Dynamic Binary Translation of VLIW Codes on Scalar Architectures

Luc Michel and Frédéric Pétrot, *Member, IEEE*

***Abstract*—Many of the recently announced integrated manycore architectures targeting specific applications embed several, if not many, very long instruction word (VLIW) processors. To start developing software while the hardware is still being designed, virtual prototypes of the full system are commonly used. Fast processor simulation is thus a requirement. To that aim, this paper introduces a strategy to perform dynamic binary translation (DBT) of VLIW codes on scalar architectures. We propose a high level simulation algorithm which takes into account VLIW oddities, such as explicit instruction parallelism, instructions with non unit register update latency, and delayed slots in branches. We present the implementation details of this algorithm within a DBT environment, as it raises many corner cases that are irrelevant in scalar DBT. Our experiments confirm that our solution is functionally correct, and show speedups of 1 and 2 orders of magnitude compared to raw instruction interpretation, even though no optimizations were performed on the code during and after translation.**

***Index Terms*—Computer architecture, design automation, dynamic binary translation (DBT), system-level design, very long instruction word (VLIW), virtual prototyping.**

I. Introduction

Multiprocessor systems on chip (MPSoC) including a few processors are currently widely used for building application specific devices. A recent architectural trend, related to complex issues in new technological nodes, advocates the use of many processors to perform demanding computations [1]. Most of these application specific MPSoCs are geared toward network and multimedia applications, which intrinsically have a high level of fine grain parallelism. Therefore, processors having a very long instruction word (VLIW) architecture are an interesting tradeoff between flexibility and efficiency, as they allow exploiting at compile time, and so with high energy efficiency, instruction level parallelism (ILP) [2], [3].

System simulation, either to take design decisions or to develop compilers or applications, is required before actual hardware availability. Therefore, and independently of the level of abstraction of the simulation, efficient instruction set interpretation is a must. Among the available MPSoC simulation solutions (see [4] for a brief overview), dynamic binary translation (DBT) is very efficient [5], [6]. DBT has been introduced in the mid 80s [7], and its first use for simulation was reported in [8]. It is nowadays largely used, for example in virtualization technology for general purpose processors that are not virtualizable [9]. However, to the best of our knowledge, no solution for the dynamic translation of VLIW codes on scalar architectures has been detailed in the literature.

Even though initially driven by system simulation requirements, this paper actually deals with the general problem of dynamically translating code that features explicit ILP, statically defined non unit latency register updates and non immediate control flow changes. Our main contributions are as follows.

1) The definition of an instruction interpretation algorithm for VLIW processors on a scalar processor that deals with the above mentioned peculiarities.
2) A refined version of the algorithm suited for use in a dynamic binary translator and its actual implementation.

II. Related Works

Binary translation aims at transforming instructions of one instruction set architecture (ISA) to another [10]. This process can be executed at two different instants: 1) offline, we then talk of static binary translation [11], [12] or 2) at run-time, we then talk of DBT [13].

The initial and primary purpose of binary translation is to exploit new features of more recent versions of an architecture, for example, UQBT with the SPARC ISA [11] or Transmeta with their *code-morphing* [14] hardware technology targeting the x86 ISA. Others examples of solutions for facilitating migration from an architecture to another are [12], [15], and [16].

Rapidly, more applications of this technique have been though of, e.g., just-in-time compilation for Java, native binary acceleration [17], or virtualization and system simulation [18], [19]. More recently, interpretive languages have benefited from these techniques (Javascript, Python, Ruby, etc.).

In the DBT field, efforts have been made to build machine adaptable binary translators, to ease adaptation to numerous architectures with reduced effort [13], [19], [20]. All these translators use an intermediate representation (IR) so that the complete process of binary translation occurs in two phases, as illustrated in Fig. 1. The IR is machine independent to guaranty

Manuscript received April 4, 2016; revised June 28, 2016; accepted July 31, 2016. Date of publication August 30, 2016; date of current version April 19, 2017. This work was supported by the French Ministry of Industry through the BGLE ACOSE Project Grant. This paper was recommended by Associate Editor T. Mitra.

The authors are with the TIMA Laboratory, Université Grenoble Alpes, 38031 Grenoble, France (e-mail: frederic.petrot@imag.fr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2016.2604294

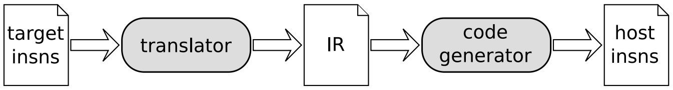


Fig. 1. Machine adaptable DBT process.

the retargetability of the translator, and can be viewed as a generic machine instruction set.

The first step translates a target instruction into machine independent IR. Code generation is then in charge of encoding the IR into host instructions.

Many of the previous works have addressed the issue of translating binaries from scalar target architecture to scalar host architecture. This case is straightforward, since by falling back in an elementary IR, one can always find a valid translation for instructions regardless of their complexity [19].

A few works have focused on the efficient translation of single instruction multiple data (SIMD) instructions. Among them we can cite FX!32 [12] which translates multimedia extensions (MMX) SIMD instructions to the Alpha instruction set, [21] which maps MMX/streaming SIMD extensions SIMD instructions on an IA64, and [22] which proposes a general approach demonstrated on ARM Neon extensions. Other works have focused on translations from scalar architectures to VLIW/EPIC architectures, such as DAISY [15], IA-32 EL for Itanium [23], or Aries for IA64 [16]. These solutions are more complex than the previous ones since on top of finding a matching between instructions, they have to extract ILP to take advantage of the host architecture.

The use of DBT as an instruction set simulation tool for system level simulation has been introduced only recently [24], [25]. Translating VLIW code into sequential code is of low practical interest outside of this topic (besides supporting legacy code, but VLIW processors are quite specialized and do not have a large code base). This explains why, to the best of our knowledge, the DBT of VLIW code into scalar code has not been researched as of today.

III. Background

We now review the background technologies on which this paper is based.

A. VLIW Architectures

VLIW architectures, as opposed to scalar ones, require the expression of ILP explicitly and make use of differentiated functional units to execute these instructions concurrently [26]. A processor able to execute n independent instructions at the same cycle is called a n-issue VLIW. Among these n instructions, some may indicate that a functional unit is idle if the compiler was not able to extract enough ILP. The binary code may either include these no-operations or use, for code size reasons, a stop bit defining whether the current instruction has to be executed in parallel with the previous one or not. In this latter case, the instruction must indicate the functional unit it uses, even though a given operation has the same behavior on all functional units that support it. A set of parallel instructions is called an execute packet (EP).

```
   L   c6x code
1  0       add    .L1   a0, a1, a2
2  0   ||  sub    .D1   a1, a4, a3
3  3   ||  mpy32  .M1   a2, a1, a3
4  0       add    .L1   a2, a3, a4
5          nop
6          nop
7  0       add    .L1   a2, a3, a5
```

(a)

```
   L   c6x code
1  0    [a0] add  .L1   a2, a3, a1
2  0   [!b1] sub  .S2   b1, b3, b0
```

(b)

```
    L   c6x code
1   5     b     .S1   lbl1 ; branch to lbl1
2         nop
3   5     b     .S1   lbl2 ; branch to lbl2
4         nop      3 ; nop for 3 cycles
5         nop        ; branch to lbl1 taken
6         ...
7       lbl1:
8   0     add   .L1   a2, a3, a4 ;executed
9         nop                    ; executed
10      ; Branch to lbl2 occurs
11  0     add   .L1   a2, a5, a6 ;not executed
12        ...
13      lbl2:
14        ...
```

(c)

Fig. 2. `c6x` code example. (a) Execution packets and instr. latency. (b) Predicated instructions. (c) Branch inside branch.

VLIW architectures, as originally defined, have the very important property of having neither stalls nor bypasses in the pipeline to handle hazards, even though some instructions stay several cycles in the execution pipeline of their own functional unit. As a consequence, compilers have to take care of the latency of each individual instruction as the value available in a register depends on the cycle at which it is read. The whole pipeline still stalls on a cache miss, it is then the responsibility of the hardware to ensure that the software visible registers updates will respect the specified latency, regardless of the duration of the stall.

To illustrate the concepts, we use the TMS320C64X (called `c6x` from now on), a rather complex VLIW DSP from Texas Instruments. However, the principles that we develop can be applied to any VLIW processor, regardless of its number of issues, maximum instruction latency, and so on. On the `c6x`, an EP is at most 256 bits wide, and thus contains up to eight 32 bits wide instructions. The `c6x` instruction syntax is of the form `op .unit src1, src2, dst`, where `op` is the operation to perform on functional unit `unit` taking `src1` and `src2` as input operands, and producing the result in `dst`. An instruction being prefixed by `||` is in the same EP as the previous one.

Lines 1–3 of Fig. 2(a) show an EP of size 3. The value of `a2` for the `mpy32` line 3 is its value before the execution of the `add` line 1. Column "L" indicates the instruction latency.

Regarding the arithmetic operation latencies, the data-sheet of the c6x indicates that the mpy32 32 bits multiplication instruction has a latency of 3. It means that the result of the instruction will be available only at the fourth cycle after the instruction execution, and access to the destination register in between will return its "old" value. In Fig. 2(a) at lines 2 and 3, a sub and a mpy32 instruction from the same execution packet write the same destination register a3. The sub having a zero cycle latency, its result is available in a3 at the next cycle. The mpy32 having a latency of 3, its result will be available at line 7, hence overwriting the sub result.

To avoid the use of branches preventing the execution of simple code depending on a condition, most c6x instructions can be predicated. In Fig. 2(b), the add is executed only if a0 is not equal to 0, and the sub if b1 is equal to 0.

Branch instructions have five delay slots, which means that the branch is eventually taken only six cycles after the effective branch instruction. From a general point of view, everything between the branch and the sixth instruction after it will be executed. In case, we have an instruction in the delay slot of the branch which also has a delay slot and for which the branch occurs before the end of this delay slot, nothing particular happens, and the instruction will continue its execution. The same rule applies for a branch inside a branch. Fig. 2(c) illustrates this case: a second branch occurs at line 3, but the first one at line 1 is not taken yet. It is eventually taken at line 7 and the add instruction of line 11 is executed, but the one at line 12 is not since the second branch occurs, and the execution continues at lbl2.

As can be seen through these examples, the c6x implements many VLIW architectural singularities. It is therefore an interesting but challenging target for efficient simulation approaches.

B. Dynamic Binary Translation

The basic principle of DBT consists of fetching each target software instruction at run time, and translating it into one or multiple host instructions. For scalar to scalar translation, each instruction is considered independently, and avoids side effects as much as possible. For scalar to VLIW translation, the instructions are added in order in the VLIW instruction with a register renaming strategy to exploit ILP whenever possible while keeping translation time short [15]. To do this wisely, the target code is split in translation blocks (TBs),[1] i.e., sequential sequence of code ended by a branch instruction. Each target block is translated into host code and then executed until a block boundary is reached, in which case the simulator translates a new block, corresponding to the current value of the simulated CPU program counter. The main advantage of this strategy over instruction interpretation is the possibility to cache the translated blocks. In case a block is encountered several times, the translation occurs only the first time, and the translated code is executed directly for the other occurrences.

In order to have a certain level of genericity and limit the number of translators, many DBT environments rely on an IR that is basically a simple bytecode. The translation mechanism occurs in two phases. The target code is first translated into bytecode, so writing a translator has to be done once for each target processor. Then a backend emits the code to run on the host, so this backend has to be developed once for each host.

c6x code	QEMU micro-operations	Simplified notation
`add .L1 a2, a1, a2`	`mov_i32 tmp0, a1`	`add_i32 a2, a2, a1`
	`mov_i32 tmp1, a2`	
	`add_i32 tmp1, tmp1, tmp0`	
	`mov_i32 a2, tmp1`	

Fig. 3. Complete translation of a target instruction into micro-operations.

Algorithm 1 Classical DBT Frontend

```
while not end of translation block do
    opcode ← get_next_instruction()        ▷ Fetch
    dec ← decode(opcode)                   ▷ Decode
    translate(dec)                         ▷ Translate
end while
```

We use QEMU [19] IR to illustrate the DBT concepts in the rest of the paper. First, QEMU uses a simple bytecode. Second, QEMU allows calling C functions, known as *helpers*, to perform complex operations from the generated code at execution time. These *ad-hoc* functions can be seen as pretranslated code, which perform a specific run-time task, like virtual memory management, SIMD instructions execution, etc. These two features are mandatory for retargetable DBT engines, which makes our solution generalizable to other translators. For the sake of clarity, QEMU IR listings of this paper are simplified. During translation, QEMU does not work directly on target registers. It uses intermediates values called *temporaries*. As an example, the complete translation of the first instruction is given in Fig. 3. The simplified notation consists of working directly on target registers. It allows removing micro-operations corresponding to data movement into and from temporaries.

IV. VLIW Architecture Simulation With DBT

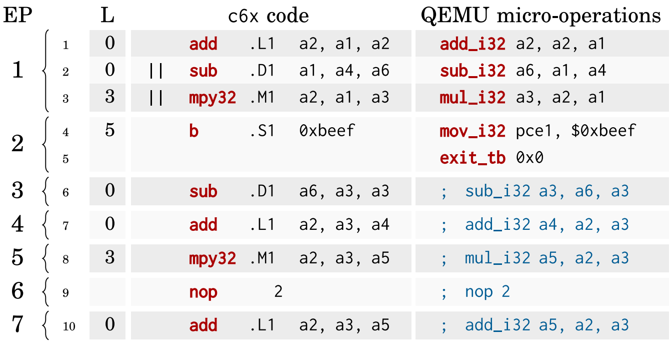
The classical DBT algorithm can be described in Algorithm 1.

This algorithm does not take into account the VLIW specificities. Notions like EPs or instruction latencies do not appear here. Fig. 4 gives the translation of some c6x assembly code resulting from this algorithm. In this listing, each c6x instruction is matched with its translation into QEMU IR micro-operations.

Fig. 4 exhibits the issues of the classical DBT algorithm applied to a VLIW target. Three distinct problems are as follows.

1) Instructions 1–3 belong to the same EP, so they are executed in parallel on the target CPU. However, the translator produces scalar code for the host, therefore some write-after-read (WAR) dependencies are not honored. For example, the first instruction writes its result into register a2, which is also read by instruction 3. But instruction 3 here should read the old value of a2, and not the result of instruction 1, because this

[1] Not to be confused with basic blocks as defined in compilation, as no instruction in a basic block but the first one can be the target of a branch.



EP		L	c6x code			QEMU micro-operations
1	1	0	add	.L1	a2, a1, a2	add_i32 a2, a2, a1
	2	0	|| sub	.D1	a1, a4, a6	sub_i32 a6, a1, a4
	3	3	|| mpy32	.M1	a2, a1, a3	mul_i32 a3, a2, a1
2	4	5	b	.S1	0xbeef	mov_i32 pce1, $0xbeef
	5					exit_tb 0x0
3	6	0	sub	.D1	a6, a3, a3	; sub_i32 a3, a6, a3
4	7	0	add	.L1	a2, a3, a4	; add_i32 a4, a2, a3
5	8	3	mpy32	.M1	a2, a3, a5	; mul_i32 a5, a2, a3
6	9		nop		2	; nop 2
7	10	0	add	.L1	a2, a3, a5	; add_i32 a5, a2, a3

Fig. 4. Naïve translation of some c6x code using the classical DBT algorithm.

value has already been overwritten when instruction 3 is simulated, the WAR dependency is violated.

2) A second problem arises because instructions latencies are ignored. Instruction 3 has a tree cycles delay. Its result in a3 should be visible from instruction 7. However, the generated code ignores this latency and overwrites directly the content of a3.

3) Finally, the branch instruction at line 4 is not translated correctly. The TB ends immediately after the simulation of this instruction, and does not respect the 5 cycles branch delay slot of the c6x architecture. Lines 6–10 are thus not translated (the corresponding micro-operations are given for illustrative purposes as comments in the listing).

A. Algorithm for VLIW Architecture Simulation

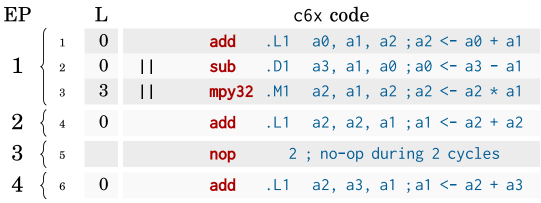
We propose a simulation algorithm able to tackle VLIW architecture specificities. As a first step, this algorithm is generic enough to be used in different kinds of simulators, not only in DBT-based ones. The main idea is to use multiple memory locations for the same target register. Each time a register is written, a location, called *replicate* hereafter, is created to store the new value. This strategy, which modelizes in an abstract way the registers of the pipeline, borrows the single assignment idea from scalar expansion [27] and static single assignment (SSA) [28].

During simulation, the simulator knows which replicate it must use to read the current value of a register. We call it the *current replicate* of this register. It also knows when it has to change the current replicate. Algorithm 2, in which EPs are processed independently, illustrates this principle. For each instruction in the current EP, a new replicate is created to store the instruction result, using its latency and output operand, thus indicating the proper time for the update of the current replicate. Then, it simulates the instruction and uses the new replicate to store the instruction result. Finally, when all instructions of the EP have been processed, it updates the current replicate of each register, if needed. During this step, the current replicate might change to reflect the fact that an instruction result is now available.

We illustrate how Algorithm 2 behaves using the c6x instructions of Fig. 5, assuming a queue is used to manage the replicates.

Algorithm 2 Register Replicates for VLIW Simulation

```
while not end of the simulation do
    ep ← get_next_execute_packet()
    for all insn ∈ ep do
        d ← get_delay(insn)    ▷ Get instruction latency
        output_op ← get_output_operand(insn)    ▷ Output operand read
        r ← new_replicate(output_op, d)    ▷ Replicate creation
        simulate(insn, r)    ▷ Instruction simulation using the new replicate
    end for
    do_end_of_cycle_updates()    ▷ Current replicate updates
end while
```



EP		L	c6x code			
1	1	0	add	.L1	a0, a1, a2	; a2 <- a0 + a1
	2	0	|| sub	.D1	a3, a1, a0	; a0 <- a3 - a1
	3	3	|| mpy32	.M1	a2, a1, a2	; a2 <- a2 * a1
2	4	0	add	.L1	a2, a2, a1	; a1 <- a2 + a2
3	5		nop		2	; no-op during 2 cycles
4	6	0	add	.L1	a2, a3, a1	; a1 <- a2 + a3

Fig. 5. c6x example code.

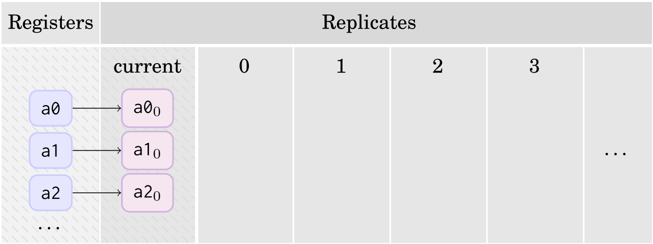


Fig. 6. Initial state.

1) Initial State: The simulator initial state is given in Fig. 6. Each register ai has its initial current replicate ai_0, used when reading it. The waiting queue is empty since no instruction has been simulated yet, thus no new replicates have been created.

2) After the First EP: After the first EP execution, (instructions 1–3 of Fig. 5), three replicates have been created, one for a0 and two for a2. The corresponding state is given in Fig. 7. The same register can be the output operand of two instructions of the same EP, as long as these instructions have different latencies. Replicates $a0_1$ and $a2_1$ are put in cell 0 of the waiting queue since instructions 1 and 2 have zero latency, while replicate $a2_2$ is put in cell 3 since the multiply instruction of line 3 has a 3 cycles latency. This explains why the latency information is a parameter of new_replicate(). At this precise time, the do_end_of_cycle_updates() function has not been called yet.

3) After the Current Replicates Update: After the call to do_end_of_cycle_updates(), the waiting queue is shifted by one. Replicates $a0_1$ and $a2_1$ become the current

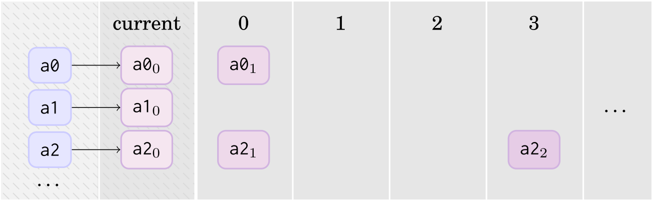


Fig. 7. State after simulating the first EP.

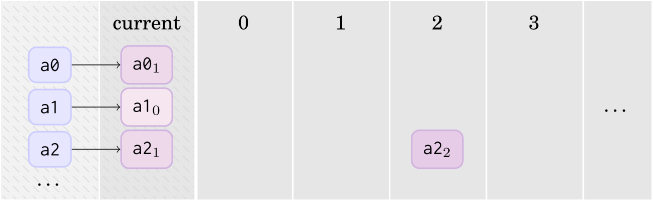


Fig. 8. State after the current replicates update.

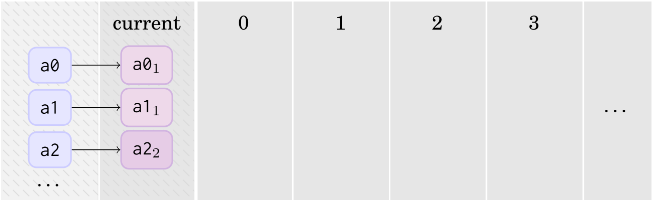


Fig. 9. State after simulation the `nop 2` instruction and updating the current replicates update.

replicates for their corresponding register. The old replicates are discarded since they are now deprecated. The replicate a2$_2$ containing the multiply result has taken one step forward in the waiting queue. The corresponding state is given in Fig. 8.

4) After EPs 2 and 3: After the simulation of EPs 2 and 3, and the current replicates update, the multiply result is available in `a2` since 3 cycles have passed. The `add` instruction of line 6 will read this result, not the older value. This can be seen in Fig. 9 with the use of a2$_2$ as the current replicate for register `a2`.

This algorithm allows to simulate the peculiarities of the VLIW architecture, even though the instructions are simulated sequentially. It draws inspiration from SSA since a replicate can be written only once, however, multiple replicates of the same register can be alive at the same time, which is not possible with the SSA form.

B. Algorithm Specialization for DBT

Using this algorithm in a DBT-based simulator requires managing the peculiarities of this simulation technique. To illustrate the key challenges, we use as example the implementation of a `c6x` frontend inside QEMU.

Adaptation of Algorithm 2 to DBT is given by Algorithm 3. The main difference with classical simulators is that this algorithm is executed during translation, in the DBT frontend. During this phase, the execution context (containing the simulated processor state) is not known. Indeed, a TB is generated regardless of any context. It is independent of other TBs and makes no assumption about its predecessors. This property is key in DBT. It allows reusing already translated TBs multiple times in any contexts.

Algorithm 3 Specialized Version for Use in a DBT Frontend

```
while not end of the translation block do
    ep ← get_next_execute_packet()
    for all insn ∈ ep do
        d ← get_delay(insn)  ▷ Instruction latency read
        output_op ← get_output_operand(insn)  ▷ Output operand read
        r ← new_replicate(output_op, d)  ▷ Replicate creation
        translate(insn, r)  ▷ Instruction translation using the new replicate
    end for
    do_end_of_cycle_updates()  ▷ Current replicate updates
end while
```

1) Efficient Replicates Allocation: When QEMU simulates a processor, it maintains its execution context into a structure called *environment*. Its content is defined by the designer of the frontend, and contains, among others, the simulated processor architectural registers (the general purpose registers `ai` and `bi` we have already seen, and `pce1` which is the program counter). Since this structure is fixed during execution, Algorithm 3 seems not well suited as new replicates are created each time a register is written. However, these replicates have a short lifespan that cannot exceed the longest latency p among all instructions. In the `c6x` instruction set, $p = 5$ cycles for the branch instructions. For a given register, the number of replicates alive at time t can be bound. At worst, the current EP contains an instruction with latency p, another with latency $p - 1$, $p - 2$, and so on until 0, all these instructions writing the same register r. In this case, we need the current replicate for r, plus p replicates to store all the pending results, namely, $p + 1$ replicates alive for r at the same time. Another way to reach this worst case is to have p EP containing each an instruction of latency p writing the same register.

Since we know the upper bound of the number of replicates that can be alive at the same time for a given register, we can use a circular buffer to store them in the environment. This buffer size is $n \times (p + 1)$ with n the number of target registers. During translation, the simulator uses the replicates of this buffer as operands for the micro-operations. Fig. 10 gives an example of a correct translation. A new replicate is used on each register write and instruction latencies are correctly handled. For example, the `mpy32` instruction line 3 having a 3 cycles latency, its destination replicate is not used as the current replicate until the micro-operation of line 7. Moreover, the branch instruction of line 4 is correctly translated since the jump address computation is done at line 4 when the `b` instruction occurs. But the TB is ended at line 9 when 5 cycles have elapsed (the instruction line 10 is thus not translated).

2) Problems Induced by TB Granularity: In the TB translation context, this algorithm produces code with correct

EP		L	c6x code	QEMU micro-operations
1	1	0	add .L1 a2, a1, a2	add_i32 a2_1, a2_0, a1_0
	2	0	|| sub .D1 a1, a4, a6	sub_i32 a6_1, a1_0, a4_0
	3	3	|| mpy32 .M1 a2, a1, a3	mul_i32 a3_1, a2_0, a1_0
2	4	5	b .S1 0xbeef	mov_i32 pce1_1, `$0xbeef`
3	5	0	sub .D1 a6, a3, a3	sub_i32 a3_2, a6_1, a3_0
4	6	0	add .L1 a2, a3, a4	add_i32 a4_1, a2_1, a3_2
5	7	3	mpy32 .M1 a2, a3, a5	mul_i32 a5_1, a2_1, a3_1
6	8		nop 2	; nop2
	9			exit_tb 0x0
7	10	0	add .L1 a2, a3, a5	; add_i32 a5_2, a2_1, a3_1

Fig. 10. Correct translation of some c6x code.

EP		L	c6x code	QEMU micro-operations
1	1	0	add .L1 a2, a1, a2	add_i32 a2_1, a2_0, a1_0
	2	0	|| sub .D1 a1, a4, a6	sub_i32 a6_1, a1_0, a4_0
	3	3	|| mpy32 .M1 a2, a1, a3	mul_i32 a3_1, a2_0, a1_0
2	4	5	b .S1 0xbeef	mov_i32 pce1_1, `$0xbeef`
3	5	0	sub .D1 a6, a3, a3	sub_i32 a3_2, a6_1, a3_0
4	6	0	add .L1 a2, a3, a4	add_i32 a4_1, a2_1, a3_2
5	7	3	mpy32 .M1 a2, a3, a5	mul_i32 a5_1, a2_1, a3_1
6	8		nop 2	; nop 2
7	9		; TB epilogue	; Canonical state return
	10			mov_i32 a2_0, a2_1
	11			mov_i32 a3_0, a3_1
	12			mov_i32 a4_0, a4_1
	13			mov_i32 a6_0, a6_1
	14			mov_i32 pce1_0, pce1_1
	15			exit_tb 0x0

Fig. 11. Canonical state return.

simulation behavior. However, the TB translation granularity imposed by DBT brings up new problems.

a) Canonical state between TBs: As described previously, the replicate allocation is done statically during translation. Consequently, the register → replicate matching is not part of the simulated processor state and thus is not known at execution time. Moreover, every TB is translated independently of each other to ensure reusability. When the TB given in Fig. 10 ends, current replicates for register a2 and a6 are replicates number one (ai_1). But this information is not available for the next TB, it is thus necessary to introduce a *canonical* state between TBs. For each register, this state corresponds to the use of replicate 0 (ai_0) as the current replicate. To build this canonical state, at the end of each TB the translator generates copies for all registers that do not use replicate 0 from the current replicate to replicate 0. Fig. 11 completes the previous example with lines 10–14, which make registers a2, a3, a4, and a6 return to canonical state. This way, TBs independence is guaranteed. They all begin and finish in the same state with regard to the current replicates.

b) Necessary context saves at execution time: In the example of Fig. 11, the mpy32 instruction at line 7 writes its result into register a5. A new replicate a5_1 is thus allocated for this result. But in canonical state return, there is no copy for register a5. This is the expected behavior since the mpy32 instruction delay is not over yet. If a5_1 was copied to a5_0, the next TB would see the mpy32 result one cycle too early.

EP		L	c6x code	QEMU micro-operations
1	1	0	add .L1 a2, a1, a2	add_i32 a2_1, a2_0, a1_0
	2	0	|| sub .D1 a1, a4, a6	sub_i32 a6_1, a1_0, a4_0
	3	3	|| mpy32 .M1 a2, a1, a3	mul_i32 a3_1, a2_0, a1_0
2	4	5	b .S1 0xbeef	mov_i32 pce1_1, `$0xbeef`
3	5	0	sub .D1 a6, a3, a3	sub_i32 a3_2, a6_1, a3_0
4	6	0	add .L1 a2, a3, a4	add_i32 a4_1, a2_1, a3_2
5	7	3	mpy32 .M1 a2, a3, a5	mul_i32 a5_1, a2_1, a3_1
6	8		nop 2	; nop 2
7	9		; TB epilogue	; Context save
	10			call save_context, 5, a5_1, 0
	11			; Canonical state return
	12			mov_i32 a2_0, a2_1
	13			mov_i32 a3_0, a3_1
	14			mov_i32 a4_0, a4_1
	15			mov_i32 a6_0, a6_1
	16			mov_i32 pce1_0, pce1_1
	17			exit_tb 0x0

Fig. 12. Context save.

However, information on the circular buffer being known at translation time only, the next TB cannot know that there is an ongoing replicate for a5. Thus, a5_1 will be considered as nonallocated during translation of the next TB and the mpy32 result will be lost. This problem arises when there are still replicates in the waiting queue at the end of the TB. To overcome this problem while keeping inter-TB independence, we must introduce a runtime mechanism that saves and restores the state of the waiting queue *between* TBs. This mechanism modifies the simulated processor state as the information must be alive between TBs during execution. We call this mechanism the *dynamic waiting queue* since it takes place during execution, contrary to the previously introduced *static* version which is used during translation.

We chose to implement this mechanism with helpers. The context save helper prototype is as follows: **void** HELPER(save_context)(**uint32_t** reg, **uint32_t** val, **uint32_t** delay).

Its parameters are the register ID that must be modified, the value to apply to it, and the number of cycles before the modification happens. It saves this information into the dynamic waiting queue inside the environment. This queue is distinct from the static one used during translation since it does not store information about which replicate to use for a register, but the *value* to write into it. Indeed, this process takes place at execution time, where the correspondence register → replicate does not exist, but where we know the actual values of the registers. Fig. 12 completes the previous example by adding a context save at line 10. Since the mpy32 instruction result is still in the static waiting queue at the end of the TB, the translator generates this context save. This save operation applies to register number 5 (a5), the value to write in it is contained in a5_1 and the number of cycles before applying this value is 0 (hence one EP).

c) Context restore: The symmetrical operation to context save at TB end is context restore at TB start. As before, we use a helper to implement it. However, context restore brings up new difficulties. The restore_context helper has no parameters, it uses the dynamic waiting queue state to know which modifications to apply at the current cycle. A call to

EP		L	c6x code	QEMU micro-operations
1	1	0	add .L1 a2, a1, a2	add_i32 a2_1, a2_0, a1_0
	2	0	|| sub .D1 a1, a4, a6	sub_i32 a6_1, a1_0, a4_0
	3	3	|| mpy32 .M1 a2, a1, a3	mul_i32 a3_1, a2_0, a1_0
	4			mov_i32 a6_0, a6_1 ;Canonical state
	5			mov_i32 a2_0, a2_1 ;Canonical state
	6			call restore_context ;Ctx restore
2	7	5	b .S1 0xbeef	mov_i32 pce1_1, $0xbeef
	8			call restore_context ;Ctx restore
3	9	0	sub .D1 a6, a3, a3	sub_i32 a3_2, a6_1, a3_0
	10			mov_i32 a3_0, a3_2 ;Canonical state
	11			call restore_context ;Ctx restore
4	12	0	add .L1 a2, a3, a4	add_i32 a4_1, a2_1, a3_2
	13			mov_i32 a4_0, a4_1 ;Canonical state
	14			mov_i32 a3_0, a3_1 ;Canonical state
	15			call restore_context ;Ctx restore
5	16	3	mpy32 .M1 a2, a3, a5	mul_i32 a5_1, a2_1, a3_1
	17			call restore_context ;Ctx restore
6	18		nop 2	; nop 2
7	19		; TB epilogue	; Context save
	20			call save_context, 5, a5_1, 0
	21			; Canonical state return
	22			mov_i32 a2_0, a2_1
	23			mov_i32 a3_0, a3_1
	24			mov_i32 a4_0, a4_1
	25			mov_i32 a6_0, a6_1
	26			mov_i32 pce1_0, pce1_1
	27			exit_tb 0x0

Fig. 13. Context restore.

EP		L	c6x code	QEMU micro-operations
1	1	0	add .L1 a2, a1, a2	add_i32 a2_1, a2_0, a1_0
	2	0	|| sub .D1 a1, a4, a6	sub_i32 a6_1, a1_0, a4_0
	3	3	|| mpy32 .M1 a2, a1, a3	mul_i32 a3_1, a2_0, a1_0
	4			call save_ctx_prolog, 3, a3_1, 3
	5			mov_i32 a6_0, a6_1 ;Canonical state
	6			mov_i32 a2_0, a2_1 ;Canonical state
	7			call restore_context ;Ctx restore
2	8	5	b .S1 0xbeef	mov_i32 pce1_1, $0xbeef
	9			call save_ctx_prolog,"p1",pce1_1,5
	10			call restore_context ;Ctx restore
3	11	0	sub .D1 a6, a3, a3	sub_i32 a3_2, a6_1, a3_0
	12			mov_i32 a3_0, a3_2 ;Canonical state
	13			call restore_context ;Ctx restore
4	14	0	add .L1 a2, a3, a4	add_i32 a4_1, a2_1, a3_2
	15			mov_i32 a4_0, a4_1 ;Canonical state
	16			mov_i32 a3_0, a3_1 ;Canonical state
	17			call restore_context ;Ctx restore
5	18	3	mpy32 .M1 a2, a3, a5	mul_i32 a5_1, a2_1, a3_1
	19			call save_ctx_prolog, 5, a5_1, 3
	20			call restore_context ;Ctx restore
6	21		nop 2	; nop 2
7	22		; TB epilogue	; Context save
	23			call save_context, 5, a5_1, 0
	24			; Canonical state return
	25			mov_i32 a2_0, a2_1
	26			mov_i32 a3_0, a3_1
	27			mov_i32 a4_0, a4_1
	28			mov_i32 a6_0, a6_1
	29			mov_i32 pce1_0, pce1_1
	30			exit_tb 0x0

Fig. 14. Context save during the first p cycles.

this helper must be interleaved with execution of each EP, at the end of which it is called to check if there is a value to apply to a register.

First, we need to know how many times this helper must be called to ensure no more modifications await in the dynamic waiting queue. During context save, the biggest `delay` that can introduced is $p - 1$, so we will need at most p cycles (from 0 to $p - 1$) to see the associated `value` applied to its destination register. As a consequence, we must call this helper p times, during the p first cycles of the TB.

Second, since we do not know the register → replicate matching at execution time, the helper ignores which replicate the value should be written into. This problem is solved once again using the canonical state. For each modified register, at the end of each EP, the translator copies the value of the current replicate back into replicate 0 and makes it the current replicate. The helper can then safely apply the modifications expected at the current cycle to replicate 0 of the registers. Fig. 13 gives the code generated with this strategy. At lines 6, 8, 11, 15, and 17, we can see the five calls to the context restore helper. During these first 5 cycles, we can also observe the canonical state preservation. Lines 4 and 5 put the results of instructions at lines 1 and 2 into replicate 0 of their respective destinations. This is also the case for instruction at line 3, but at line 14 only since it has a latency of 3 cycles.

d) Branch instructions special case: Thanks to this save and restore mechanism, inter-TBs latencies are correctly handled. However, the branch instructions still need some attention. In DBT, a branch ends a TB. If a branch is still in the waiting queue at the end of a TB, it means that the next TB will have to end a few cycles after it started. Since during translation, we do not know if a branch will be waiting in the queue at the start of a TB, we cannot generate code to end the TB. QEMU allows to dynamically end a TB thanks to a call to a dedicated function (`cpu_loop_exit`). The context restore helper can call this function to stop TB execution, regardless of the generated code. In this case, we must ensure that we are in a coherent state before leaving the TB. Since this mechanism occurs during the first p cycles, we can use the canonical state to make sure that the next TB will read the correct register values.

Unfortunately, we still have a problem with values in the waiting queue. Let us take the example of the `mpy32` instruction (see line 3 in Fig. 13). If a branch in the dynamic waiting queue is restored just after the `mpy32`, for example at line 6 during context restore, then the `mpy32` result in `a3_1` will be lost. This kind of events is not predictable at translation time, so we must generate a context save for this instruction result. This context save is special because it happens during the first p cycles of the TB. We cannot use the `save_context` helper here because it writes unconditionally the modification information into the dynamic waiting queue. In our case, it is only correct to write it if there is a pending branch that will make the simulator leaves this TB dynamically. Otherwise, the continuation of the TB will handle it normally. This is why we use another helper, `save_ctx_prolog`, which checks the existence of a branch into the dynamic waiting queue before inserting the modification into it. Fig. 14 depicts the final and functional version of the generated code. We can see at lines 4, 9, and 19 the calls to the `save_ctx_prolog` helper, for the three instructions having a nonzero latency.

3) Predicated Instructions Handling: We can generate `if...then` like control structures to simulate predicated instructions. However, as the conditions outcomes are not

EP		L	c6x code	QEMU micro-operations
1	1	0	add .L1 a0, a1, a2	add_i32 a2_1, a0_0, a1_0
	2	0	||[a0] sub .D2 b3, b2, b4	brcond_i32 a0_0, $0x0, eq, lbl1
	3			sub_i32 b4_1, b3_0, b2_0
	4			lbl1:
	5	5	||[!b2] b .S2 some_label	brcond_i32 b2_0, $0x0, ne, lbl2
	6			mov_i32 pce1_1, some_label
	7			lbl2:
2	8	0	add .L2 b4, b4, b0	add_i32 b4_1, b4_1, b0_0

Fig. 15. Naïve translation of predicated instructions.

EP		L	c6x code	QEMU micro-operations
1	1	0	[a0] sub .D2 b3, b2, b4	brcond_i32 a0_0, $0x0, eq, lbl1
	2			sub_i32 b4_1, b3_0, b2_0
	3			br lbl2
	4			lbl1:
	5			mov_i32 b4_1, b4_0
	6			lbl2:

Fig. 16. Predicated instruction without latency translation.

known during translation, new problems with replicates allocation happen. If we naïvely translate the previous example, we obtain the micro-operations listed in Fig. 15. Micro-operations at lines 2 and 5 check the conditions. Line 2, a jump to lbl1 is taken if a0_0 is 0 (a0_0 eq $0x0). Line 5, the branch to lbl2 is taken if b2_0 is not 0 (b2_0 ne $0x0).

For each instruction result, we statically allocate a new replicate. Unfortunately, if the instruction at line 2 is not executed, the replicate b4_1 will not be initialized but will still be used by the instruction at line 8. The translator must also handle the case where the condition is false.

Multiple nontrivial cases can appear with predicated instructions translation.

1) When a predicated instruction has 0 cycle latency, as just explained.
2) When the predicated instruction has a nonzero latency.
3) When we have two predicated, parallel instructions, one having a condition that is the Boolean complement of the other, (for example [a0] and [!a0]), and the two instructions write to the same destination register. This is the only case in which two instructions can write to the same register at the same cycle, since we can be sure that there is exactly one of the conditions that holds.
4) When branch instructions are predicated.

a) Predicated instruction with zero latency: This is the case of the sub at line 2 of Fig. 15. The easier way to not interfere with the replicate static allocation algorithm is to copy back the value of the old replicate into the new one if the condition is false. Fig. 16 depicts this solution. Here, a if...then...else like control structure is generated. If the condition is not met, the value of b4_0 is propagated into b4_1.

b) Predicated instruction with nonzero latency: For an instruction with nonzero latency, the previous solution does not work. Fig. 17 highlights this problem by inserting an instruction in the delay slot of the mpy32 instruction at line 1. The add instruction at line 7 induces the a1_2 replicate allocation for register a1. After the mpy32 delay slot, the current replicate for a1 is a1_1, which contains a deprecated value if the condition is not met during execution. As a consequence, the add result will be lost.

The solution consists of moving the propagation taking place line 5 at the location, where the conditional instruction delay slot is ending. But to be able to do so, we must remember the condition outcome later in the generated code. To that aim, we introduce in the environment some condition flags (cf_x) that can be used by the translator to save the conditions outcomes. Fig. 18, cf_0 is written at lines 3 or 6, depending on the condition result. It is then checked at the end of the mpy32 delay slot at line 10. If the condition appears to be false, the value of the previous replicate is propagated into the new one (line 11).

EP		L	c6x code	QEMU micro-operations
1	1	3	[a0] mpy32 .M1 a3, a2, a1	brcond_i32 a0_0, $0x0, eq, lbl1
	2			mul_i32 a1_1, a3_0, a2_0
	3			br lbl2
	4			lbl1:
	5			mov_i32 a1_1, a1_0
	6			lbl2:
2	7	0	add .L1 a2, a2, a1	add_i32 a1_2, a2_0, a2_0
3	8		nop 2	; nop 2
4	9	0	add .L1 a1, a3, a2	add_i32 a2_1, a1_1, a3_0

Fig. 17. Wrong translation of a nonzero latency predicated instruction.

EP		L	c6x code	QEMU micro-operations
1	1	3	[a0] mpy32 .M1 a3, a2, a1	brcond_i32 a0_0, 0, eq, lbl1
	2			mul_i32 a1_1, a3_0, a2_0
	3			movi_i32 cf_0, 1
	4			br lbl2
	5			lbl1:
	6			movi_i32 cf_0, 0
	7			lbl2:
2	8	0	add .L1 a2, a2, a1	add_i32 a1_2, a2_0, a2_0
3	9		nop 2	; nop 2
	10			brcond_i32 cf_0, 0, ne, lbl3
	11			mov_i32 a1_1, a1_2
	12			lbl3:
4	13	0	add .L1 a1, a3, a2	add_i32 a2_1, a1_1, a3_0

Fig. 18. Correct translation of a nonzero latency predicated instruction.

EP		L	c6x code	QEMU micro-operations
1	1	0	[a0] sub .D2 b3, b2, b4	brcond_i32 a0_0, $0x0, eq, lbl1
	2			sub_i32 b4_1, b3_0, b2_0
	3			br lbl2
	4			lbl1:
	5			mov_i32 b4_1, b4_0
	6			lbl2:
	7	0	||[!a0] add .L2 b3, b2, b4	brcond_i32 a0_0, $0x0, ne, lbl3
	8			add_i32 b4_1, b3_0, b2_0
	9			lbl3:

Fig. 19. Two parallel instructions with complemented conditions.

c) Complemented conditions with same destination: This case, illustrated in Fig. 19, must be taken into account during translation. The sub instruction at line 1 is translated as explained before. However, for the add instruction at line 7, the translator realizes that there is already a waiting replicate for this register. It also knows that it comes from a predicated instruction having as condition the complement of the current one. In this case, it does not allocate a new replicate for the destination register, but reuses the already allocated one. Moreover, it does not generate the instruction that propagates the old value when the predicate is false since, when arriving at

EP		L	c6x code	QEMU micro-operations
1	1	5	[a0] b .S2 0x80ec	brcond_i32 a0_0, 0, eq, lbl1
	2			mov_i32 pce1_1, some_label
	3			movi_i32 cf_0, 1
	4			br lbl2
	5			lbl1:
	6			movi_i32 cf_0, 0
	7			lbl2:
2	8	0	add .L1 a0, a0, a0	add_i32 a0_1, a0_0, a0_0
3	9		nop 4	; nop 4
4	10		; Prologue TB	brcond_i32 cf_0, 0, eq, lbl3
	11			mov_i32 a0_0, a0_1
	12			mov_i32 pce1_0, pce1_1
	13			exit_tb $0x0
	14			lbl3:
	15			movi_i32 pce1_1, next_insn
	16			mov_i32 a0_0, a0_1
	17			mov_i32 pce1_0, pce1_1
	18			exit_tb $0x0

Fig. 20. Predicated branch translation.

that point, it can be sure that exactly one of the two predicated instructions will have been executed, so the current replicate will contain the correct value. Note that the propagation at line 5 is useless but generated, as the instructions are translated one by one.

d) Predicated branch instructions: Predicated branch instructions add the constraint of TB termination. From a general point of view, it is preferable to end the TB at the location, where the branch should be taken regardless of the predicate outcome. This avoids translating code after the branch that will possibly never be executed. Fig. 20 describes the solution. The TB prologue contains context saving and canonical state return for both the branch taken case (lines 11–13) and the branch not taken case (lines 15–18). The only difference between these two cases is that the next instruction address is copied to pce1 if the branch is not taken. This can be seen as redundant and suboptimal since the context save and canonical state return are the same for both cases. However, this separation is necessary to handle a special case described in Fig. 21. In this case, the last instruction is a nop that lasts for 4 cycles. If the branch is taken, the processor behavior here is to *stop* the nop instruction and thus cancel the 3 left cycles.

Our translation method allows handling this case correctly. If the branch is taken, the context save is done as if the nop instruction had lasted one cycle. We can see the dynamic context save at lines 14 and 15, with 0 cycle left for the mpy32 result and 2 cycles left for the ldw (memory load) instruction. If the branch is not taken, the 3 more cycles of the nop allow waiting replicates to be statically applied. Thus, there is no dynamic context saving but only canonical state return (lines 20–22).

4) Conclusion: All the techniques presented in this section make the VLIW simulation algorithm usable in a DBT context. The algorithm must be specialized by moving part of the waiting queue management at execution time. After handling all the corners cases due to the separation of the translation and execution phases, the simulator is able to reproduce the actual ISA behavior, at the cost of bigger generated code size.

EP		L	c6x code	QEMU micro-operations
1	1	5	[a0] b .S2 some_label	brcond_i32 a0_0, $0x0, eq, lbl1
	2			mov_i32 pce1_1, some_label
	3			movi_i32 cf_0, 1
	4			br lbl2
	5			lbl1:
	6			movi_i32 cf_0, 0
	7			lbl2:
2	8		nop 2	; nop 2
3	9	3	mpy32 .M1 a0, a1, a2	mul_i32 a2_1, a0_0, a1_0
4	10	4	ldw .D1 *+a0(0), a1	add_i32 tmp0, a0_0, $0x0
	11			qemu_ld32 a1_1, tmp0
5	12		nop 4	; nop 4
6	13		; Prologue TB	brcond_i32 cf_0, 0, eq, lbl3
	14			call save_ctx, 2, a2_1, 0
	15			call save_ctx, 1, a1_1, 2
	16			mov_i32 pce1_0, pce1_1
	17			exit_tb $0x0
	18			lbl3:
	19			movi_i32 pce1_1, next_insn
	20			mov_i32 a1_0, a1_1
	21			mov_i32 a2_0, a2_1
	22			mov_i32 pce1_0, pce1_1
	23			exit_tb $0x0

Fig. 21. Special case: a predicated branch with a nonzero latency nop at TB end.

V. Experimentations

We now present experimental results of our c6x translator implementation in QEMU. We compare ourselves against the official Texas Instrument simulator integrated into Code Composer Studio (CCS), which uses instruction interpretation-based simulation techniques. We choose this simulator because it provides a simple and widely available reference for both functional ISA implementation and simulation performance.

A. Experimental Protocol

Our implementation has been verified with unitary tests not presented here, then with simple programs written on purpose for a detailed study of code generation, and finally with all programs of a benchmark, to give results on many examples.

We use three simple tests with different characteristics parameterized with a value n to perform an in-depth analysis of the generated code.

fibo Computes the Fibonacci number for index n. The algorithm used here is a naïve version which realizes a high number of recursive calls [time complexity is $O(2^n)$].

matrix Computes the product of two square matrices of size n. This is also a naïve implementation that realizes a multiplication and an addition in a triple nested loop [time complexity is $O(n^3)$]. It executes a lot of control instructions and few arithmetic instructions.

idct Computes the inverse discrete cosine transform of n blocks of 8×8 coefficients using fixed point arithmetic [time complexity is $O(n)$]. It implements an optimized version of the algorithm (due to Loeffler) which does many arithmetic and logic instructions and a few control instructions, thus producing TBs with a lot of instructions.

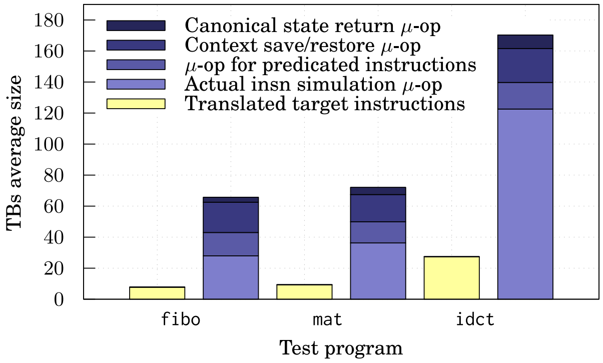


Fig. 22. Statistics on generated micro-operations for the three tests.

The second set of programs aims at showing the stability of the results on a broader base. It is the full Polybench test suite [29], composed of computation kernels that have potentially a high ILP.

All tests have been compiled with GCC 4.9.1 using maximum optimization (`-O3`) and executed on a machine with 8 Intel Xeon E5-2665 processors clocked at 2.40 GHz, and 256 GB memory. The parameter n is defined as a constant at compilation time, so that optimizations such as loop unrolling and complex instruction scheduling can be performed.

B. Detailed Analysis of the First Programs

1) Profiling of the Generated Code: To study the impact of our solution on the generated code, we have instrumented QEMU. For each TB, we gather the number of translated `c6x` target instructions and the number of generated micro-operations needed to handle.

1) Canonical state return.
2) Predicated instructions.
3) Context save and restore.
4) Simulation of the actual behavior of the target instructions.

Fig. 22 gives the average number of translated target instructions in a TB, and the average number of generated micro-operations, split using the above criteria, for our three tests programs. `Fibo` and `matrix` have small TBs on average because they include a large proportion of control instructions. Context save and restore represent an important part of the total micro-operations. Indeed, for TBs with an average of 7–9 target instructions, the fixed cost of context restore during the first 6 cycles is not amortized. However, the canonical state return part is negligible since the small number of target instructions imply few replicate allocations. This cost is about 3 micro-operations for `fibo` and 4 for `matrix`. For `idct`, the average TB size is about 30 translated target instructions which is way bigger. The micro-operations for actual instructions simulation ratio is much more important, with about 72% of the total micro-operations. The context save and restore and canonical state return costs are significantly amortized.

The micro-operations/target instructions ratio is about 8.45 for `fibo`, 7.74 for `matrix`, and 6.21 for `idct`, which confirms the previous observations.

2) Simulation Times as Function of n: To evaluate the simulation speed of our simulator, we measure the execution time of the three programs on our implementation and compare them against CCS execution time. Each performance test has been realized when both the simulators are initialized and ready to run. For QEMU, the measures are done around the main loop. For CCS, the whole Java virtual machine is already running and ready to simulate the first instruction. For each test, we extract the mean and standard deviation values of the execution time for 120 runs. It allows us to consider that the runs follow a normal distribution and makes it possible to compute a confidence interval for the mean.

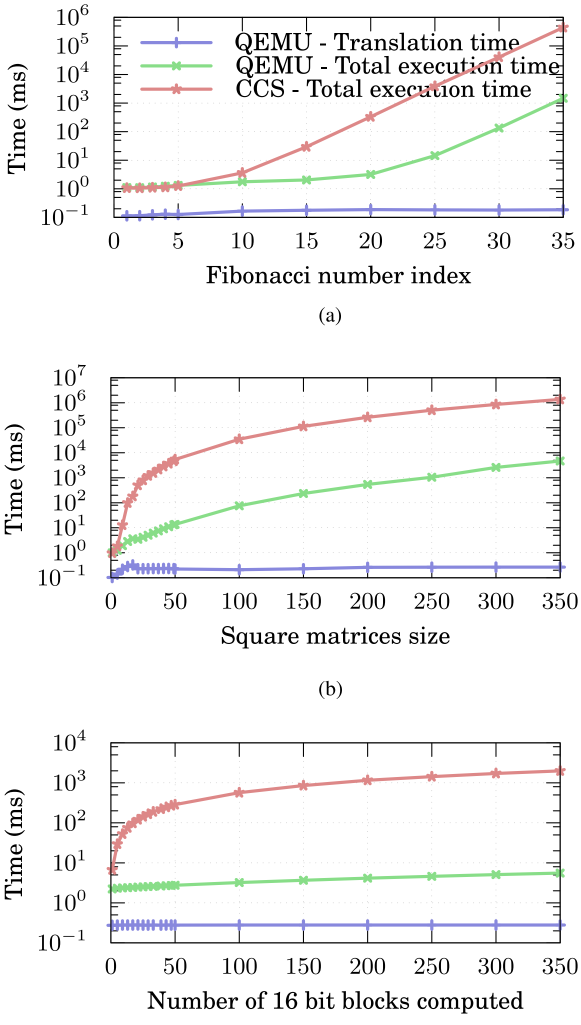
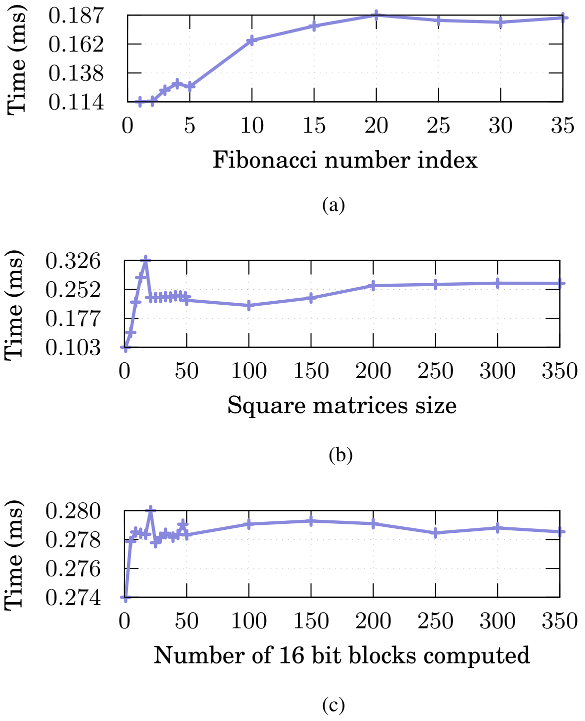


Fig. 23. Execution times of the simple programs as a function of their parameter. (a) `fibo` execution time. (b) `mat` execution time. (c) `idct` execution time.

Each of our simple programs has a parameter n that we change across the experiments. Note that n is fixed at compile time so the compiler can perform different optimizations according to its actual value. Fig. 23(a)–(c) give the results for the `fibo`, `matrix`, and `idct` programs, respectively.

In abscissa, we give the programs parameter value and in ordinate the execution time in ms, on a logarithmic scale. There is a dramatical advantage for DBT-based simulation compared to interpretation techniques. A two orders of magnitude difference appears for `fibo` at index 20 and before



(a)

(b)

(c)

Fig. 24. Translation times of the simple tests as a function of their parameter. (a) `fibo` translation time. (b) `mat` translation time. (c) `idct` translation time.

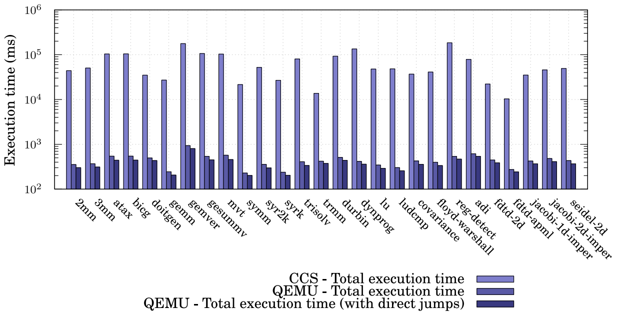


Fig. 25. Polybench execution times.

index 50 for `matrix` and `idct`. Fig. 24(a)–(c) plot the translation time using a linear scale for *y*. The variations are due to compiler optimizations that produce different versions of the binary for different values of *n*. However, asymptotically the translation time reaches a plateau, and the total execution time is only influenced by the number of times the TB are executed. Overall, the translation time is stable with respect to *n*, and stays negligible compared to the total execution time.

C. Results on the Polybench Benchmark

Polybench comes with predefined problem sizes. For each individual program, we choose the problem size so that the simulation time under CCS is less than 10 min.[2] Fig. 25 plots the run times of the programs for CCS, for our base DBT solution, and for a DBT solution implementing chaining [18]. In specific but frequent situations, this simple optimization links a TB to its successor with a direct branch micro-operation.

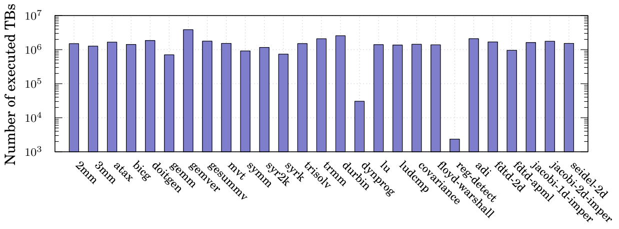


Fig. 26. Number of executed TBs.

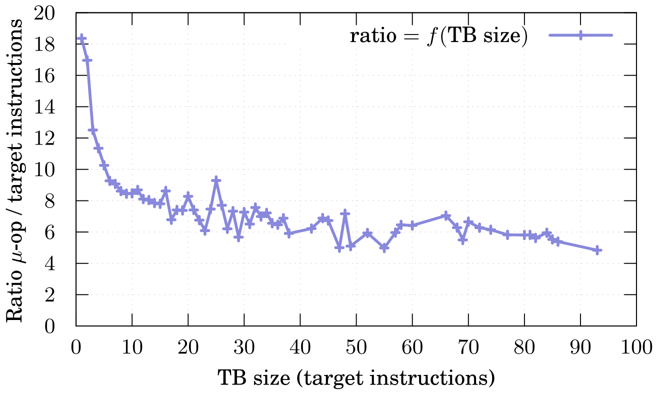


Fig. 27. Evolution of the average TB size ratio given its target instructions size.

As can be seen, our base solution is in average around two orders of magnitude faster than CCS, with a minimum of 32× for trmm and a maximum of 343× for reg-detect. Regarding the optimized version, TBs can be chained in average 35%–40% of the time. While performing this optimization slows down the translation process imperceptibly (it is made 1.04 times slower in average), it leads to a significant mean speedup of slightly less than 1.2. The average translation time using the chaining solution is 240 ms, which is quickly amortized with TB reuse. It is worth noting that our tests did not experience TB cache flushes since the number of instructions of the kernels is small compared to the TB cache size.

Fig. 26 plots the number of TBs executed during simulation as a function of the programs (please note the log scale on the *y*-axis). As can be seen, there is a large variation in the number of executed TBs.

The average expansion ratio, computed as the mean of the number of micro-operations over the number of target instructions for all TBs of a programs, is very stable and almost constant: 8.25×. In the end, this is relatively close to the average 7.02 that we measure over the same benchmark for x86 to x86 scalar DBT.

The last experiment we present is a rough evaluation of the overhead due to the added micro-operations needed to warranty TB independence. Fig. 27 plots the expansion ratio as a function of the size (in number of target instructions) of the TB. The ratio, very high for small TBs, stabilizes around 6 for large TBs. This confirms the need of having big enough TBs to amortize the cost of our translation mechanisms.

[2] This allows to make all measures 120 times in less than a day.

To summarize, these results show that even if our solutions implies possibly many extra micro-operations, the simulation performances stay extremely good.

VI. Conclusion

This paper presents a solution to handle VLIW peculiarities when translating binary codes dynamically on scalar architectures. We propose an algorithm and detail its concrete implementation in a dynamic binary translator, which outlines that DBT adds corner cases that truly complicate the theoretical algorithm. These cases can be handled by generating code that takes some run-time decisions, thus, increasing the generated code size and lowering the efficiency of the DBT approach. Even though our prototype does not perform any optimization on the generated code, the experimental results show that this technique is still very interesting to achieve high simulation speed, as it runs from 1 to 2 orders of magnitude faster than a reference interpretive simulator.

References

[1] B. D. de Dinechin *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2013, pp. 1–6.

[2] R. A. Lethin, "How VLIW almost disappeared-and then proliferated," *IEEE Solid State Circuits Mag.*, vol. 1, no. 3, pp. 15–23, Aug. 2009.

[3] D. Wentzlaff *et al.*, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Sep./Oct. 2007.

[4] F. Pétrot *et al.*, "On MPSoC software execution at the transaction level," *IEEE Des. Test Comput.*, vol. 28, no. 3, pp. 32–43, May/Jun. 2011.

[5] M.-C. Chiang, T.-C. Yeh, and G.-F. Tseng, "A QEMU and systemc-based cycle-accurate ISS for performance estimation on SoC development," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 593–606, Apr. 2011.

[6] R. Leupers *et al.*, "Virtual manycore platforms: Moving towards 100+ processor cores," in *Proc. Design Autom. Test Europe Conf.*, Grenoble, France, Mar. 2011, pp. 715–720.

[7] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *Proc. 11th ACM SIGACT-SIGPLAN Symp. Principles Program. Lang.*, 1984, pp. 297–302.

[8] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *Proc. ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.*, Nashville, TN, USA, 1994, pp. 128–137.

[9] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[10] C. Cifuentes and V. Malhotra, "Binary translation: Static, dynamic, retargetable?" in *Proc. IEEE Int. Conf. Softw. Maint.*, Monterey, CA, USA, 1996, pp. 340–349.

[11] C. Cifuentes and M. V. Emmerik, "UQBT: Adaptable binary translation at low cost," *Computer*, vol. 33, no. 3, pp. 60–66, Mar. 2000.

[12] P. J. Drongowski, D. Hunter, M. Fayyazi, D. Kaeli, and J. Casmira, "Studying the performance of the FX!32 binary translation system," in *Proc. 1st Workshop Binary Transl.*, 1999, pp. 15–23.

[13] D. Ung and C. Cifuentes, "Machine-adaptable dynamic binary translation," in *Proc. DYNAMO*, 2000, pp. 41–51.

[14] J. C. Dehnert *et al.*, "The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proc. Int. Symp. Code Gener. Optim.*, San Francisco, CA, USA, 2003, pp. 15–24.

[15] K. Ebcioğlu and E. R. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility," in *Proc. ISCA*, Denver, CO, USA, 1997, pp. 26–37.

[16] C. Zheng and C. Thompson, "PA-RISC to IA-64: Transparent execution, no recompilation," *Computer*, vol. 33, no. 3, pp. 47–52, Mar. 2000.

[17] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proc. ACM Conf. Program. Lang. Design Implement.*, Vancouver, BC, Canada, 2000, pp. 1–12.

[18] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, 1996, pp. 68–79.

[19] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 41–46.

[20] Y.-C. Bao, A.-L. Liang, and H. B. Guan, "Design and implementation of crossbit: Dynamic binary translation infrastructure," *Comput. Eng.*, vol. 33, no. 23, pp. 100–102, 2007.

[21] J. Li, Q. Zhang, S. Xu, and B. Huang, "Optimizing dynamic binary translation for SIMD instructions," in *Proc. Code Gener. Optim.*, Austin, TX, USA, Mar. 2006, pp. 269–280.

[22] L. Michel, N. Fournel, and F. Pétrot, "Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation," in *Proc. Design Autom. Test Europe Conf.*, 2011, pp. 277–280.

[23] L. Baraz *et al.*, "IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on itanium-based systems," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchitect.*, 2003, pp. 191–204. [Online]. Available: http://portal.acm.org/citation.cfm?id=956417.956550

[24] M. Gligor, N. Fournel, and F. Pétrot, "Using binary translation in event driven simulation for fast and flexible MPSoC simulation," in *Proc. 7th IEEE/ACM/IFIP Int. Conf. Hardw. Softw. Codesign Syst. Synth.*, 2009, pp. 71–80.

[25] M. Montón, J. Carrabina, and M. Burton, "Mixed simulation kernels for high performance virtual platforms," in *Proc. Forum Specification Design Lang.*, 2009, pp. 1–6.

[26] J. A. Fisher, "Very long instruction word architectures and the ELI-512," *ACM SIGARCH Comput. Architect. News*, vol. 11, no. 3, pp. 140–150, 1983.

[27] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, vol. 29, no. 12, pp. 1184–1201, 1986.

[28] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proc. 16th ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, Jan. 1989, pp. 25–35.

[29] L.-N. Pouchet. (2011). *Polybench: The Polyhedral Benchmark Suite*. [Online]. Available: http://www-roc.inria.fr/~pouchet/software/polybench

Luc Michel received the Ph.D. degree in computer science from Université Grenoble Alpes, Grenoble, France, in 2014.

In 2016, he co-founded the Antfield SAS Company, Grenoble, that provides fast virtual platforming solutions and SystemC modeling for SoCs and IoT. His current research interests include fast simulation of multiprocessor SoC, especially using dynamic binary translation techniques.

Frédéric Pétrot (M'06) received the Ph.D. degree in computer science from Université Pierre et Marie Curie (Paris VI), Paris, France, in 1994.

From 1994 to 2004, he was an Assistant Professor of Computer Science with Université Pierre et Marie Curie (Paris VI). From 1989 to 1996, he was one of the main contributors of the open source Alliance VLSI CAD system. From 1996 to 2004, he led a team focusing on the specification, simulation, and implementation of multiprocessor SoCs. He joined the Grenoble Institute of Technology, Grenoble, France, as a Professor in 2004. Since 2006, he has been the Head of the System Level Synthesis Group with the TIMA Laboratory, Université Grenoble Alpes, Grenoble, where he is currently the Deputy Director. His current research interests include multiprocessor systems on chip architectures, including circuits and software aspects, and CAD tools for the design and evaluation of hardware/software systems.