Hybrid-DBT: Hardware/Software Dynamic Binary Translation Targeting VLIW

Simon Rokicki, Erven Rohou, Steven Derrien

Hybrid-DBT: Hardware/Software Dynamic Binary Translation Targeting VLIW

Simon Rokicki, Erven Rohou, Steven Derrien
Univ Rennes, Inria, CNRS, IRISA

***Abstract*—In order to provide dynamic adaptation of the performance/energy trade-off, systems today rely on heterogeneous multi-core architectures (different micro-architectures on a chip). These systems are limited to single-ISA approaches to enable transparent migration between the different cores. To offer more trade-off, we can integrate statically scheduled micro-architecture and use Dynamic Binary Translation (DBT) for task migration. However, in a system where performance and energy consumption are a prime concern, the translation overhead has to be kept as low as possible. In this paper we present Hybrid-DBT, an open-source, hardware accelerated DBT system targeting VLIW cores. Three different hardware accelerators have been designed to speed-up critical steps of the translation process. Experimental study shows that the accelerated steps are two orders of magnitude faster than their software equivalent. The impact on the total execution time of applications and the quality of generated binaries are also measured.**

I. Introduction

Maximizing energy efficiency while ensuring sufficient performance levels is a key issue for embedded system designers. This can only be achieved if the underlying hardware platforms exposes energy/performance trade-offs. This is done with Dynamic Voltage and Frequency Scaling (DVFS), which trades performance (clock speed) for energy efficiency (supply voltage). DVFS enables fine-grain control of energy/performance trade-off at the processor core level, with only modest performance overhead, and is available in many embedded computing platforms. However, with the end of Dennard scaling, DVFS is becoming inefficient: energy savings are obtained at the price of an unacceptable loss of performance.

Because of this, computer designers now also rely on architectural heterogeneity to expose energy/performance trade-off, and this heterogeneity can take many different forms. For example, it is possible to combine different type of architectures (CPU, GPU, DSP...) on the same device. Since each type of architecture has its own performance/energy ratio, the designer can determine the mapping of tasks that best fits its requirements. The problem with this approach is the lack of flexibility, as this mapping must be chosen at design time, and cannot be changed at run-time, due to the difference in programming models and/or instructions sets between cores.

A solution to this problem is to use single-ISA heterogeneous architectures, as proposed by the Arm big.LITTLE architecture [1], which uses two distinct micro-architectures to run Arm binaries. One is an aggressive Out-of-Order (OoO) processor capable of delivering high-performance at the expense of energy-efficiency, the other is a simple in-order core favoring energy efficiency over performance. Since both micro-architectures share the same ISA, dynamic task migration from one core to the other becomes possible: when an application requires higher performance levels, it is run on the OoO architecture. As soon as performance requirements become low enough, the task is migrated to the in-order core.

The strength of OoO architectures is that they guarantee good performance levels regardless of the target application. The downside is that, for many applications, the use of a simpler, statically scheduled, hardware architecture (such as a VLIW core) could have achieved similar performance levels with a much lower energy budget. The problem is that VLIW exposes instruction level parallelism directly in their ISA, which prevents from providing direct binary compatibility with the other cores in the platform. This issue can be circumvented through the use of Dynamic Binary Translation (DBT).

DBT consists in executing binaries targeting a *guest* ISA onto a *host* machine with a different instruction set. This is made possible by performing the translation on-the-fly, each instruction being translated right before its execution. DBT was originally used for fast simulation of an architecture and/or for providing inter-system compatibility, but is now also used in the context of heterogeneous architectures. In this case, DBT is used to support architectures with different ISA while giving the illusion of a single-ISA system.

This idea was initially demonstrated by Transmeta Code Morphing Software [2] and was more recently revived with NVidia's Denver [3] architecture. In these approaches, DBT is used to execute x86 or Arm binaries on a VLIW-like architecture to reduce the overall energy consumption (compared to an OoO core) without impacting performance. In this case, the performance/energy overhead caused by the DBT stage must be as small as possible to make the approach viable.

In this paper we introduce Hybrid-DBT, a hardware-software DBT framework capable of translating RISC-V into VLIW binaries. Since the DBT overhead has to be as small as possible, our implementation takes advantage of hardware acceleration for performance critical stages (binary translation, dependency analysis and instruction scheduling) of the flow.

Thanks to hardware acceleration, our implementation is two orders of magnitude faster than a pure software implementation and enable an overall performance improvements by 23 % on average, compared to a native RISC-V execution.

In addition, and to the difference of others industrial DBT tools targeting VLIW (e.g. Transmeta CMS and NVidia's Denver), Hybrid-DBT is open source and is meant to serve as an open platform for academic research on the topic.

The remainder of this paper is organized as follows. The global organization of Hybrid-DBT framework and its run-time management are presented in Section II and III. Section IV presents the different hardware accelerators used in the framework and Section V presents the experimental study.

II. Hybrid-DBT Framework

In this section, we provide an overview of the system and describe its hardware organization as well as the intermediate program representation used by our DBT framework.

A. Overview of the system

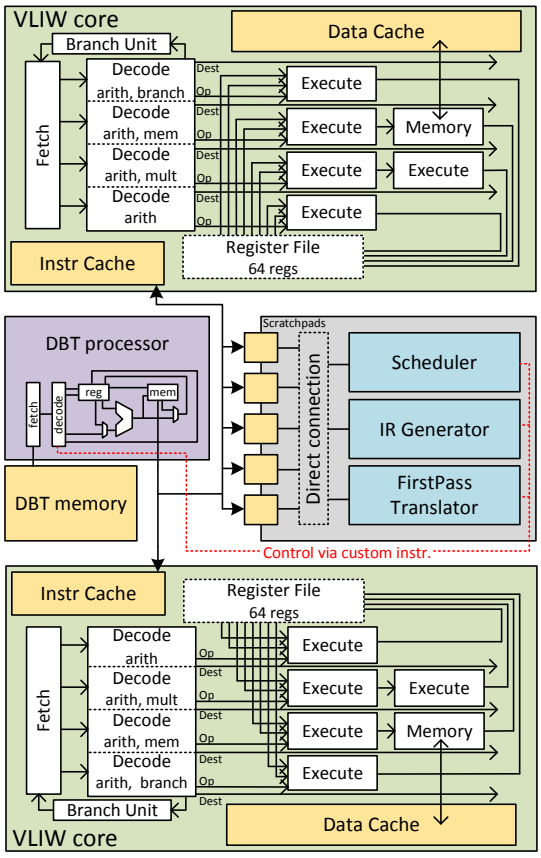

Fig. 1. Overview of the Hybrid-DBT system coupled with two execution VLIWs

Hybrid-DBT system is a hardware software co-designed DBT framework. It translates RISC-V binaries into VLIW binaries and uses dedicated hardware accelerators to reduce the cost of certain steps of the translation/optimization process. Figure 1 gives an overview of the DBT system with the following components:

- The **VLIW cores** are custom VLIW implementations loosely based on the ST200 processor. This is a 4-issue processor with 64×64 bits registers. Pipelines are organized in two different ways: general purpose ways have 3 steps while specific ways, which perform memory accesses and/or multiplications, have 4 pipeline steps. Contrary to the ST200, the VLIW has no forwarding or stall logic. This core has been modified to add custom instructions to profile the execution. These cores have been designed specifically for this work using High-Level Synthesis. Consequently, they can be modified and/or extended easily if we want new features.
- The **DBT processor** is a low-footprint in-order core that manages the DBT process and performs optimizations while the VLIW core is busy executing the application. The DBT processor has access to the VLIW memories: the translated binaries are written in the VLIW instruction cache and the profiling information are read from the VLIW data memory. In order to reduce the cost of some optimizations, the DBT processor has access to three different hardware accelerators.
- The **Hardware Accelerators** are a set of three hardware components that have been developed for specific tasks of the DBT process: *IR Scheduler*, *IR Generator* and *Firstpass Translator*. These accelerators have access to small scratchpads which are used to provide memory bandwidth. These scratchpads can also be accessed by the DBT processor which is in charge of initializing them. Accelerators are controlled through a custom instruction of the DBT processor. These different accelerators are described in Section IV.

In this work, we use a dedicated core to manage the DBT process and perform optimization. This helps reducing the translation overhead on the execution time, as the VLIW core is only used to execute code. In the big.LITTLE platform described in Figure 1, one of the LITTLE core could be used as a DBT processor to save some area.

B. Hardware/Software partitioning

As we said previously, Hybrid-DBT uses three different hardware accelerators to reduce the cost of the translation and optimization process. During the development, we had to choose the partitioning between hardware and software parts of the process. Several ideas were used to drive our decisions:

- If hardware accelerators can deliver higher performance and higher energy efficiency than software, it is also less flexible. If a transformation has many context-specific conditions, it is not suitable for hardware. Transformations that have been mapped into hardware are simple and regular algorithms and strong assumptions have been made to keep them as regular as possible.
- Dynamic compilation is based on the idea that every cycle/joule spent optimizing binaries has to be recouped by the benefits of this optimization. Consequently, while executing cold-code, the framework has no idea of how often the generated binaries will be used and tries to spend as little time as possible optimizing it.

In the system, the accelerator called *First-Pass Translator* is used in the first optimization level to perform a first translation that needs to be as cheap as possible. The *IR Generator* is then used at the next optimization level to generate a higher level representation of the translated binaries, which is used by the *IR Scheduler* to generate VLIW binaries by performing instruction scheduling and register allocation.

Other optimizations done in the DBT framework are more complex and more data dependent. Moreover, these transformations are done only on hotspots, which ensure that the time spent optimizing is paid back. For these two reasons, other optimizations are done in software by the DBT processor.

A complete description of Hybrid-DBT transformation and optimization flow can be found on Section III and more details on the hardware accelerators are given on Section IV.

C. The Intermediate Representation

Hybrid-DBT optimization flow is based on the use of an Intermediate Representation (IR). Designing this IR was important and we answered two challenges while doing so: i) the IR is generated and used by hardware accelerators, consequently the structure has to be as regular as possible and may not use object/pointers structures; ii) the IR has to be designed in order to reduce the cost of some software optimizations (unrolling, construction of traces for trace-based scheduling, register allocation, etc.).

As the IR is mainly designed for performing instruction scheduling, the data-flow graph is directly encoded inside: each instruction of a block knows which instruction has produced its operands and also which instruction has to be executed before it. These last dependencies are used to ensure that the control-flow is correct while scheduling a trace or that memory accesses are done in the same order than in original sources. Figure 2 provides an overview of the IR bit-encoding:

- The field `nbRead` contains the number of times the result of an instruction is used as an operand in the current block. This value will be used by register allocation.
- Fields `nbPred` and `nbDPred` correspond to the number of predecessors and to the number of data predecessors on the block data-flow graph.
- Fields `predNames` are a list of predecessor IDs (their index in the current block) which has to be scheduled before the current instruction.
- Field `dest` is an ID of the destination register and bit `alloc` controls if the `dest` has to be renamed or not.
- Field `instr` encode all the necessary information about the instruction (type, opcode and immediate values).

In the following, we discuss the different assumptions taken on the IR structure. We present the different types of dependencies used inside a block, the way registers are allocated and finally we give an idea of how control-flow is represented in the IR.

a) Data and control dependencies: Inside a block of the IR (e.g. basic block or trace), each instruction has access to its predecessors in the data-flow graph. There are two kinds of dependencies that can be encoded in the IR: data-dependencies and control-dependencies. Data-dependencies mean that the current instruction needs the result of another instruction and consequently, satisfying a data dependency depends on the latency of the predecessor.

Control-dependencies are used to ensure that some instructions are scheduled in a given order. For example, in the firsts optimization levels of Hybrid-DBT, we add control-dependencies to ensure that memory accesses are scheduled in the same order as in the original program. Satisfying a control-dependency does not depend on the instruction latency.

At the bottom of Figure 2, data-dependencies are represented using plain arrows while control dependencies are represented using dashed arrows. A control dependency ensures that store number 0 is executed before load number 1, which is also constrained to be scheduled before store number 4.

As we will see in next paragraph, register allocation is partially solved at schedule time. Consequently, name dependencies (e.g. Write-after-Read and Write-after-Write) are not encoded in the IR. The scheduler will have to ensure that they are satisfied, according to the effective allocation.

b) Local and global values: In the IR, we made several assumptions on the register allocation, which eases the scheduling and several software optimizations while maintaining performance. When an instruction reads a value, it can read a **local value** (i.e. a value created in the same block) or a **global value** (i.e. a value created in another block). Similarly, when an instruction writes a value, it can be written in a **temporary register** or in a **global register**. The temporary register is usable only in the current block while the global register may be used as a global value in another block. Temporary register are allocated to physical register at schedule time.

By default, each instruction has a global register allocated. These registers represent a valid register allocation and can be used by the scheduler if it fails at building a different allocation. In practice, this register allocation is first derived from the original register allocation in the RISC-V binaries.

However, each instruction has a `alloc` field that means the scheduler can store the value in a temporary register. At schedule time, it allocates, if possible, a physical register where to store the value. When an instruction has a data dependence, it knows the ID of the instruction that created the local value and can find out the physical register used to store it. Of course, a value used in another block (e.g. used as a global value) cannot be stored in a temporary register.

The advantage of allocating a temporary register is to remove name dependencies while scheduling instructions. For example, in Figure 2, two interpretations of the data-flow graph are given, depending on whether instructions 2 and 3 have `alloc` bit or not. On the left-most graph, instructions 2 and 3 do not have the `alloc` bit set to one and consequently writes in registers 8 and 9. As those registers are also written by instructions 5 and 6, there are register dependencies between those instructions (instruction 2 has to write its result before instruction 6 does and instruction 3 has to read it before instruction 6 erases it). The right most graph corresponds to the same block where a new register has been allocated to store the local values created by instructions 2 and 3. The register based dependencies is removed and the critical path of the graph is reduced.

Using this representation, we can have a default register allocation for the first stages of the translation process and update it through the `alloc` bit of each instruction that writes a value which is only accessed in current block. In case a more complex register allocation is needed, it is possible to modify the global register assigned to each instruction and stop using the `alloc` bit.

c) Control-flow graph: The intermediate representation also uses an object representation for representing function and block as well as control flow graph of a given function. As this information is only used for software transformation, we now use a pointer representation: each block of a function has pointers to its successors in the control-flow graph.

This graph is then used to perform inter-block optimization such as loop unrolling and trace building.

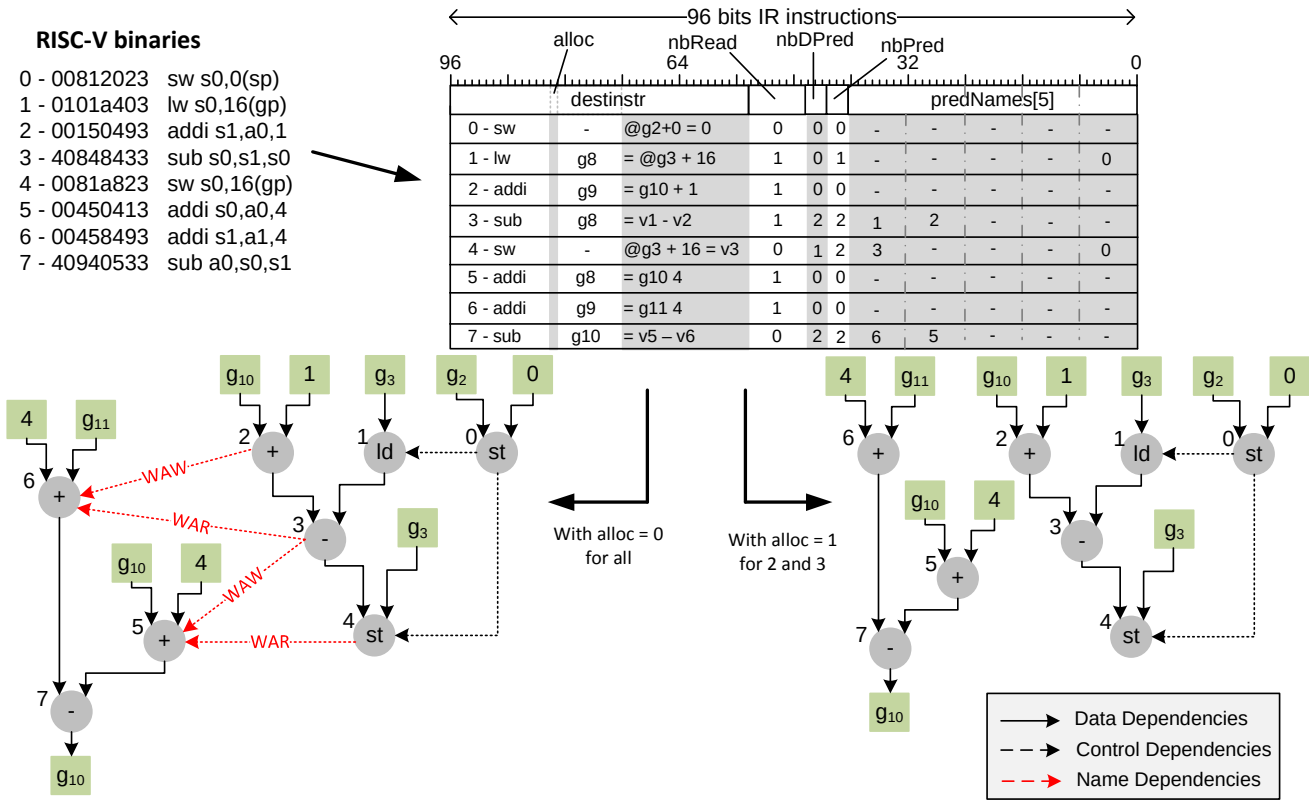

Fig. 2. Illustration of the structure of the Intermediate Representation and its interpretation in terms of data-flow graph. The two graphs pictured also show the impact of the bit 'alloc' on the dependencies. Left-most graph has a critical path of 5 while right-most graph has 3.

III. Translation and Optimization Flow

In this section, we present the translation and optimization flow which is used in Hybrid-DBT. We will first give an overview of the different steps involved and we will describe them in depth.

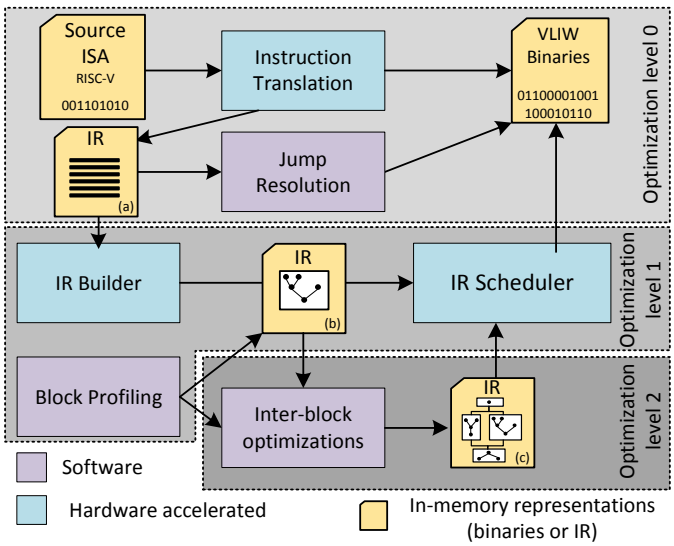

Fig. 3. Hybrid-DBT translation and optimization flow

Hybrid-DBT is organized as a three-step translation and optimization flow. A schematic view of the flow is provided on Figure 3. The three steps are the following:

- **Translation of instructions** is the optimization level 0. When executing cold-code, the DBT framework tries to spend as little time/energy as possible to translate binaries. In the framework, we first perform a naive translation of each source instruction into one or several instructions from the VLIW ISA. This optimization step does not try to exploit ILP at all.
- **Block building and scheduling** is the optimization level 1. When a block has more than a certain number of instructions, it is considered as a candidate for optimization level 1: an intermediate representation of the block is built and used for performing instruction scheduling. This optimization level is triggered aggressively, without any profiling information because of the use of hardware accelerators that reduces its cost. Scheduled blocks are then profiled to trigger further optimizations.
- **Inter-block optimization** are performed at optimization level 2. When a given block has been executed enough, it triggers the inter-block optimization. This optimization level analyzes the control flow (place and destination of each jump instruction) to build a control-flow graph of the function (i.e. all blocks that are accessed through direct branches except calls). Then inter-block optimizations are performed on this function to extract more ILP.

A. Instruction Translation

When a portion of code is first encountered, the execution have to start as soon as possible and the framework cannot know if the number of executions will be high enough to recoup any optimization performed.

For these reasons, it performs a naive translation of each source instruction into VLIW ISA without trying to exploit ILP at all. To reduce even more the cost of this first translation, a hardware accelerator (named `First-pass Translator`) has been designed to translate a fragment with

a very high throughput. See Section IV-A for more details on this accelerator.

During this translation, the accelerator extracts two kinds of information:

- the location and the destination of each jump
- the divergence between source binaries and translated binaries (e.g. places where an instruction is translated into two VLIW instruction or where `nop` cycles are inserted to handle instruction latency). This information can be used to translate an address in source binaries into the equivalent address in VLIW binaries.

Using this information the DBT framework is able to build block boundaries, which are stored for later use. In Figure 3 this information is represented as `IR (a)`. The translation framework also handles direct and indirect jumps. During the translation, all jumps go through system code that computes the correct destination using the initial destination (e.g. destination in source binaries) and the divergence information. During the execution, the translation framework solves all direct jumps by computing the effective destination and modifying the jump instruction.

Indirect branches are branches that can only be solved at run-time. For those, the execution always goes through the calculation of the effective destination.

Hiser et al. [4] evaluated how to efficiently handle those indirect branches. In our work, we use a small hashtable containing previous initial and effective destinations.

This first translation is the only one exposed to the source ISA. Consequently, if we want to handle a new ISA in the framework, we just have to design a new first translation. Current flow already handles MIPS and RISC-V ISAs.

B. Block building and scheduling

During the first translation, the DBT framework build a list of basic blocks from the jump locations and destinations. At this step of the translation process, these blocks only contains start/end addresses and the translated code does not exploit instruction parallelism. During this optimization level, the DBT framework optimizes basic blocks by building a higher level representation of the binaries and uses it to perform instruction scheduling. Generated binaries exploit the VLIW capabilities to execute several instructions per cycle.

To generate the Intermediate Representation presented in Section II-C, the framework uses a hardware accelerator called `IR Builder`. This accelerator reads the translated VLIW binaries and generates the data-flow information accordingly. Details on this accelerator can be found on subsection IV-B.

Once the IR has been built for a block, the DBT framework uses another hardware component called `IR Scheduler`. This accelerator is in charge of performing instruction scheduling and register allocation on the intermediate representation of the block. This scheduling follows the assumptions taken on the IR design: it ensures that data-flow and control flow dependencies represented in the IR are satisfied and that name-dependencies (which are not encoded) are correctly handled. This accelerator also support the use of the `alloc` bit which enable register renaming. More details on the accelerator can be found in subsection IV-C.

Finally, the generated binaries are inserted in place of the first translation and other jump instructions do not need to be modified. If the optimized block contains a backward jump (i.e. if it is in a loop), the DBT process profiles it to trigger further optimization: profiling instructions are inserted in the IR before the scheduling step.

Note that this optimization level does not need any profiling information to be triggered: it is applied on any block that is bigger than a certain threshold. This aggressiveness is made possible by the use of hardware accelerators which drastically reduce the cost of this optimization level. These results will be seen in the experimental study on Section V.

C. Function building and inter-block optimizations

When the profiling shows that a block has been executed often enough, the DBT framework will trigger the second optimization level that consists in inter-block transformations. First, the framework will use the aforementioned information on jump location and destination to build a control-flow graph of the function containing the block. It also profile different blocks to have information on loops and branches taken.

Once the IR of the function has been built, the software may apply different kind of software optimizations, which increase the available ILP after scheduling.

These software optimization can affect the control-flow graph of the function. This is the case for **loop unrolling**, **trace construction** and **function inlining**. These transformations rely on a process that merges two IR blocks to form one larger block. This process requires to update the data-flow graph inside the block. This is made simple by the IR design: merging two blocks only require to bind the first access to global values of the second block with the last write to a global register in first block. Dependencies have to be inserted to ensure that the last and first memory instruction of first and second blocks are scheduled in a correct order.

Other software transformations modify the register allocation in the function to increase the ILP available or perform a memory disambiguation by removing some of memory dependencies.

IV. Hardware Accelerators

In this section, we describe the structure of the different accelerators we developed. All these components have been designed using High-Level Synthesis: we developed a behavioral description of the algorithm in C and used Mentor Catapult to generate the VHDL. To increase performance of the hardware component, we made the following optimizations:

- rewriting algorithms with regular data-structures;
- memory partitioning to provide more access ports;
- loop pipelining to increase throughput;
- explicit forwarding of memory values to remove inter-iteration dependencies.

The performance of HLS generated VHDL is often considered as less efficient than what a designer could do in RTL. However, we do not have the resources to to develop the accelerator manually. The rest of this section will describe the accelerators used in Hybrid-DBT framework.

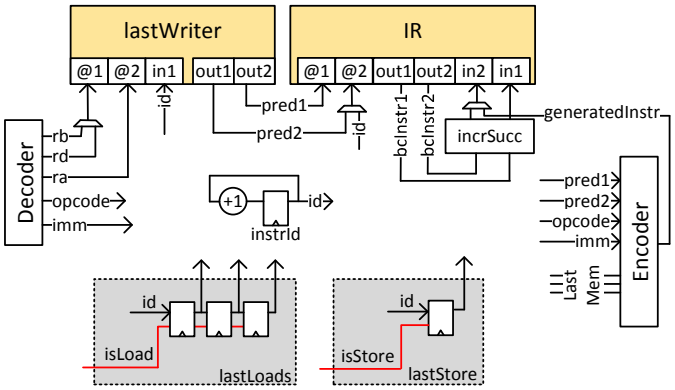

Fig. 4. Simplified representation of the IRBuilder datapath.

A. The First-pass translator

The goal of the first-pass translator is to translate blocks of instruction at the lowest price possible. This translation does not have to generate optimized code.

We developed an accelerator that reads RISC-V instruction, extract the different fields (opcode, funct3, funct7, rs1, rs2 and rd) and use them to create one or several VLIW instructions. Standard instruction like additions are straightforward to translate because they have there equivalent in VLIW ISA. Some RISC-V instructions may require several VLIW instructions to preserve the original semantic. It is the case for conditional branch instructions: RISC-V offers many different branch operations (bne, bge, blt) while our VLIW ISA only offers two (beqz and bnez). These RISC-V instructions will have to be translated using two different instructions.

Concerning register allocation, our VLIW has 64 general purpose register allowing us to keep a one to one mapping of the 32 registers used in RISC-V ISA.

The hardware accelerator we developed can be seen has a simple pipelined finite state machine: a new VLIW instruction is generated at each cycle. Because RISC-V ISA and our VLIW ISA are similar, this accelerator is small. However, the same idea could be applied for more complex translations, keeping similar throughput. Only the size of the translator may increase due to the more complex input ISA.

Note that the accelerator keeps track of the insertions (when one input instruction is translated into two or more VLIW instructions) as well as jump locations and destinations. This information can be computed easily during the translation process and are stored in dedicated scratchpads. It will be used by the DBT framework to solve jump destinations, as described in III-A.

B. The IR builder

The goal of the IR builder is to analyze VLIW binaries in order to build the Intermediate Representation described in Section II-C. The accelerator goes through all instructions in sequential order and build the data-flow graph while recording dependencies. Dependencies will be added between memory stores and memory loads to ensure the memory coherency.

Figure 4 is a simplified representation of the hardware accelerator. We can see the following components:

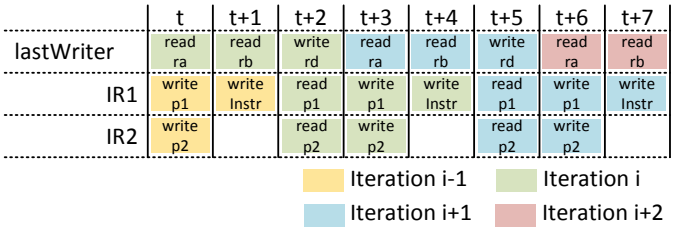

Fig. 5. Representation of the pipeline organization of irBuilder.

- A memory called `lastWriter` where we store the ID of the last instruction that wrote a given register. At the beginning of the block, the value is initialized with -1.
- A memory called `IR` will contains the IR being generated.
- A register called `last store` hold the ID of the last store instruction in the current block while `lastRead` is a shifted register storing the last three memory loads in current block.

Adding an instruction in the IR consists in five steps, which are presented below. Figure 5 represents a possible schedule of all memory accesses needed to perform these steps. These memory accesses are the performance bottleneck of the accelerator.

- The instruction uses at most two registers for its operands. For each of them, the value of the last instruction that modified it in the current block are read from the `lastWrite` memory. If the value is -1 then the IR will be built with a global value; otherwise it use the ID of the instruction that created the value.
- The instruction may create a value that is stored in a register. The accelerator stores the ID of the current instruction as the last writer of this physical register. Any other instruction that accesses this register before any modification depends on this instruction.
- The accelerator then handles data dependencies. Current instruction have a reference to the two data predecessors; these two predecessors will have one more read. To handle this last modification, the accelerator reads `IR` memory to get the IR instruction corresponding to each predecessor. It then increments the field `nbRead` and write back the IR instruction.
- Finally, if the instruction is a memory instruction (e.g. load or store), the accelerator adds control dependencies to handle memory coherency.
  - If the instruction is a load, a dependency is added from the last store and the instruction is added to the list `lastLoads`. If this list is full, a dependency is added from the older load instruction to the new one. This old instruction is then replaced by the new one.
  - If the instruction is a store, all last three loads instructions are marked as control predecessor of the instruction. The instruction is then stored as the last store instruction.
- Finally, the IR instruction is generated using all the information gathered and it is stored in the `IR` memory.

To ensure the memory coherency, control dependencies are inserted between load and store instructions. For example, a store instruction has to be scheduled after all previous load

instructions in RISC-V binaries. However, since the maximum number of predecessor that can be encoded in the IR is 5, only the three last load instructions are treated. If there are more than 3 loads, dependencies are inserted between those loads to maintain the coherency. When a store occurs, all three last loads are marked as predecessors.

As shown on Figure 5, the accelerator is pipelined with an initiation interval of three. It means that a new instruction is generated every three cycles. Figure 5 represents three of these iterations overlapping.

C. The Instruction Scheduler

The instruction scheduler is the most complex hardware accelerator of the Hybrid-DBT system. This accelerator is in charge of scheduling each IR instruction in an execution unit of the VLIW at a given cycle. It ensures that IR dependencies and name dependencies are satisfied. Moreover, the scheduler is in charge of performing basic register renaming when needed. The rest of this Section is divided in two parts: first we give a behavioral definition of the algorithm used and then we give the intuition of the accelerator organization.

1) Scoreboard Scheduling algorithm: The algorithm used for instruction scheduling is the scoreboard scheduling. It is a greedy heuristic that tries to place each instruction in a window, ensuring that all its predecessors are already scheduled. This algorithm has been modified to detect name dependencies and support register renaming when needed.

Algorithm 1 describes the method used for scheduling. There are six arrays involved in the algorithm:

- The array called `IR` is the array containing the intermediate representation of the block being scheduled.
- The array `window` is the moving window of the scoreboard scheduler. Each time we want to schedule an instruction, each place of the window will be inspected to see if a place is available. If the instruction cannot be placed in the window, the window is shifted so that the instruction fits at the last place of the window.
- The array `placeOfInstr` contains the place of each instruction of the block. It is updated every time a new instruction is placed.
- Arrays `lastRead` and `lastWrite` contain, for each physical register, the place of the last instruction that read/write it.
- The array `placeReg` is a mapping between instructions of the block and the physical register used to store the value produced.
- The array `nbRead` keeps track of how many times a physical register has to be read before being freed. This is done using the corresponding value from the IR.
- The array `binaries` is the place where VLIW binaries are stored.

Algorithm 1 is divided in three parts. The first part corresponds to the first `for` loop that iterates over all predecessors of the current instruction and calculates the earliest place where all dependencies are satisfied. The cycle where a predecessor has been scheduled is found in the array `placeOfInstr` and an offset is added according to the type of the predecessor (e.g. control or data predecessor).

The second loop of the algorithm handles the register renaming mechanism: if the IR instruction is marked as `alloc`, a new physical register will be picked from a FIFO and the system will adapt the earliest place by considering the last read and the last write on this register. The value of `nbReads` is then initialized with the value from the IR. If there is no free register, the scheduler will ignore the renaming.

Finally, the third loop will go through the scoreboard window to find the earliest slot where an execution unit is available and where all dependencies are satisfied.

After these three loops, different arrays are updated: `lastRead`, `lastWrite`, `placeReg` and `nbReads` are updated according to used registers; the place in the window is marked as used and the place of the instruction is stored in `placeOfInstr`. Finally, the instruction is assembled and is stored in generated binaries.

```
for oneInstruction in IR do
    earliestPlace = windowStart;
    for onePredecessor in oneInstruction.predecessors do
        if onePredecessor.isData then
            earliestPlace = min(earliestPlace,
              placeOfInstr[onePredecessor] + latency);
        else
            earliestPlace = min(earliestPlace,
              placeOfInstr[onePredecessor] + 1);
        end
    end
    rdest = oneInstruction.getDest();
    if oneInstruction.isAlloc then
        rdest = getFreeRegister();
    end
    earliestPlace = min(earliestPlace, lastRead[rdest],
      lastWrite[rdest]);
    for oneCycle in window do
        for oneIssue in oneCycle.issues do
            if oneIssue.isFree and oneCycle ≥ earliestPlace
              then
                place = oneCycle.address
            end
        end
    end
    if notFound then
        window.move(max(1, earliestPlace - window.end))
        place = window.end
    end
    window[place] = instrId;
    placeOfInstr[instrId] = place;
    ra = placeReg[pred1];
    rb = placeReg[pred2];
    placeReg[instrId] = rdest;
    lastRead[ra] = place;
    lastRead[rb] = place;
    lastWrite[rdest] = place;
    binaries[earliestPlace] = assembleInstr(ra, rb, rdest);
end
```

Algorithm 1: Scheduling algorithm used for the accelerator

Figure 6 illustrates how the algorithm schedules the instructions on a short example. The targeted architecture is a 3-issue VLIW where br and $arith$ instruction has one cycle of latency and mem has 2 cycles. The left-most part of the figure represents the block being scheduled and its corresponding data-flow graph. The initial state of the scheduling window is also depicted, with instructions a to d already scheduled. To

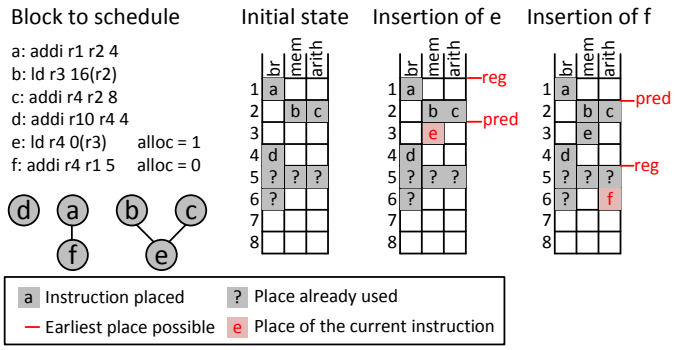

Fig. 6. Short example of the behaviour of the scheduler on a 3-issue architecture where br and $arith$ ways needs 1 cycle of latency and mem needs 2 cycles. left-most part represents the block to schedule, the rest represents the different states of the reservation table while scheduling instructions e and f.

make our example more complete, we consider a scheduling window in which earlier instructions have been scheduled (represented with a question mark).

The first step is to schedule instruction e: to find the earliest timestamp for which dependencies are satisfied, each predecessor's timestamp are considered and an offset is added according to the instruction latency. In our example, predecessors are scheduled at cycle 2, and have, respectively, a one and a two cycle latency. As a consequence, the earliest timestamp for e is 3. The next step is to consider register-based dependencies. One can observe that instruction e is to write in register $r4$ which is also read by instruction d. However, instruction e also has `alloc` set. A new register is therefore allocated and the write after read constraint is removed. Instruction e is placed at timestamp 3.

The algorithm then proceeds to instruction f, which has only one predecessor scheduled at timestamp 1. Consequently, instruction f could be scheduled at timestamp 2. However, instruction f has a write after read dependency with instruction d, scheduled at timestamp 4. As `alloc` is not set, dependency cannot be removed and the earliest timestamp (register-wise) for f is 5. The instruction is scheduled at cycle 6 (which is the first slot available in the scheduling window).

2) Organization of the accelerator: The scoreboard scheduling algorithm described previously has been implemented in hardware. Figure 7 gives a basic idea of the internal organization of this component. We can see that most of the arrays used are mapped into memories while the window is mapped into registers. This allows to have a parallel access to all cycles of the window. This enables to unroll the third loop of Algorithm 1 and to execute it with the lowest overhead possible.

Other parts of the algorithm are mainly instruction decoding/encoding and memory updating. All this can be done in parallel at low expense. However, the first loop still requires to access the memory `placeOfInstr` for each potential predecessor. This loop is likely to become the main bottleneck of the accelerator.

Similarly to other accelerators in the system, the scheduler has been pipelined to hide the latency of scheduling an instruction. Figure 8 gives a schedule of the different memory accesses needed to schedule an instruction. The initiation interval used in this schedule is 4. As the window has been implemented using registers instead of memory blocks, there is no constraint on port utilization. Consequently, the window is not represented on Figure 8.

We can see that the main bottlenecks of the accelerator are the accesses on `placeOfInstr` memory as well as the inter-iteration dependency inside. Indeed, Figure 8 also represents a violated dependency: if this instruction scheduled at iteration $i+1$ depends on instruction scheduled at iteration i then the read on `placeOfInstr` at cycle $t+5$ may collide with the write schedule at the same cycle. This problem has been solved using explicit forwarding: the two addresses are compared and the result of the load is used only if there is no collision. Otherwise it will use the value which was to be written.

The area used by the accelerator mainly depends on the size of the window. We used a window of size 16 which appears to give the best performance while keeping a reasonable area.

V. Experimental results

This section presents the experimental study we made as well as the result obtained. In the different subsections, we demonstrate that our hardware accelerators brings real improvements in DBT translation process.

All accelerators mentioned in Section IV and the VLIW core have been generated using Mentor Graphics Catapult HLS tool. Entry files are C++ description of the component behavior as well as constraints on the loops, dependencies and clock frequency. The sources of Hybrid-DBT toolchain are available on Github[1].

Once the RTL description has been generated by the HLS tool, we used Synopsis Design Compiler combined with a 28 nm library from STMicroelectronics to generate a gate-level representation of the component. This representation is used to get the frequency and the area of the design. To obtain an average power consumption, we used Modelsim to perform a gate-level simulation with real input data, derived the average switching activity of each gate and Design Compiler used it to give an estimation of the power consumption.

In some experiments we compare our approach with existing micro-architectures for RISC-V: the Rocket-Core. For these approaches, we used the Chisel HDL generator to get a RTL level description of the core that we gave to Synopsis Design Compiler with the 28 nm gate library.

To measure the performance of the translation tool, we used several benchmarks from the Mediabench suite. All were compiled for the RV64IM version of RISC-V ISA using GCC 7.1 with O2 flag. These binaries are given as inputs to the DBT flow or are used, for some experiments, as input to Rocket simulator. Table I provides a short description of the benchmark code size. Numbers provided are the number of instruction in the application as well as the number of blocks and functions optimized by our flow. Please note that the number of functions optimized may depend on the data-set as our flow relies on profiling to decide whether Level 2 DBT is triggered or not.

[1] https://github.com/srokicki/HybridDBT

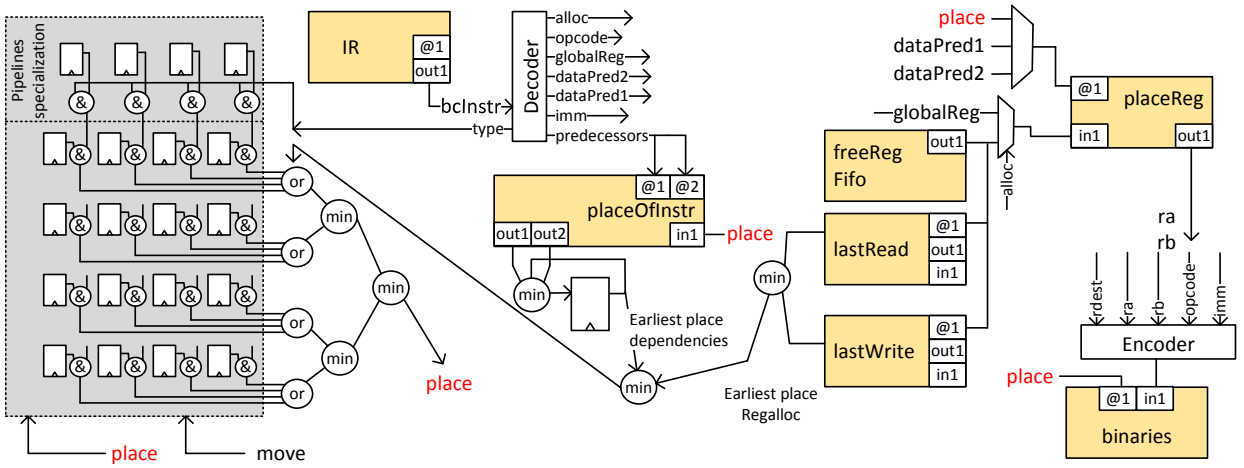

Fig. 7. Simplified representation of the Instruction Scheduler datapath.

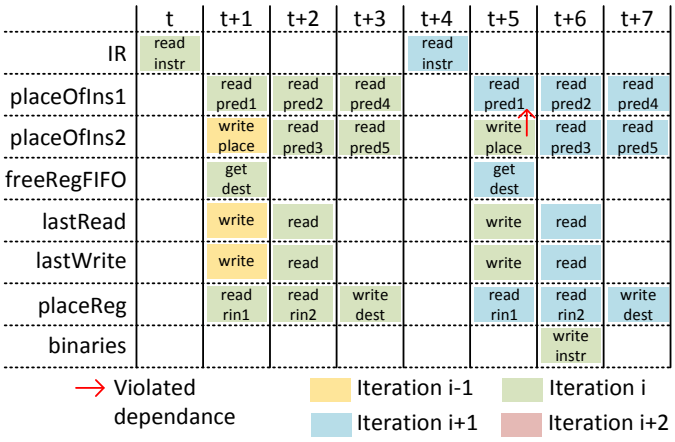

Fig. 8. Schedule of the different memory operations of IRScheduler with initiation interval at 4.

	#Instr.	#Blocks	#Funct.
adpcm	15 129	934	6
epic	29 153	1 797	33
g721	17 028	1 057	13
gsm	22 842	1 414	17
jpeg	42 408	2 770	18
mpeg	28 063	1 725	25

TABLE I
CHARACTERISTICS OF THE DIFFERENT BENCHMARKS USED IN THE EXPERIMENTAL STUDY. VALUES DISPLAYED ARE THE NUMBER OF INSTRUCTION TRANSLATED AND THE NUMBER OF BLOCK AND FUNCTIONS OPTIMIZED BY THE FLOW.

A. *Cost of each optimization level*

The first experiment measures the efficiency of the different hardware accelerators. We executed the different optimization passes and profiled the execution time for all our benchmarks. We used two experiments: one running in full software (with an equivalent implementation of the first translation, the IR building, and the scheduling), executed on a cycle-accurate RISC-V simulator (with a micro-architecture similar to the one of the Rocket core); the other one models the time taken by accelerators. The results shown here compare the execution time and the energy cost to perform these optimizations.

a) Performance improvement due to accelerators: Figure 9 shows the improvement from using hardware accelerators

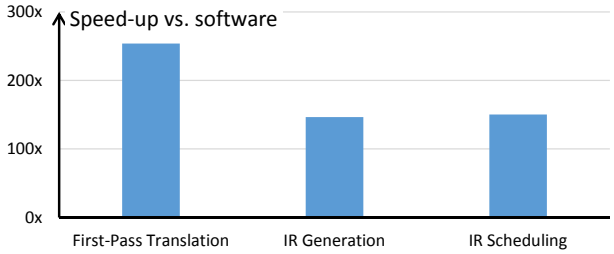

Fig. 9. Speed-up against software DBT

compared with software implementation. We can see that `First-Pass Translation` is around 250× faster while `IR Generation` and `IR Scheduling` are 150× faster than the software version running on a Rocket core.

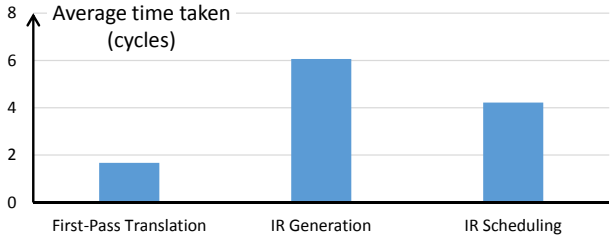

Fig. 10. Average time taken for performing transformations with Hybrid-DBT

Figure 10 represents the number of cycles required to transform one instruction. For `First-Pass Translation` it corresponds to the time taken by the hardware accelerator divided by the number of RISC-V instruction effectively translated. For the two others it has been divided by the number of IR instructions.

We can see that `First-Pass Translation` needs on average 1.7 cycles to translate one RISC-V instruction. `IR Generation` needs 6 cycles to generate one IR instruction while `IR Scheduling` needs 4.2 cycles to schedule it. Figure 10 also reports the min and max values obtained for each benchmark. We can see that the variation is low.

b) Energy consumed: Figure 11 shows the gain in energy consumption coming from the use of hardware accelerators, compared with a software approach. For these experiments, we measured the average power consumption of the Rocket core and of the different accelerators and coupled that with the time needed to perform the transformation. We consider

that accelerators are power gated when not in use. However the DBT processor is only stalled while waiting for the result of an accelerator. Consequently its energy consumption is included in this experiment. Results shows that `First-Pass Translation` consumes on average 200× less energy than its software counterpart. `IR Generation` consumes 80× less energy while `IR Scheduling` consumes 120× less energy than the software version running on a Rocket core.

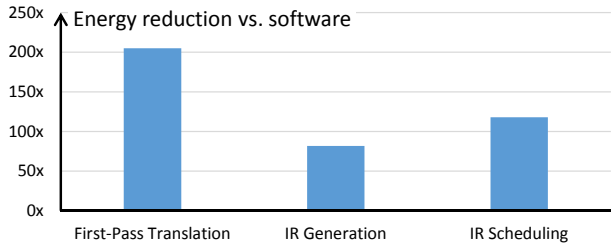

Fig. 11. Improvements in energy consumption vs software DBT

As a dedicated core is used for the DBT process, one would argue that the time spent optimizing code is not critical. However, the energy spent doing the transformation is an important factor, even more in a context of heterogeneous systems that are meant to offer interesting performance/power consumption trade-offs.

B. Translation and execution costs

As we have shown in the previous experiment, the use of hardware accelerators drastically reduces the cost of the translation process compared with a software implementation. In this experiment, we show its impact on the overall execution cost. All benchmarks have been executed with the translation framework. One first run is made using the different hardware accelerators and another with the software implementation.

	Hybrid-DBT		Software DBT	
	Execution time (ms)	Energy (mJ)	Time overhead	Energy overhead
adpcm dec	61	3.2	105%	118%
adpcm enc	11	0.6	117%	154%
epic dec	31	1.3	104%	157%
g721 dec	293	15.4	102%	105%
g721 enc	347	18.2	101%	104%
gsm dec	80	4.2	102%	119%
gsm enc	175	9.2	121%	128%
jpeg dec	5	0.3	151%	315%
mpeg dec	2 307	121.4	100%	101%

TABLE II
OVERHEAD COMING FROM THE USE OF SOFTWARE TRANSLATION INSTEAD OF HARDWARE ACCELERATORS. AS A BASELINE WE RAN THE APPLICATION IN HYBRID-DBT SYSTEM, USING THE THREE HARDWARE ACCELERATORS.

Table II lists the overheads in execution time and in the energy consumed while using software translation instead of the hardware accelerators. We can see that the use of the software translation process increases execution time from 0% up to 51% and the energy consumed from 1% up to 215%. The increase in energy can be explained by two phenomena: i) for benchmarks where execution time is small or where code is large, the energy used for translating the code will have an important impact on the overall energy consumed (see `jpeg dec, adpcm enc and epic dec`); ii) for benchmarks with a longer execution time, the increased translation time will make all optimizations to happen later than in the hardware accelerated version, which leads to lower performance and higher energy consumption.

Results presented for this experiment focus on the benefits of the hardware accelerators compared to a software implementation. A detailed discussion of the possible benefits of the hardware accelerated tool-chain is provided in section VII.

C. Performance of the DBT framework

In previous sections, we demonstrated the usefulness of the different hardware accelerators. The goal of this experiment is to measure the performance of the generated code. All benchmarks have been executed with the Hybrid-DBT system (using hardware accelerators for the translation process) with different optimization levels. They are also executed on a Rocket core simulator to provide a baseline. Results of this experiments are provided on Figure 12. For each benchmark we can see the normalized execution time obtained for the different scenarios. We can visualize the impact of the different optimization levels.

The first observation is that the naive translation gives performance close to the native execution on a Rocket. For some benchmarks, the performance of the naive translation is much lower than the one of the Rocket core. This can be explained by the fact that some RISC-V instructions are translated into more than one VLIW instructions. If these insertions are localized in hotspots, it may have an important impact on the execution time. The naively translated binaries can also be faster than a native execution because of small differences in the pipeline organization: branches are not resolved at the same moment and the organization of pipeline lanes are different. Indeed, the in-order core have a single lane that can execute every instruction, with forwarding mechanism, while the VLIW core have specialized lanes.

Performance improvement from optimization levels 1 and 2 shows the impact of instruction scheduling on execution time. The difference between level 1 and 2 depends on how efficient the tool is at locating functions and performing inter-block optimizations. Developing those software transformations on the tool-chain was not a priority for our work. Consequently we believe that the performance for optimization level 2 could be improved with additional work on this part.

We can see that at the highest optimization level, Hybrid-DBT is on average 22 % faster than the native execution on the in-order core and up to 29 % for `jpeg dec`.

D. Area overhead

As we have shown before, the different hardware accelerators brought important speed-ups in the translation process. However, this is done at the expense of additional silicon. In this section we present the area cost of the different components of the design. These results are generated by Synopsis Design Compiler with a STMicroelectronics 28 nm technology. All values can be found on Table III.

We can see that the additional area coming from the different accelerators only represents 25 % of the VLIW area

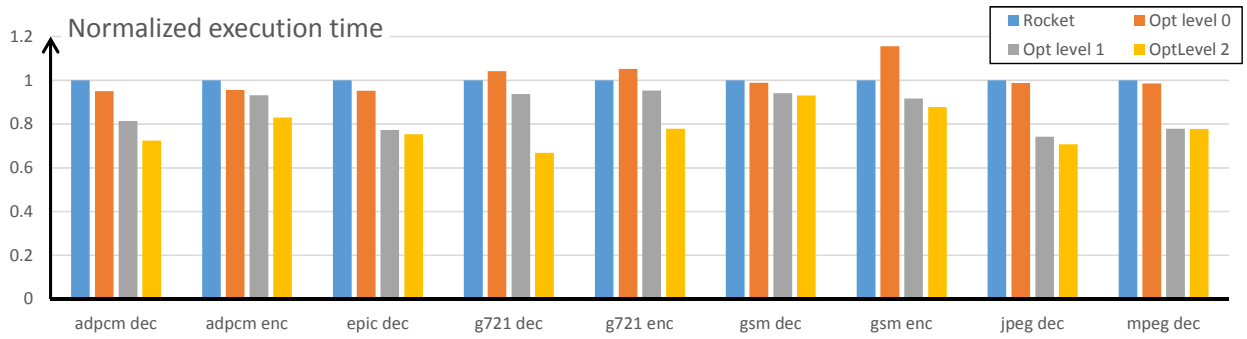

Fig. 12. Normalized execution times with different micro-architectures and different optimization levels.

Component	Area (μm^2)
VLIW	106 621
First Pass Translator	6 146
IR Builder	15 374
IR Scheduler	7 155
Rocket	30 792

TABLE III
AREA OF THE DIFFERENT COMPONENT IN HYBRID-DBT SYSTEM.

and the Rocket core (which is used as a DBT processor in this setup) represents 30 %. If this DBT system were used in a multi-VLIW system (as pictured in Figure 1), the area overhead would be smaller. We believe that in the context of dark-silicon, this overhead is acceptable in exchange for the reduction in cost shown in previous experiments.

VI. Related Work

The work presented in this paper lies at the interface of several topics/fields: processor architecture, run-time systems, optimizing compiler techniques, etc. In this section, we discuss how our approach compares with existing work in these different fields. We first present several work addressing the issues raised by multi-ISA heterogeneous multi-core architectures. We then discuss some of the most relevant works in dynamic compilation and more specifically DBT. The last part of the section focuses on the Transmeta and Denver architectures, which share many similarities with our approach.

A. Heterogeneous architectures

Historically, Dynamic Voltage and Frequency Scaling (DVFS) is used to provide dynamic adaptation. The system is able to dynamically modify the voltage and the frequency of the architecture according to the needs in performance. This method has the interest of being simple to model, as DVFS has a direct impact on frequency and power consumption.

With the end of Dennard scaling, the use of hardware heterogeneity (accelerators, DSPs, etc.) has proven to be an efficient way to adapt the energy/performance of a system. However, developing applications for these kind of systems and dynamically mapping tasks to the different execution resources is challenging.

Kumar et al. proposed to use single-ISA heterogeneous multi-core [5], [6]. The binary compatibility between the different cores was the enabling characteristic: development of applications as well as dynamic task migration became accessible. This idea has been used in the Arm big.LITTLE architecture [1], which features an aggressive 3-issue out-of-order processor for high-performance (called ‘big’) and a 2-issue in-order core for energy efficiency (called ‘LITTLE’). More recent architecture proposed up to three different types of cores to offer more trade-offs to the end-user [7].

Using different kinds of micro-architectures offers different energy/performance trade-off. This trade-off depends on the application being executed: for data-intensive applications, the use of statically scheduled architecture (e.g. VLIWs, CGRAs) offers high-performance and high-energy efficiency. Some other applications are more suitable for OoO architectures. Combining OoO and inO architecture (as for the big.LITTLE) is convenient as they are both based on the same programming model (and on the same ISA). However, moving toward other micro-architectures, such as statically scheduled ones, is more challenging due to fundamentally different ISAs.

Different approaches have been proposed to enable the use of different micro-architectures in such a system:

- The first idea consists in reusing the schedule generated by an OoO processor in a inO processor [8], [9], [10]. This may require to modify the inO core consequently: for example in DynaMOS cores, the in-order core has been upgraded with a larger register file, a mechanism for constrained register renaming and a load-store queue for detecting errors on memory accesses [9].
- Finally, Dynamic Binary Translation has often been used: Transmeta Code Morphing Software and NVidia’s Denver Architectures can execute x86 or Arm binaries respectively on a VLIW architecture [2], [3]. A software translation and optimization process is in charge of generating the binaries for this micro-architecture. Similarly, other approaches proposed to translate binaries toward CGRAs to increase performance and energy efficiency [11], [12], [13]. However, as code generation for CGRA architecture is a difficult task, they use an approach with a tightly coupled OoO core and a CGRA, mapping only hotspots on the CGRA. CGRA is not used as a core but as an accelerator for the OoO architecture.

Task migration overhead may be critical in such system. Indeed, even if binaries do not need to be translated, task migration induces a large number of cache misses (qualified as cold misses) when starting its execution on the new core. Some authors have therefore investigated polymorphic processors, that is clusters of processors sharing resources (storage or execution) between cores [14], [15], [16], [17]. Different

paradigms are used to exploit these architecture: ρVex [14], Voltron [17] and TRIPS [16] rely on specific instruction sets to support their architectures. The executable binaries can be used transparently on different configurations. On the other hand, Ipek et al. exploit OoO mechanisms to generate dynamic schedules for their configurable core [15].

B. Dynamic compilation

The idea of dynamic compilation is to translate and optimize source code into binary instructions at run-time. The most widely known dynamic compilation framework is probably Oracle Hostpot, the Java virtual machine Just-in-Time (JIT) dynamic compiler [18]. Other widely used JIT compilers include Microsoft CLR [19], for executing .NET applications, and Google's v8 compiler for executing Javascript. The first two are based on a split-compilation process where a first static compilation stage generates a low-footprint bytecode which is then translated at run-time. In contrast, Google v8 compiler operate directly at the source level.

All these tools face the same problem of cold-code execution: when the tool first translates a code snippet, it cannot guess whether this snippet will be executed often enough to recoup the cost of an optimization stage. This problem is often identified as one of the bottlenecks of dynamic compilation [20], [21]. To handle this issue, most dynamic compilation tools are decomposed into several optimization levels which are triggered whenever a region of binaries becomes hot. Our approach addresses the cold-code issue by taking advantage of hardware accelerators to speed-up (by two orders of magnitude) the first steps of the translation.

An interesting approach is the use of a platform specific bytecode within a split compilation framework: applications are first analyzed statically by the compiler framework to produce a custom bytecode embedding additional information (on memory dependencies, on vectorization opportunities, etc.). This bytecode is then used as input to the run-time translation stage, which takes advantage of this information to increase the performance of generated code. For example, Nuzman et al. [22] propose to statically compute vectorization opportunities in the original binary and then encode this information in a custom bytecode format. Thanks to this, the bytecode can be optimized for the vector extensions available in the processor. Our approach is different in its goals: we use DBT to provide binary compatibility with a preexisting instruction set, as ISA legacy is a key factor even for embedded systems.

DBT is a subset of dynamic compilation, but where the compiler operates directly from binaries. DBT is mainly known for its use in fast simulation of instruction set architectures (e.g. QEMU [23]), inter-generation portability (e.g. IBM Daisy [24], [25]) or inter-ISA portability (e.g. Apple Rosetta). DBT often translates binaries from an ISA to another one but some tools perform DBT targeting the same ISA in order to perform an analysis (e.g. Valgrind) or for continuous optimization (e.g. DynamoRIO [26]). This is very different from our work which aims at improving performance.

Dynamic Binary Translation is also used to execute generic binaries (e.g. ARM, x86) on a different kind of micro-architecture. There are two famous products exploiting this idea: Transmeta's Code Morphing Software (CMS) [2] and NVidia's Denver architecture [3]. They both execute x86 or ARM binaries on a VLIW architecture transparently, in order to reduce the power consumption. They are also based on a multi-stage translation and optimization process, but very few details are given on their actual implementation. Those two approach will be discussed in details in Section VI-C.

It should be noted that several work have addressed the problem of dynamic compilation for VLIW architectures. For example, Agosta et al. [27] propose to translate Java bytecode into VLIW binaries. Their approach uses a list-based instruction scheduler combined with a greedy register allocation (our register allocation stage shares many similarity with their work). Dupont de Dinechin [28] proposed to translate .NET bytecode into VLIW binaries, but instead use scoreboard scheduling algorithm, but only little information is provided on how register allocation is performed (the paper only suggests that the register allocation algorithm is more complex than the greedy heuristic used by Agosta).

C. Hardware acceleration

To the best of our knowledge, there were very few attempts to accelerate dynamic compilation using a combination of hardware and software. This subsection discusses Transmeta's CMS [2] and NVidia's Denver [3], which both deploys HW/SW co-designed platform based on DBT. It also discusses the work from Carbon et al. [29] which used dedicated hardware to accelerate LLVM JIT compilation.

Transmeta's Code Morphing Software has been co-designed along the VLIW core used as a target [2]. Although no direct hardware acceleration is provided, they modify their target processor to simplify speculation in their DBT system. This was achieved through the use of shadow registers and special rollback mechanisms to support various style of of speculation. In our approach, we accelerated the optimization process.

The approach followed by NVidia with the Denver processor [3] is slightly different. In the Denver system, the host VLIW architecture is a in-order 7-issue processor. It embeds a complete ARM8 ISA hardware decoder which can decode up to two instructions per cycle. These decoded instructions are then stored in a dedicated buffer, and later used by the processor. This decoding bears some similarity with our first-pass translator. Another difference is that Denver uses an in-order dynamic instruction scheduler during this translation step. As a consequence, the system can start exploiting some ILP right after the first translation. Additional ILP can later be obtained by the Dynamic Code Optimization engine, which computes a static schedule to fully exploit the ILP capabilities of the underlying VLIW core. In our approach, the execution after the first-pass translation does not take advantage of ILP, this is compensated by the fact that our IR-scheduler is fast thanks to hardware acceleration (a new instruction can be translated/scheduled every 4.2 cycles).

Finally, Carbon et al. studied haw to use specialized hardware to accelerate LLVM JIT compilation [29]. They designed a specialized interface for using red-black trees and modified some of LLVM algorithm to use these data-structure as much as possible.

VII. Discussion

In this work, we have developed from scratch a complete hardware/software co-designed system for executing RISC-V binaries on a VLIW processor. Although somewhat similar approach have been proposed by industry (NVidia Denver, Transmeta Crusoe), these systems are completely closed with very little information on their actual implementation. In contrast, our framework is open-source, and is meant to enable academic research on the possibility offered by dynamic binary translation for VLIW.

A. Toward continuous optimizations

The experimental study we have conducted demonstrates that the use of hardware accelerators reduces the cost of the translation process by two orders of magnitude. Even on a system with perfect caching (i.e. where the full translation can be stored on the translation cache), the use of accelerators leads to speed-ups as the flow is more aggressive. On a real-life system, our approach could lead to a higher speed-up.

In their work, McFarlin et al. concluded that the performance advantage of OoO processors against statically scheduled architectures mainly comes from the ability to speculate aggressively [30]. Their experimental study shows that, by replaying dominant schedules as if they were static schedules, the execution can reach 80% of the performance of normal OoO. We believe that this performance gap could be reduced through a continuous optimization process, at the price of additional HW mechanisms on the VLIW core. The static instruction schedule would be iteratively improved by speculating on memory dependencies and branch resolution.

Such an approach would be impractical in the context of a software DBT, as the instruction scheduler would quickly become a performance bottleneck. In contrast, thanks to our hardware accelerator, our DBT flow can re-schedule a RISC-V instruction in 4 cycles (compared to roughly 600 cycles for a software implementation), making such a strategy viable, and paving the way for very aggressive DBT back-ends.

Other continuous optimizations may benefit from this reduced scheduling time. For example, Hybrid-DBT has already been used to exploit dynamically reconfigurable VLIW processor [31]. Thanks to power gating, the issue width and the size of the register file can be modified dynamically. Due to this, each configuration of the VLIW needs its own binaries to work properly. In this work, Hybrid-DBT is used to dynamically explore the different configurations of the VLIW and to pick the most suitable one for each function of the application. If the operating conditions are modified and if the system wants to change the trade-off between energy and performance, the tool-chain picks a new VLIW configuration for each function of the application and generates new binaries accordingly.

It is also to note that the use of a hardware accelerated DBT process reduces the penalty from cold-code execution, and thus increases the ability to quickly start executing an application. We believe this is also a very important issue, as for such machine, the effect of cold-code translation has often been reported to impact user experience.

Finally, the hardware acceleration reduces the need for caching previous translations. Indeed, previous work on dynamic compilation uses a translation cache to store generated binaries [2], [3]. However, for large applications or for systems where many different applications are executed at the same time, the size of the translated binaries becomes huge. Reducing the cost of re-generating the binaries also reduces the interest of caching them. This may result in a reduction of the size of the translation cache.

B. Limits of Hybrid-DBT

There are some features that would be of primary importance for a commercial tool but that are not currently not handled in this version of Hybrid-DBT. The support of precise exception or self-modifying code, for example. For the latter one, accelerating the translation process is of great interest as such system often re-translate portions of binaries that have been modified. Since these issues are not performance critical, and do not pose significant research challenges, we made the choice of not supporting them for now.

The current version of Hybrid-DBT lacks a full translation cache system. Designing an efficient translation cache policy for such an accelerated DBT is not trivial, as the trade-off between performance and cache size is more complex due to hardware acceleration. We believe that this is an interesting research direction, which is part of our ongoing work.

Retrospectively, it turns out that one of the main performance bottleneck lies in an insufficient amount of middle-end compiler optimizations in our flow (constant propagation, more aggressive superblock construction, etc.). Although such optimizations are classical text-book algorithms, their efficient and correct implementation within a DBT framework remains quite challenging, and require very significant engineering efforts, which go well beyond what can be achieved in the scope of a PhD work.

C. The key role of High-Level Synthesis

Implementing a functional compiler back-end using application specific hardware represents a major design challenge, that would normally be out of reach of an academic research project. Yet, we were able to implement a fully functional hardware accelerated DBT stack and prototype it on an FPGA platform (Altera DE2-115) to check for its functional correctness. We see this as a significant achievement, that was only made possible through the systematic use of state of the art High-Level-Synthesis tools (Catapult-C) instead of classical HDL/RTL design flows. To our opinion, this demonstrates both the maturity and importance of such tools.

As a side note, the idea of implementing compiler using custom hardware is not new: for example, the Symbol computer project, which took place in the 60s/70s, aimed at building a machine capable of executing program specified in high-level programming languages[32]. The project turned out to be extremely difficult to design and debug. As a matter of fact, it is considered by many as a turning point in computer system designs, corresponding to the decline of hardware solutions and the rise of software implementations.

VIII. Conclusion

In this paper we presented Hybrid-DBT, an open-source, hardware/software co-designed DBT system targeting VLIW cores. The use of three hardware accelerator drastically reduces the cost of critical parts of the translation process and impact the performance of the architecture. Other optimizations are done on software, using a dedicated low-power core. Our experimental study demonstrates that i) the use of hardware accelerators are two orders of magnitude faster than their software counterpart; ii) this speed-up in the translation process induces a reduction of the total execution time.

Future work will focus on improving software optimization in order to increase peak performance. The integration of this flow in heterogeneous systems will also be studied.

References

[1] P. Greenhalgh, "Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," *ARM White Paper*, 2011.

[2] 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*, pp. 15–24, IEEE Computer Society.

[3] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's First 64-bit ARM Processor," vol. 35, no. 2, pp. 46–55.

[4] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers, "Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, IEEE.

[5] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *MICRO-36*.

[6] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pp. 64–75.

[7] H. T. Mair *et al.*, "A 20nm 2.5GHz ultra-low-power tri-cluster CPU subsystem with adaptive power allocation for optimal mobile SoC performance," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 76–77.

[8] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, "Mirage Cores: The Illusion of Many Out-of-order Cores Using In-order Hardware," MICRO-50 '17, ACM.

[9] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, "DynaMOS: Dynamic Schedule Migration for Heterogeneous Cores," MICRO-48.

[10] C. Villavieja, J. A. Joao, and R. Miftakhutdinov, "Yoga: A Hybrid Dynamic VLIW/OoO Processor," 2014.

[11] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1208–1213, ACM.

[12] A. Brandon and S. Wong, "Support for Dynamic Issue Width in VLIW Processors Using Generic Binaries," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 827–832, EDA Consortium.

[13] M. A. Watkins, T. Nowatzki, and A. Carno, "Software Transparent Dynamic Binary Translation for Coarse-Grain Reconfigurable Architectures," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 138–150.

[14] A. Brandon, J. Hoozemans, J. van Straten, and S. Wong, "Exploring ILP and TLP on a Polymorphic VLIW Processor," in *30th International Conference on Architecture of Computing Systems*.

[15] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pp. 186–197, ACM.

[16] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," ISCA '03, ACM.

[17] H. Zhong, S. A. Lieberman, and S. A. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 25–36.

[18] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the Java HotSpot™ Client Compiler for Java 6," vol. 5, no. 1, pp. 7:1–7:32.

[19] J. J. Gough and K. J. Gough, *Compiling for the .NET Common Language Runtime*. Prentice Hall PTR.

[20] S. Hu and J. E. Smith, "Reducing Startup Time in Co-Designed Virtual Machines," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, IEEE Computer Society.

[21] E. Borin and Y. Wu, "Characterization of DBT Overhead," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*.

[22] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks, "Vapor SIMD: Auto-vectorize once, run everywhere," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 151–160, IEEE.

[23] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," ATEC '05, pp. 41–41, USENIX Association.

[24] K. Ebcioğlu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," ISCA '97, ACM.

[25] K. Ebcioğlu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic Binary Translation and Optimization," vol. 50, no. 6, pp. 529–548.

[26] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pp. 265–275.

[27] G. Agosta, S. Crespi Reghizzi, G. Falauto, and M. Sykora, "JIST: Just-In-Time scheduling translation for parallel processors," pp. 239–253.

[28] B. Dupont de Dinechin, "Inter-Block Scoreboard Scheduling in a JIT Compiler for VLIW Processors," pp. 370–381.

[29] A. Carbon, Y. Lhuillier, and H.-P. Charles, "Hardware acceleration for Just-In-Time compilation on heterogeneous embedded systems," in *ASAP'13*, IEEE.

[30] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the Dominant Out-of-order Performance Advantage: Is It Speculation or Dynamism?," ASPLOS '13, pp. 241–252, ACM.

[31] S. Rokicki, E. Rohou, and S. Derrien, "Supporting runtime reconfigurable VLIWs cores through dynamic binary translation," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2018*.

[32] D. R. Ditzel, "Reflections on the High-Level Language Symbol Computer System," vol. 14, no. 7, pp. 55–66.

Simon Rokicki obtained the BSc and MSc degrees from ENS Rennes, France, in 2012 and 2014, respectively. He is currently working toward the PhD degree in computer sciences at the University of Rennes, under the supervision of Steven Derrien and Erven Rohou. His research interests include embedded systems architecture, dynamic compilation and HW/SW co-design.

Erven Rohou obtained his PhD from the University of Rennes 1 in 1998. He was a post-doc at Harvard University in 1999. He spent 9 years working in R&D at STMicroelectronics before joining Inria in 2008. He is now a senior researcher and the head of the PACAP project-team. His research interests include static compilation, JIT compilation, and dynamic binary optimization.

Steven Derrien obtained his PhD from University of Rennes 1 in 2003, and is now professor at University of Rennes 1. He is also a member of the Cairn research group at IRISA. His research interests include High-Level Synthesis, loop parallelization, and reconfigurable systems design.