Fast Simulation of Systems Embedding VLIW Processors

Luc Michel
Tima Laboratory
CNRS/Grenoble INP/UJF
luc.michel@imag.fr

Nicolas Fournel
Tima Laboratory
CNRS/Grenoble INP/UJF
nicolas.fournel@imag.fr

Frédéric Pétrot
Tima Laboratory
CNRS/Grenoble INP/UJF
frederic.petrot@imag.fr

ABSTRACT

Virtual prototyping of MPSoCs requires fast processor simulation models. Dynamic binary translation is an efficient technology for instruction set simulation, but as it is basically used for effortless code migration, it targets mostly general purpose processors. As many heterogeneous MPSoCs include VLIW processors, we propose and detail in this paper a strategy to perform dynamic binary translation of VLIW codes on scalar architectures for simulation purposes. Our simulation experiments show that it is a few orders of magnitude faster than direct instruction interpretation, although the translator includes no optimization.

Categories and Subject Descriptors

C.0 [**General**]: Modeling of computer architecture; B.4.4 [**Performance Analysis and Design Aids**]: Simulation; D.3.4 [**Processors**]: Interpreters

General Terms

Performance, Design

Keywords

VLIW, Simulation, Binary Translation

1. INTRODUCTION

Many current integrated systems are composed of a general purpose processor assisted by one or a few (usually very long instruction word - VLIW - based) digital signal processors (possibly deeply embedded into devices) on which generally regular but computation intensive functions are offloaded. Since depending on the final product target many designs alternatives are possible, efficient simulation of the complete hardware/software system is needed. Studies have shown that the time spent in Instruction Set Simulators (ISS) is a significant part of the whole MPSoC (Multi-Processor System on Chip) simulation time [1], and this part raises with the internal complexity of the processors. As the trend is to embed more and more processors, fast instruction set simulation is of utmost importance.

Instruction interpretation technologies are numerous, as detailed in, *e.g.*, [2], and dynamic binary translation (DBT) is now a solution making its way into MPSoC simulation [3, 4, 5, 6] that is fast while allowing accurate performance evaluation [4]. DBT is a mature technology, and it is used for scalar to scalar and scalar to VLIW codes translations, at first democratized by companies building processors to allow the use of legacy codes on newer [7], sometimes disruptive [8], processor architectures. MPSoC simulation opens a new ground for this technology: translation of VLIW code on a scalar architecture. Indeed, and as far as we know, such a translation has never been reported, due to the fact that this VLIW to scalar translation is of interest only in simulation, since there is no captive VLIW legacy code base.

The focus of this work is thus to define a strategy for the translation of VLIW codes on scalar processors and demonstrate its efficiency for building ISS.

The paper is organized as follows. Section 2 presents the field of work and the related researches. Section 3 defines the problem in a comprehensive way, and section 4 details the solution that we propose to solve it. Section 5 presents and analyzes experimental data, before summarizing our contribution in Section 6.

2. RELATED WORK

Binary translation is aiming at transforming instructions of one ISA to another[7]. This process can be executed at two different instants: offline, for the static binary translation (SBT)[9, 10], or at run-time, for the dynamic binary translation (DBT)[11, 12].

The main and original goal of binary translation has been migration and compatibility. Indeed, numerous works propose solutions based on binary translation (static or dynamic) aiming at exploiting new features of more recent versions of an architecture, for example UQBT with the SPARC ISA[9] or Transmeta with their so called "code-morphing"[13] hardware technology targeting the x86 ISA. Others propose solutions easing migration from some architecture to another [10, 14, 15].

More applications of this technique have emerged in the recent years, for example Just-in-time compilation (JIT) for Java, native binary acceleration[16] or virtualization and system simulation [12]. Even more recently, interpretive languages have benefited from these techniques, in web pages (the Tamarin JIT for Javascript) or for scripting (Python, Ruby, . . .). The technique was popularized in the early 80's by the implementation of SmallTalk[17].

In the DBT field, efforts have been made to build machine adaptable binary translator. In that way, the translator can be adapted to numerous architectures with a reduced effort [11, 12, 18]. These translators are all based on an Intermediate Representation (IR) so that the complete process of binary translation can be described in a two-phases manner, as proposed on Figure 1. The IR is machine independent to guaranty the re-targetability of the translator. This representation can be viewed as a generic machine instruction set, it is even called micro-operations in QEMU [12] or I-RTL in Walkabout [11].

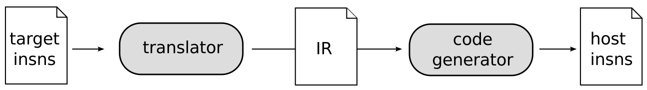


Figure 1: Machine adaptable DBT process

The first phase is in charge of translating the target instruction into the machine independent IR. In most cases, this phase is called translator. The second phase is then in charge of encoding the IR into host instructions. It is called either code generation by some authors, by analogy with the compilation notion, or encoder.

Among all the works presented until now, many have addressed the issue of translating binaries from a scalar target architecture to another scalar architecture. This case is the most straightforward one, since by falling back in an elementary IR, one can always find a valid translation for instructions regardless of their complexity [12].

A few works have focused on the efficient execution of SIMD instructions. Among them we can cite an effort from FX!32 [10] to translate MMX SIMD instructions to the Alpha instruction set, the work of [19] which focuses the SIMD instructions (MMX/SSE) on an IA64, and [20] that proposes a general approach that is demonstrated on ARM Neon extensions.

Other works have proposed translations from scalar architectures to parallel architectures, more precisely to VLIW architectures such as DAISY [14] or the Itanium or IA64 [8, 15]. These solutions are more complex than the previous ones since on top of finding a matching between instructions, these solutions have to extract the instruction level parallelism (ILP) to take advantage of the host architecture for better performances.

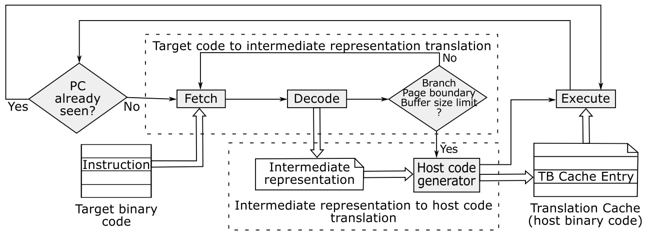
The use of Dynamic Binary Translation as an instruction set simulation tool for system level simulation has been introduced only recently[4, 3]. As the translation of VLIW code into sequential code does not have a practical interest outside of this topic (unless for the support of legacy code, but VLIW processor are indeed quite specialized and they do not have a large code base), and to the best of our knowledge, the DBT of VLIW code into scalar code has not been detailed in the literature as of today.

3. PROBLEM DEFINITION

3.1 Dynamic Binary Translation Technique

Figure 1 gives a conceptual overview of the Dynamic Binary Translation (DBT) process, whereas Figure 2 (borrowed from [2]) gives the overall details of the process. As stated before, this translation is made in a two-phase manner with a translation phase (from target instruction to IR) and a code generation (from IR to host instructions). This is mainly for easy retargetability purposes.

As shown on Figure 2 these two phases are not called sequentially on each instructions. Indeed, the translation phase is first accomplished on a complete sequence of instructions. This group of instructions are sequentially executed instructions, which means that the translation of a target branch instruction triggers the end of the group. These groups are called Translation Blocks (TB) and are very close to the compilation notion of Basic Block. The main difference with Basic Blocks is that Translation Blocks also end on other conditions like memory page frontiers. Once a TB is translated into IR operations, the code generator produces at once the corresponding host instructions, called Translated Basic Block (TBB). The TBB is then wrapped with a prologue and an epilogue to match a function template. Finally, the TBB is executed using a function call. At the end of the execution of the TBB, the control returns to the DBT engine to execute the next TBB. For performance reasons, the TBB are cached for later reuse in order to amortize the translation and code generation costs. The DBT engine will then first lookup in the cache to find an existing translation and execute it in case of hit or call the translator in case of miss. As can be realized from this process, the TBs are translated and generated independently of their context of use. Indeed, all possible predecessors of a specific TB are not known at translation time, more precisely only one is known: the one from which the TB was encountered the first time.



Target code to intermediate representation translation
No
PC already seen?
Yes
No
Fetch
Decode
Branch Page boundary Buffer size limit ?
Execute
Yes
Instruction
Intermediate representation
Host code generator
TB Cache Entry
Target binary code
Intermediate representation to host code translation
Translation Cache (host binary code)

Figure 2: Overview of the DBT process

Figure 3 gives an example of the translation results of an ARM instruction on an x86 host made by the QEMU DBT engine. The example also depicts, in the middle, the IR operations used for this translation, QEMU micro-operations.

```
and r2,r1, r5          mov_i32 tmp0, r5                  mov 0x05(%ebp), %esi
                       mov_i32 tmp1, r1                  mov 0x04(%ebp), %ebx
                       and_i32 tmp0, tmp0, tmp1          and  %esi, %ebx
                       mov_i32 r2, tmp0                  mov %ebx, 0x08(%ebp)
```

Figure 3: DBT example: ARM to x86

As can be realized from this example, the translation of a simple logic instruction results in a sequence of many instructions of the host architecture. Lines 1, 2 and 4 of the x86 generated code also underline the fact that the registers of the target are placed in memory for the simulation. The sequence of extra generated instructions aims at moving values of target registers in memory to host registers and vice-versa. This resembles the *to* and *from* memory moves an unoptimized compiler would produce, where the target registers would be the variables of the source code.

For the sake of readability, in the rest of this article we will simplify the IR by aggregating the memory to host register and the arithmetic and logic IR operations. The previous example IR code will be written as a single IR operation: `and_i32 r2, r1, r5` (whose meaning is $r_2 \leftarrow r_1 \wedge r_5$).

3.2 VLIW Architectures specificities

The VLIW idea can be summarized as making a processor simpler yet powerful by having the compiler provide the Instruction Level Parallelism (ILP) explicitly in the instruction word. The VLIW approach can be implemented in general purpose processing, *e.g.* tilera, transmeta crusoé or IA64, but is most often used for digital signal processing, *e.g.* NXP TriMedia, AD SHARC, STM ST200, Atmel mAgic-VLIW and TI C6x, where applications are data dominated. Many *ad-hoc* processing engines in embedded and integrated systems are also VLIW based, as they provide a high computing vs power efficiency. All match the main concept of VLIW but each offers different architectural implementation choices. The implementation choices are mainly trade-offs regarding simplicity of the design against compiler complexity, the major one being bypasses and stalls against register update latencies (called delay slots in the sequel).

Our goal here is not to give a complete overview of these architectures, but to provide clues about which architectural specificities of VLIW could be issues in the DBT process. For that purpose we illustrate this article using the TI C6x DSP architecture, which contains all of the architectural features making DBT on a scalar architecture difficult. The Listing 1 is a small piece of C6x code that will we use throughout the paper.

Listing 1: VLIW code Example: TI C6x

```
   ;  Instruction             ; Cycle ; L ;
2     add   .L1 a2, a1, a2    ; 1     ; 1 ; a2 <- a2 + a1
   || sub   .D1 a1, a4, a6    ; 1     ; 1 ; a6 <- a1 - a4
4  || mpy32 .M1 a2, a1, a3    ; 1     ; 3 ; a3 <- a2 * a1
      b     .S1 0xbeef        ; 2     ; 5 ;
6                                     ;
      sub   .D1 a6, a3, a3    ; 3     ; 1 ; a3 <- a6 - a3
8     add   .L1 a2, a3, a4    ; 4     ; 1 ; a4 <- a2 + a3
      mpy32 .M1 a2, a3, a5    ; 5     ; 3 ; a5 <- a2 * a3
10    nop    2                ; 6,7   ; X ;
      add   .L1 a2, a3, a5    ; XXX   ; X ; a5 <- a2 + a3
```

First of all, ILP expressed in the instruction word, is illustrated in the example by lines 2, 3, 4. The '||' means that the following instruction will be executed in the same cycle as the previous one in the code. A group of parallel instruction is an Execution Packet. The main characteristic of the execution packet is that multiple instructions can use the same register(s) as input, but more interesting, in the same execution packet, one instruction can write its result in a register whereas other instructions use it as operand. Obviously, the operand will have the value at the beginning of the cycle and not the result value. For example, the register `a2` is an operand register at line 2 and 4, but also a destination register at line 2.

The next characteristic concerns the delay slot of arithmetic and logic instructions. Indeed, some of the instructions available in these processors are too complex to be completed in one cycle. Instead of stalling the pipeline until the availability of the result in the destination register, the compiler has the responsibility to ensure that instructions using the result of this instruction will not be scheduled before the end of the delay. It also means that all instructions executed in between can read this register to get its old value, and it is even possible to write it, the actions occurring after the corresponding latency (result undefined in case of a tie). For example, register `a3` value resulting of the `mpy32` instruction will only be available at cycle 5 for the second `mpy32` since `mpy32` has a three cycle delay. Moreover, the `a3` value of lines 7 is the old value and its new value, used in the `add` line 8 is overwritten by the result of the first `mpy32`.

The last specificity is related to branches. Branch instructions have also delay slots, which means that instructions following the branch will be executed before branching actually occurs. For example, in the C6x the delay of a branch is 5 cycles. This means that the branch will effectively occur after the second cycle of `nop` in the example of Listing 1. The instructions between the branch and the effective branch are executed as if they were placed before the branch and the branch was immediate.

3.3 A Naïve DBT of VLIW code

The Algorithm 1 presents a simple version of the classical scalar DBT algorithm that translates the target code until the end of the translation block. The Listing 2 presents the resulting translation of the VLIW code depicted in Listing 1 after applying this algorithm.

Algorithm 1 Naive VLIW DBT algorithm

```
while ¬end_of_tb() do
    insn ← get_next_insn()
    translate(insn)
end while
```

Listing 2: Naive DBT of VLIW code

```
   ; IR operation               ;    original instruction ; Cyc
2  add_i32   a2, a2, a1         ;    add   .L1 a2, a1, a2 ; 1
   sub_i32   a6, a1, a4         ; || sub   .D1 a1, a4, a6 ; 1
4  mul_i32   a3, a2, a1         ; || mpy32 .M1 a2, a1, a3 ; 1
   mov_i32   pce1, $0xbeef      ;    b     .S1 0xbeef     ; 2
6  exit_tb   0x0                ;                         ;
   sub_i32   a3, a6, a3         ;    sub   .D1 a6, a3, a3 ; 3
8  add_i32   a4, a2, a3         ;    add   .L1 a2, a3, a4 ; 4
   mul_i32   a5, a2, a3         ;    mpy32 .M1 a2, a3, a5 ; 5
10 # nop 2                      ;    nop    2             ; 6,7
   add_i32   a5, a2, a3         ;    add   .L1 a2, a3, a5 ; XXX
```

Because of the VLIW architectural specificities, the naïve scalar DBT produces code that has an incorrect behavior.

First, the parallel execution and more precisely the parallel read of source registers is broken. As the translation has to sequentialize the instructions of an execution packet, the Write-after-Read (WAR) dependency due to the parallel read of operands which occurs before the result write back in destination registers is not respected. For example, the execution of the first `add` will be finished when the `mul` will be executed. The result of the first `add` will have overwritten the value of the correct operand for the `mul`.

Second, delay slots of arithmetic operations are also not respected, because all IR operations are instantaneous from a simulation point of view. Then results of arithmetic and logic instructions with delay slots will be written back to their destination register before the next cycle is evaluated. For example the result of the `mul` will be available for the following `sub`.

Finally, the translation for instructions following the branch is represented in the Listing 2, but in the standard process, the translator will never translate them since it will end the TB at the branch instruction. The delay slot instructions should be part of this TB, but they are not in the current process. The branch delay slot are thus not respected.

To conclude, the current scalar DBT is not able to translate VLIW code for execution on scalar machines. To solve this issue, it is mandatory to modify the translation process to handle as efficiently as possible the delayed register updates.

4. DBT FOR VLIW

4.1 VLIW DBT Extension principles

First, the DBT has to handle the fact that multiple instructions are executed in parallel which may have destination registers being source registers of others. The simple approach that consists of reordering the instructions in the execute packet so that the instructions writing a register are placed after the one reading it is the most efficient, but does not always have a solution (*e.g.* circular dependencies).

A heavier but systematic approach consists of introducing copies of the original registers for instructions using an overwritten register, so we opt for this solution. A simple way to apply it to solve the WAR dependency due to parallel execution is to rename the target registers using a single assignment scheme. This strategy is similar to the Static Single Assignment (SSA [21]) form used in compilation if we consider target registers as variables.

This renaming is quite simple. Each register, for example `a0`, is present under multiple names build incrementally, *i.e.* `a0_0`, `a0_1`, and so on. We create one new version of the register each time it is assigned as a destination of an instruction and use it in the subsequent execute packets.

Applied to the intra-instruction WAR dependency, we will have at most two living versions of the register in each execute packet, one corresponding to the old value and one corresponding to the updated value.

The Algorithm 2 gives the DBT algorithm adapted for this single assignments in which we create replicates *r* on each output operand *op* assignments in an Execution Packet (*ep*). At the end of each Execution Packet we finally alias the target registers to their newer replicates and remove old ones.

Algorithm 2 Single Assignment registers applied on naïve DBT

```
while ¬end_of_tb() do
    ep ← get_next_execute_packet()
    for $insn \in ep$ do
        op ← get_op_out($insn$)
        r ← new_replicate($op$)
        translate($insn$, $r$)
    end for
    do_end_of_cycle_updates()
end while
```

The application of this algorithm on original VLIW code of Listing 1 produces the IR operations sequence presented in Listing 3.

Listing 3: Single Assignment registers example

```
; IR operation              ;    original instruction ; Cyc
add_i32  a2_1, a2_0, a1_0 ;    add   .L1 a2, a1, a2 ; 1
sub_i32  a6_1, a1_0, a4_0 ; || sub   .D1 a1, a4, a6 ; 1
mul_i32  a3_1, a2_0, a1_0 ; || mpy32 .M1 a2, a1, a3 ; 1
mov_i32  pce1_1, $0xbeef  ;    b     .S1 0xbeef     ; 2
exit_tb  0x0              ;                         ;
sub_i32  a3_2, a6_1, a3_1 ;    sub   .D1 a6, a3, a3 ; 3
add_i32  a4_1, a2_1, a3_2 ;    add   .L1 a2, a3, a4 ; 4
mul_i32  a5_1, a2_1, a3_2 ;    mpy32 .M1 a2, a3, a5 ; 5
# nop 2                   ;    nop    2             ; 6,7
add_i32  a5_2, a2_1, a3_2 ;    add   .L1 a2, a3, a5 ; XXX
```

This solution clearly solves the WAR problem, since the operand register of the `mul` is the old version of register `a2` and not the newer one that is the outcome of the `add` instruction. Unfortunately this simple register renaming strategy does not solve the issue of arithmetic operations delay slots.

Indeed, `a3_1` will be used for the `sub` instruction whereas this value is not the correct one since it is not available yet in the architecture. To solve this, the old version must be used instead of the new one for all cycles within the producing instruction delay slots. This results in keeping possibly alive more than two versions of the same register at the same moment, which is clearly not the case in conventional SSA.

The translation algorithm is adapted to handle these arithmetic and logic instructions delay slots and presented in Algorithm 3. The creation of replicates takes into account the result availability delay. At the end of each Execution Packet we update the delays of replicates and alias the target registers to new replicates only if delay falls to zero. The translation result of the previous example is given in Listing 4.

Algorithm 3 Delay slot handling

```
while ¬end_of_tb() do
    ep ← get_next_execute_packet()
    for $insn \in ep$ do
        d ← get_delay($insn$)
        op ← get_op_out($insn$)
        r ← new_replicate($op$, $d$)
        translate($insn$, $r$)
    end for
    do_end_of_cycle_updates()
end while
```

The number of living versions of the registers increases, but fortunately, this number is still bounded to a reasonable amount, which is the maximum instruction latency plus 1. Indeed, the worst case is a sequence of largest latency instructions writing to the same register. In that case, one living version must exist for each instruction executed between the execution of the first occurrence and the availability of its result, plus one version for the previous value.

Listing 4: Arithmetic delay slots handling

```
; IR operation              ;    original instruction ; Cyc
add_i32  a2_1, a2_0, a1_0 ;    add   .L1 a2, a1, a2 ; 1
sub_i32  a6_1, a1_0, a4_0 ; || sub   .D1 a1, a4, a6 ; 1
mul_i32  a3_1, a2_0, a1_0 ; || mpy32 .M1 a2, a1, a3 ; 1
mov_i32  pce1_1, $0xbeef  ;    b     .S1 0xbeef     ; 2
exit_tb  0x0              ;                         ;
sub_i32  a3_2, a6_1, a3_0 ;    sub   .D1 a6, a3, a3 ; 3
add_i32  a4_1, a2_1, a3_2 ;    add   .L1 a2, a3, a4 ; 4
mul_i32  a5_1, a2_1, a3_1 ;    mpy32 .M1 a2, a3, a5 ; 5
# nop 2                   ;    nop    2             ; 6,7
add_i32  a5_2, a2_1, a3_1 ;    add   .L1 a2, a3, a5 ; XXX
```

Finally, the branches are handled in two phases. Firstly, the code responsible for the computation of the branch target address is generated when encountering the branch. Secondly, a TB ending instruction is produced after the delayed instructions. Listing 5 gives the corresponding modified sequence of IR operations.

Listing 5: Branch delay slots handling

```
; IR operation              ;    original instruction ; Cyc
add_i32  a2_1, a2_0, a1_0 ;    add   .L1 a2, a1, a2 ; 1
sub_i32  a6_1, a1_0, a4_0 ; || sub   .D1 a1, a4, a6 ; 1
mul_i32  a3_1, a2_0, a1_0 ; || mpy32 .M1 a2, a1, a3 ; 1
mov_i32  pce1_1, $0xbeef  ;    b     .S1 0xbeef     ; 2
                          ;                         ;
sub_i32  a3_2, a6_1, a3_0 ;    sub   .D1 a6, a3, a3 ; 3
add_i32  a4_1, a2_1, a3_2 ;    add   .L1 a2, a3, a4 ; 4
mul_i32  a5_1, a2_1, a3_1 ;    mpy32 .M1 a2, a3, a5 ; 5
# nop 2                   ;    nop    2             ; 6,7
exit_tb  0x0              ;
add_i32  a5_2, a2_1, a3_1 ;    add   .L1 a2, a3, a5 ; XXX
```

4.1.1 TBB entry and exit states

Using the above described techniques (plus others not described here but similar in spirit, to handle for example predicated instructions), we obtain a sequential intermediate translation using registers replicates accomplishing the exact functional behavior of the source VLIW code. As can be realized by using the renaming strategy, the working version (currently used versions) of the registers is unpredictable. More precisely, as TB are translated independently, we cannot rely on these working versions. Indeed, when a TBB has several different predecessors, we cannot guaranty that the working version will be identical at the exit of all the preceding TBBs, and the translator does not even know if a TBB has more than one predecessor when performing the translation.

The solution for handling this need of TBB independence is to define an unique entry and exit working register set for TBs. This set will be called the canonical state of replicate registers. In that way, once translated, the TB will be reusable from all TBs pointing to it. The canonical state will be composed of the first replicate of each register, in which are mapped the first version of each register at the beginning of TB translation (`an_0` in our case).

The Algorithm 4 is called at the end of each translation block generation to fulfill the canonical state requirements

Algorithm 4 Canonical state return algorithm

```
for $r \in$ target_registers do
    $rep \leftarrow$ get_current_replicate($r$)
    if $\neg$is_first_replicate($rep$) then
        gen_mov(get_canonical_replicate($r$), $rep$)
    end if
end for
```

Unfortunately, in some cases, the delay slots crosses the border of a TBB, for example the `a5_1` register which is the result of the last `mul` in the code example. In that case, an external mechanism needs to be set up to handle these delay slots. This mechanisms needs to be external to the translator because of the TBB independence requirements. Indeed, the translator looses the information about delay slots when exiting a TB and thus only an execution time mechanism can manage the delay slots in that case.

This mechanism records the pending delayed registers with their current cycle delay when exiting the current TBB at run-time. The next TBB executed consults the recorded data during its first execution cycles (bounded by the maximum possible latency) to check if there are some registers needing to be updated as they reach their latency. This is done through function calls inserted by the translator.

4.2 Complexity of the modifications

From a pure functional point of view, all modifications needed to implement the VLIW DBT are part of the translator. The code generator does not strictly need to be modified for this support.

The modifications needed are first a change from a simple array of registers in which each cell represents a target register to an array of these registers in which a line represents all the replicates of one target register. This change requires to modify all mechanisms at translation time to allocate the correct replicate each time an assignment to a target register occurs.

The delay slot handling, delaying use of a newer replicate, can be modeled using a queue, in which future versions of registers can be inserted at the corresponding delay. At the end of each cycle all delayed register versions progress of one cycle in the delay queue. All these computations occur at translation-time. Complexity is once again limited.

The handling of the canonical state implies a modification of the translator, but impacts the generated code. Indeed, the easiest way to return to the canonical state at the end of a translation block is to generate instructions that move current working replicates of registers to the ones defined in the canonical state. This solution has a limited complexity as far as we consider only the translation phase but has the unfortunate side-effect of increasing the TBB size, and thus its execution time.

Finally, the handling of delay slots crossing TBB edges is a pure run-time mechanism. This mechanism has to be identical for all TBB to be valid for all sequences of TBB. The amount of generated code for this purpose is not huge, as it consists of generating function calls (usually called *helpers*) to propagate correctly the register updates. Even though the functions themselves are quite straightforward since they only handle registers updates through a runtime delay queue similar to the one described before, the runtime overheads necessary to set-up and perform these calls are quite large compared to a simple sequence of generated host instructions.

5. EXPERIMENTS

We have implemented a translator for the C6x processor family using the previously presented concepts within the QEMU DBT environment. Firstly, we have performed unitary tests (not reported here) to ensure correctness of the translation. Secondly, we have analysed the generated code to quantify the contribution of each stage of the translation to it. Finally, we have run a few kernels for performance measurements that we compared with execution speeds obtained using the simulator, based on instruction interpretation, included in Texas Instruments Code Composer Studio (CCS).

- The `fibo` program computes the Fibonacci number of a given index using the naïve recurrence relation $F_i = F_{i-1} + F_{i-2}$ with $F_0 = 0$ and $F_1 = 1, \forall i \in \mathbb{N}$. It involves a lot of branches,
- The `matrix` program multiplies two square matrices of size $j \times j$. It performs a lot of memory accesses and operations, and branches occur only at loop boundaries,
- The `idct` program computes the inverse discrete cosine transform of k jpeg macro-blocks of 64 16 bit unsigned integers using Loeffler algorithm. It does a huge amount of arithmetical and logical operations on fixed point integers, but not many loop iterations.

Note that the parameters i, j, k are given as constants in the program so as to benefit from the compiler static optimizations, if any.

The first Figure (Fig. 4) represents the *origin* of the host instructions generated in the TBB in average for varying values of each program parameter. In order to evaluate actual general purpose code, we also translated an application containing calls to the standard C library functions.

The *Target instructions* gives the average target instructions within a translation block for the given test. It ranges from an average of 7 for the stdlib code to up to almost 40 for the matrix one. The *Micro-ops for pure target instructions simulation* represents the number of bare instructions, similar to what is necessary for a scalar to scalar translation, and features a factor of 4 to 5 (`idct` has the biggest factor) compared to the target instructions over all programs. The other origins are due to the VLIW nature of the target code, as explained in Section 4. The *Micro-ops to handle canonical state return* reports the number of prologue and epilogue instructions necessary to make each TBB independent of its calling context. It varies between 20 and 40 instructions, and represents a larger percentage overhead for the smaller translated blocks (`fibo` and `stdlib`). The *Micro-ops to handle cross-border delays* represents the overhead in term of micro-operations due to the calls to the run-time mecanism that handle cross-border delays. It represents overall very few instructions, always less than 10 even for the largest basic blocks. Overall, for these simple programs, the instruction mix is very consistent for the whole range of the parameter.

The following set of curves (Figs. 5, 6 and 7) gives the simulation run times using the exact same binary code produced by the TI compiler on both platforms. The `fibo` and `matrix` tests have an

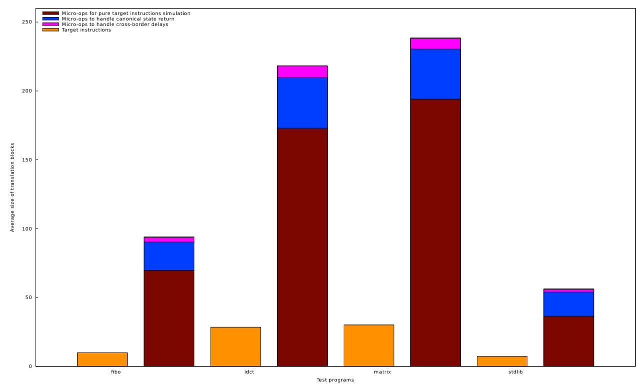


Figure 4: Host instruction origin for the `fibo`, `idct`, `matrix`, `stdlib` test

exponential algorithmic complexity with respect to their parameter, whereas the `idct` has a linear one. Note that for these measures, we didn't take into account the overhead due to the simulators initializations. For the CCS simulator, the entire java environment is loaded, and the program is ready to run when the time measurement begins. For QEMU, the measure is started when QEMU initiates the translation of the very first translation block.

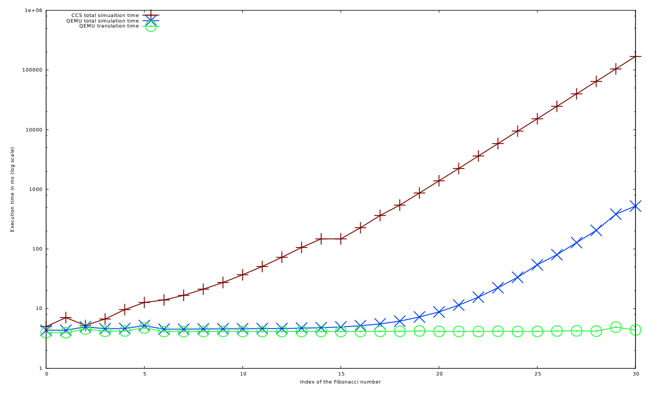


Figure 5: Execution time of the `fibo` program

These tests clearly show the performance advantages of the DBT, compared to the direct interpretation. When the simulated program is very small in term of computation workload, (when the parameter is near 0), we can see that the performances are equivalent. On the `idct` test, the CCS simulator is even better when computing one block. But when the parameter increases, the dynamic binary translator gains a dramatical advantage, thanks to the reuse of the translated code.

For the `fibo` test, there is about a factor 320 for the lasts indexes between QEMU and CCS. For the `matrix` test, the factor is about 175 for the bigger matrices. Finally, for the bigger number of blocks to compute in the `idct` test, QEMU is arount 18 to 25 times faster than the CCS simulator. It shows that, even with all the corner cases workarounds described in Section 4 that make the generated code more complex, the DBT approach is still efficient enough to be considered as a viable solution.

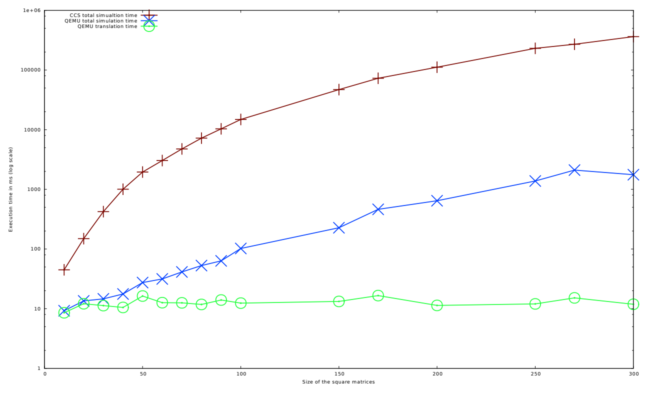


Figure 6: Execution time of the `matrix` program

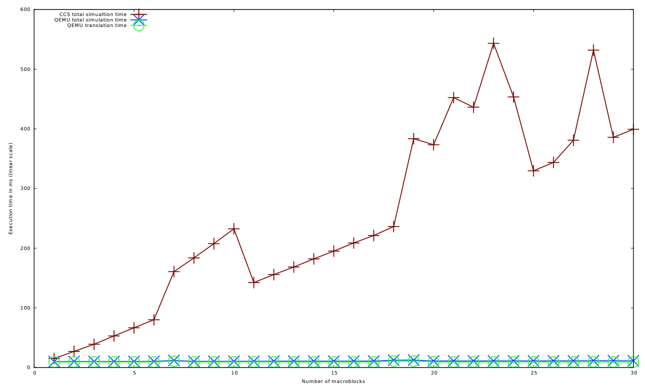


Figure 7: Execution time of the `idct` program

6. SUMMARY AND PERSPECTIVES

In this paper we have presented a solution for the Dynamic Binary Translation (DBT) of VLIW codes onto scalar machines. The approach requires deep modifications in the translator due to the registers update latencies. It has been dealt with through a mix of techniques acting both at translation and execution time. Even though these overheads are not negligible, our experiments have shown gains from 20× to 300×. Nevertheless, the translation is still basic, in the sense that our goal was to demonstrate the feasibility of VLIW code translation on scalar architecture and we did not perform any optimization on it. As a matter of fact, QEMU is known to perform very fast translation, but to produce straightforward host code. Due to the nature of the translated code, it is quite likely that optimizing often executed translated blocks (called hotspots) will result in large gains. Indeed, very naïve local optimization easily gains 20% speed according to [22], and [23], optimizing along the most often used paths, claims a gain of above 60%. A future work is thus to enhance the quality of the translated code while keeping the translation overhead under control, by using the above solutions and by trying to optimize the VLIW specific parts of the translator.

References

[1] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *39th Design*

Automation Conference, pages 22–27, 2002.

[2] F. Pétrot, N. Fournel, P. Gerin, M. Gligor, M. Hamayun, and H. Shen. On MPSoC software execution at transaction-level. *Design & Test of Computers, IEEE*, 28(3):32–43, 2011.

[3] M. Montón, J. Carrabina, and M. Burton. Mixed simulation kernels for high performance virtual platforms. In *Forum on Specification & Design Languages*, pages 1–6, September 2009.

[4] M. Gligor, N. Fournel, and F. Pétrot. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *CODES+ISSS*, pages 71–80, October 2009.

[5] W. Mueller, M. F. da S. Oliveira, H. Zabel, and M. Becker. Verification of real-time properties for hardware- dependent software. In *High Level Design, Verification and Test Workshop*, Anaheim, CA, June 2010.

[6] R. Leupers, L. Eeckhout, G. Martin, F. Schirrmeister, N. Topham, and X. Chen. Virtual manycore platforms: Moving towards 100+ processor cores. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 715–720, Grenoble, France, March 2011. IEEE.

[7] C. Cifuentes and V. Malhotra. Binary translation: Static, dynamic, retargetable? In *IEEE International Conference on Software Maintenance*, page 340, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[8] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on itanium-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–202, 2003.

[9] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *Computer*, 33(3):60–66, 2000.

[10] P. J. Drongowski, D. Hunter, M. Fayyazi, D. Kaeli, and J. Casmira. Studying the Performance of the FX!32 Binary Translation System. In *the First Workshop on Binary Translation*, 1999.

[11] D. Ung and C. Cifuentes. Machine-adaptable dynamic binary translation. In *DYNAMO '00*, pages 41–51, 2000.

[12] F. Bellard. QEMU, a fast and portable dynamic translator. In *the USENIX Annual Technical Conference*, pages 41–46, 2005.

[13] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 15–24, 2003.

[14] K. Ebcioğlu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *ISCA 24*, pages 26 – 37, 1997.

[15] C. Zheng and C. Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer*, 33(3):47–52, 2000.

[16] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM conference on Programming language design and implementation*, pages 1–12, 2000.

[17] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proc. of the 11th ACM SIGACT-SIGPLAN Symp. on Principles of programming languages*, pages 297–302, 1984.

[18] B. Yuncheng, L. Alei, and G. Haibing. Design and implementation of crossbit: Dynamic binary translation infrastructure. *Computer Engineering*, 33(23):100 – 134, 2007.

[19] J. Li, Q. Zhang, S. Xu, and B. Huang. Optimizing dynamic binary translation for SIMD instructions. In *Code Generation and Optimization*, March 2006.

[20] L. Michel, N. Fournel, and F. Pétrot. Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation. In *Proceedings of the Design, Automation & Test in Europe Conference*, pages 277–280. IEEE, 2011.

[21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Sytems*, 13:451–490, 1991.

[22] Y. Hu, H. Jin, Z. Yu, and H. Zheng. An optimization approach for qemu. In *1st International Conference on Information Science and Engineering*, pages 129 –132, dec. 2009.

[23] D. Jones and N. Topham. High speed CPU simulation using LTU dynamic binary translation. In *High Performance Embedded Architectures and Compilers*, volume 5409 of *LNCS*, pages 50–64. 2009.