* or *buffered* update calls for the ISA execution *Results*, by considering delay slots associated with the target instructions. This post update scheme brings *implicit* parallelism to the sequential instructions in IR and solves the data hazards issue, as was discussed in Section III (B). Some target instructions can also have side-effects i.e. modifications of registers other-than the destination operand(s), as is possible for load/store instructions in TI C6x processors. Group C_3 also handles such cases, by generating immediate update calls for side-effect results. Instructions in group C_4, evaluate the individual ISA return values to determine if *Early Termination* condition is set i.e. $ET_C \neq 0$. At run-time, the control flows to the *Return* basic block for non-zero ET_C values, otherwise it continues to the *Update* basic block.

Each *Update* basic block is composed of two groups i.e. U_1 and U_2. In group U_1, instructions for updating processors state are generated including general purpose registers, program counter (*PC*) and *CPU* cycles. The register updates deal with the buffered (delayed) update of registers, including the program counter for branch instructions issued earlier. In cases when the *PC* gets updated, it returns the new value of *PC*, which is shown as RPC_T (Returned Program Counter of Target) in group U_1. At run-time if $RPC_T \neq 0$, it indicates that a branch has been *taken* and control should be returned to the *software kernel*, which finds the next native function using the RPC_T value, and passes control to it. The $RPC_T = 0$ indicates that no branch was *taken* and the control flows to the next *Core* basic block, as shown in group U_2. This scheme handles nested branch instructions as discussed in Section III (B), because we check for RPC_T value after every *Update* basic block.

TABLE I
CODE GENERATION MODES

Generation Mode	Execute Packets	Basic Blocks	Hybrid (BB+EP)
Simulation Speed	Slow	Medium	Fast
Simulator Size	Medium	Small	Large
Dynamic Trans.	Not Required	Required	Not Required
Synchronization	Per EP	Per EP/BB	Per EP/BB
Self Modifying Code	No	Yes	No

E. Data and Instruction Memory Accesses

The native simulator generated by the proposed static translation flow accesses data memory using *exactly* the same addresses as in the original target binary. As the size of generated simulator is *much* larger than the original target binary, it cannot *fit* in the same memory regions as the target binary. We use the notion of *Extended Target Address Space*, where we load both the original and translated binaries in the simulated memory, as illustrated in Fig. 3(c).

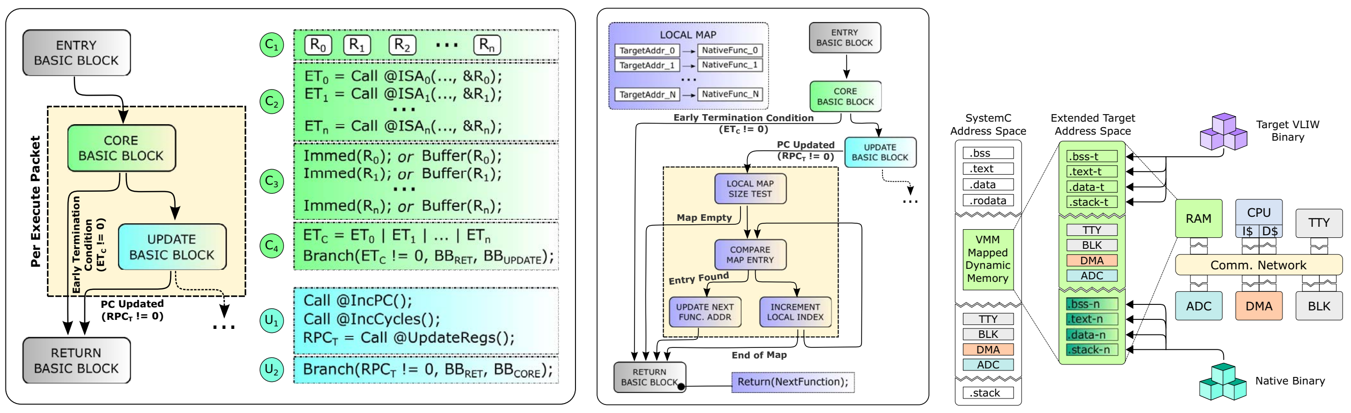
The target software is simulated in units of basic blocks by generating corresponding native functions. Once a native function completes its execution, it passes control to the *software kernel*, which looks at the simulated processor state and maps the target program counter to next native function. A simple option for accomplishing this control transfer is to generate a global map containing target addresses and corresponding native function pointers. This global map can be searched using traditional techniques, which impose significant overhead. We examine the following two *alternative* strategies for decreasing this search overhead:

- *Replacing Global Map with a Hashmap*: A hash function can be used to map a given target address to an index in a table containing native function pointers. A better idea is to use a *perfect hash* function, in order to avoid hash collisions and to evaluate if a significant gain in simulation speed can be achieved. For the current study we use the following simple but perfect hash function:
  $index = (target\_addr - text\_start) / word\_size$
- *Local Mappings in combination with Global Maps*: This scheme profits from the existence of direct branch instructions in target basic blocks and creates local target-to-native function mappings in each simulation function. To use these mappings, a linear search loop is introduced at the end of each generated function, as shown in Fig. 3(b). Each exit from the simulation function, except for early terminations, goes through this local search mechanism. In case of a match, the `NextFunction` pointer is updated and returned to the *software kernel*.

Once the next native function is found, using either global hashmap or local maps, the *software kernel* simply calls it without searching the global map.

F. Code Generation Modes

We can generate native code using different modes, as shown in Table I. One possible option is to generate native


ENTRY BASIC BLOCK
CORE BASIC BLOCK
UPDATE BASIC BLOCK
RETURN BASIC BLOCK
Per Execute Packet
Early Termination Condition (ET_C != 0)
PC Updated (RPC_T != 0)
C1 R0 R1 R2 ... Rn
C2 ET0 = Call @ISA0(..., &R0); ET1 = Call @ISA1(..., &R1); ... ETn = Call @ISAn(..., &Rn);
C3 Immed(R0); or Buffer(R0); Immed(R1); or Buffer(R1); ... Immed(Rn); or Buffer(Rn);
C4 ETC = ET0 | ET1 | ... | ETn Branch(ETC != 0, BBRET, BBUPDATE);
U1 Call @IncPC(); Call @IncCycles(); RPCT = Call @UpdateRegs();
U2 Branch(RPCT != 0, BBRET, BBCORE);
LOCAL MAP
TargetAddr_0 NativeFunc_0
TargetAddr_1 NativeFunc_1
TargetAddr_N NativeFunc_N
Early Termination Condition (ETc != 0)
PC Updated (RPCT != 0)
LOCAL MAP SIZE TEST
Map Empty
COMPARE MAP ENTRY
Entry Found
UPDATE NEXT FUNC. ADDR
INCREMENT LOCAL INDEX
End of Map
Return(NextFunction);
SystemC Address Space
Extended Target Address Space
.bss .text .data .rodata
VMM Mapped Dynamic Memory
TTY BLK DMA ADC
.stack
.bss-t .text-t .data-t .stack-t
.bss-n .text-n .data-n .stack-n
Target VLIW Binary
Native Binary
RAM CPU I$ D$ TTY
Comm. Network
ADC DMA BLK

(a) (b) (c)

Fig. 3. (a) Native Code Generation for Target Basic Blocks (b) Local Mappings per Simulation Function and (c) Memory Maps of Target and Native Binaries

code for execute packets only using the flow Ⓔ in Fig. 2(b). This mode suffers from slow simulation speed, as the control flow cost between native functions becomes dominant. In this mode, the generated simulator does not require dynamic translations, as we *know* that all branch instructions including indirect ones, will jump to start of execute packets, as explained in Section III (C). To improve the simulation speed, we can generate code using basic block mode using the flow Ⓑ, but this option is impractical using static translations as indirect branch instructions can jump to execute packets that lie in the middle of a target basic block. Lastly the hybrid mode using Ⓔ and Ⓑ in Fig. 2(b), which is practically feasible and interesting from the simulation point of view, is used to generate code for both basic blocks and execute packets in a *mutually exclusive* manner, i.e. in addition to generating code for basic blocks, we also generate code for all non basic block startup execute packets. The resulting simulator is larger in size but does not require runtime translations and executes faster as compared to other modes. Moreover this flow is better suited to VLIW context, as the minimum granularity of translations is an execute packet compared to an instruction in RISC machines, as shown in Fig. 2(a) Ⓘ.

V. Experimental Results

The experiments were conducted on a modest desktop machine *i.e.* Pentium(R) Dual-Core CPU E5300 (2.60 GHz, 2M Cache) with 2 GB of memory and running Linux version 2.6. We use a set of compute and control flow oriented applications to validate our results. The fibonacci example tests the control dominant behavior, by recursively calling itself twice until an index of 2 or less is reached, resulting in a tree-like structure of function calls. The factorial example computes its result in a recursive fashion as well but it calls itself once, resulting in list-like function call-stack. The *Inverse Discrete Cosine Transform* **(IDCT)** example crunches a number of input data blocks and shows the simulation cost of a compute-intensive application. Figs. 4(a), 4(b) and 4(c) compare our results with two different cycle accurate simulators from Texas Instruments. The *TI-C6x-FCA* is a Full Cycle Accurate simulator, which models all system components with cycle accuracy. The *TI-C6x-DFCA* is a Device Functional Cycle Accurate simulator, which models the processor with cycle accuracy and rest of system components at functional level. We observe a simulation speed-up of more than two orders of magnitude compared to *TI-C6x-FCA* simulator and an improvement between 30 to 40 times compared to *TI-C6x-DFCA* simulator, as shown in Table III. Furthermore the *functional correctness* of native simulators has been established using Execution Trace comparisons with TI simulators.

In case of static translation the execute packet mode is the slowest, as quite significant execution time is spent in control flow between packets. For improving the simulation speed, we *should* use the basic block mode, but as discussed in Section IV(F), it is not possible without dynamic translation support. Consequently we move to the hybrid translation mode, where we translate all of the *non-basic block startup* execute packets, in addition to the basic blocks. This mode costs more in terms of generated code size, but avoids the dynamic translation requirement as well as improves the simulation speed.

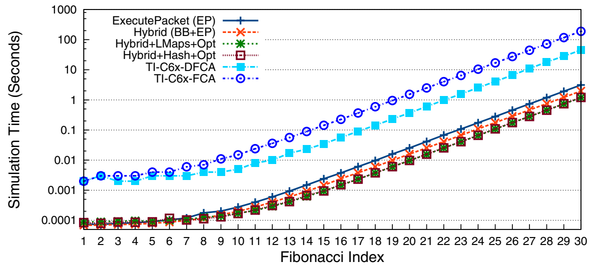
To further improve the simulation speed we have implemented a hashmap based switching in the *software kernel*. Similarly the local mapping support is also interesting, as we can find the *next native function* in more than 50% of the cases, as shown in Table II. We observe that the use of global hashmap or local maps results in *almost* the same amount of improvement. As our framework is implemented using LLVM, we can profit from the already available optimization passes without any additional costs. Lastly we compare our results to the different simulation/execution options. Fig. 4(d) and Table III show the overall comparison, from Full Cycle Accurate sim-

TABLE II
Target to Native Address Search Statistics

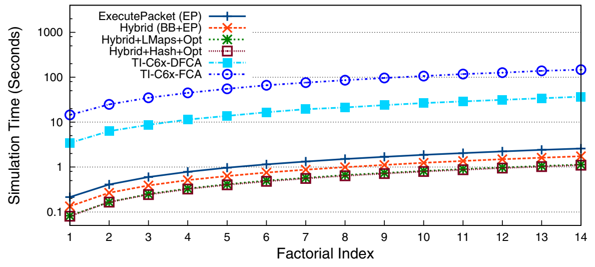
Application	Search Count Without LMaps	Search Count With LMaps	Improvement (Percent)
Fibonacci (30)	6,656,419	3,328,241	50%
Factorial (14, 100K)	5,700,104	2,800,085	51%
IDCT (40)	82,188	36,474	56%

TABLE III
Average Speedups/Slowdowns of SBT-Based Simulation

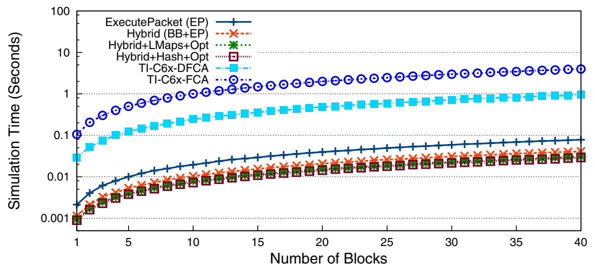
Application	TI-C6x-FCA Speedup	TI-C6x-DFCA Speedup	Native Slowdown	DirectHost Slowdown
Fibonacci	159x	39x	90x	101x
Factorial	132x	33x	205x	220x
IDCT	129x	31x	133x	141x



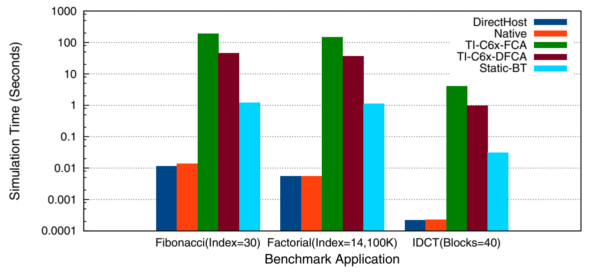
(a)



(b)



(c)



(d)

Fig. 4. Results for (a) Fibonacci (Recursive) (b) Factorial (100K Times) (c) IDCT and (d) Comparison of Simulation/Execution Methods

ulations to the Native and Direct Host executions. We must clarify that the Native and DirectHost execution options *do not* model the target processor architecture. These options are interesting to visualize the VLIW processor modeling overhead, which slows down the cycle accurate as well as static translation solutions.

VI. Conclusions and Future Work

In this paper we introduced a static binary translation flow for VLIW processors within native simulation context. We implemented this solution using the LLVM framework and our results show around two orders of magnitude simulation speed-up, as compared to the cycle-accurate simulators. On the down-side, as this approach is completely static, we have redundancy in the generated code, which could be compromised in the virtual prototyping contexts. In future we will add annotation generation support for performance estimation and further optimize the generated code in order to improve the simulation speed.

References

[1] J. R. Bell. Threaded code. *Communication of the ACM*, 16(6):370–372, 1973.

[2] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, and A. A. Jerraya. Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 969–972. ACM, 2005.

[3] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(12):1625 – 1639, dec. 2004.

[4] C. Cifuentes, M. Van Emmerik, and N. Ramsey. The design of a resourceable and retargetable binary translator. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 280 –291, oct 1999.

[5] E. Flamand. Strategic directions towards multicore application specific computing. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, page 1266, april 2009. Keynote speech.

[6] P. Gerin, M. M. Hamayun, and F. Pétrot. Native MPSoC co-simulation environment for software performance estimation. In *Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis*, pages 403–412, 2009.

[7] A. Gerstlauer, H. Yu, and D. D. Gajski. RTOS Modeling for System Level Design. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10130. IEEE Computer Society, 2003.

[8] A. A. Jerraya and W. Wolf. Hardware/Software Interface Codesign for Embedded Systems. *Computer*, 38(2):63–69, 2005.

[9] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–87, Mar 2004.

[10] J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen, and A. Polyak. Simulation/evaluation environment for a VLIW processor architecture. *IBM Journal of Research and Development*, 41(3):287 –302, may 1997.

[11] F. Pétrot, N. Fournel, P. Gerin, M. Gligor, M. M. Hamayun, and H. Shen. On MPSoC Software Execution at the Transaction Level. *IEEE Design & Test of Computers*, 28(3):2–11, 2010.

[12] M. Reshadi, P. Mishra, and N. Dutt. Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation. *ACM Trans. Embed. Comput. Syst.*, 8(3):20:1–20:27, Apr. 2009.

[13] H. Shen, M.-M. Hamayun, and F. Petrot. Native Simulation of MPSoC Using Hardware-Assisted Virtualization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(7):1074–1087, July 2012.

[14] V. Zivojnovic, S. Tijang, and H. Meyr. Compiled simulation of programmable DSP architectures. In *VLSI Signal Processing, VIII, 1995. IEEE Signal Processing Society [Workshop on]*, pages 187 –196, oct 1995.