Hardware-Accelerated Dynamic Binary Translation

Simon Rokicki
Université de Rennes 1/IRISA

Erven Rohou
INRIA/IRISA

Steven Derrien
Université de Rennes 1/IRISA

***Abstract*—Dynamic Binary Translation (DBT) is often used in hardware/software co-design to take advantage of an architecture model while using binaries from another one. The co-development of the DBT engine and of the execution architecture leads to architecture with special support to these mechanisms. In this work, we propose a hardware accelerated Dynamic Binary Translation where the first steps of the DBT process are fully accelerated in hardware. Results shows that using our hardware accelerators leads to a speed-up of 8× and a cost in energy 18× lower, compared with an equivalent software approach.**

I. INTRODUCTION

Dynamic binary translation (DBT) consists in translating – at runtime – a program written for a given instruction set to another instruction set. Dynamic Translation was initially proposed as a means to enable code portability between different instruction sets and can be implemented in software or hardware [1], [2]. DBT is also used to improve the energy efficiency of high performance processors, as an alternative to out-of-order microarchitectures. In this context, DBT is used to uncover instruction level parallelism (ILP) in the binary program, and then target an energy efficient wide issue VLIW architecture. This approach is used in Transmeta Crusoe [3] and NVidia Denver [4] processors.

Since DBT operates at runtime, its execution time is directly perceptible by the user, hence severely constrained [5]. As a matter of fact, this overhead has often been reported to have a huge impact on actual performance, and is considered as being the main weakness of DBT based solutions. This is particularly true when targeting a VLIW processor: the quality of the generated code depends on efficient scheduling; unfortunately scheduling is known to be the most time-consuming component of a JIT compiler or DBT system (this fact is also confirmed by our experiments, in Section IV).

Improving the responsiveness of such DBT systems is therefore a key research challenge. This is however made very difficult by the lack of open research tools or platform to experiment with such platforms. In this work, we address the two aforementioned issues:

- As opposed to closed commercial solutions such as Crusoe or Denver, we propose an open hardware/software platform supporting DBT. This platform was designed using HLS tools and validated on a FPGA board. Our DBT uses MIPS as host ISA, and targets a 4-issue VLIW architecture[1].
- We show how custom hardware accelerators can be used to improve the reactivity of our optimizing DBT flow. Our results show that, compared to a software implementation, our approach offers speed-up by 8× while consuming 18× less energy.

The remainder of this paper is organized as follows, Section II provides some background on DBT and describes our approach from a system-level point of view. Section III details our proposed DBT hardware accelerators, both from an algorithmic and architectural point of view. Section IV presents our experimental results, and Section V discusses related work. Conclusion and future work directions are sketched in Section VI.

II. PROPOSED APPROACH

In this section, we detail our approach to dynamic binary translation and describe its corresponding hardware platform. This platform consists in a hardware-software co-designed DBT framework which operates on MIPS binaries (for convenience) and targets a custom VLIW core.

We first start by providing background information on DBT systems, and follow by an overview of our platform and its main component. We then describe the different stages involved in our DBT strategy, and explain our choice of resorting to hardware acceleration for certain DBT stages.

A. Dynamic Binary Translation

As mentioned in Section I, DBT is traditionally intended to enable efficient execution of non-native binaries. For example, the QEmu [1] software enables the execution/emulation of ARM, MIPS, or PowerPC code on Intel x86 hardware. In some cases, DBT can be directly supported by the underlying hardware. For example the IBM Daisy [2] processor proposed to use DBT to offer binary compatibility between different generations of VLIW processors (with different issue widths).

More recently DBT was also used as a means to improve the energy efficiency of high performance processors. Instead of relying on a power-hungry out-of-order superscalar microarchitecture, DBT is used to uncover the instruction level parallelism (ILP) available in the binary program, and take advantage of this ILP to target a more energy efficient VLIW micro-architecture. Such an approach was first implemented, for the x86 instruction set, in the Transmeta Crusoe processor [3]. The more recent NVidia Denver and Denver 2 processors [4] also take advantage of this technique, but instead target the ARM instruction set. For both Crusoe and Denver processors, the DBT process is directly managed by the hardware, and is therefore invisible to the user.

Cold code execution is often a major limitation of DBT systems. Indeed, for some applications, most of the code will be executed only a few times. Cycles spent optimizing such code are wasted, since they will never be recouped by faster future invocations. For this reason, DBT systems often define several levels of optimization, starting from a fast translation of source binaries to start execution as soon as possible [5], [2]. When the system determines that a sequence of instructions has been – and will likely be – executed often enough, this sequence is further optimized with the hope that the induced speed-up will recoup the cost of this optimization. This idea is very similar to the way the HotSpot JIT compiler works [6].

[1]The whole platform (source code, and HDL for IPs) is availlable at https://github.com/srokicki/HybridDBT.

The computational cost of these optimization levels is critical in a DBT system as they determine when/if a given code fragment should be optimized. In this context, running these optimizations faster becomes particularly helpful.

B. Hardware platform

A system level view of the platform is provided in Figure 1. The platform is built around a VLIW core (⑤) which serves as the execution engine for the translated/optimized binary. This core is loosely based on the Vex architecture [7] and uses its own instruction format. Our VLIW uses 4 issues, with a non-clustered 64×32 register file. The instruction pipeline has 4 stages.

In order to minimize the impact of the DBT process on the system performance, we make the choice running the DBT engine on a separate customized architecture. The main benefit of this approach is that DBT can be run in parallel with the main program execution, enabling transparent ahead-of-time translation of code.

This customized architecture is based on a small footprint single-issue in-order processor dedicated to the *DBT* process (①). This DBT processor is enriched with three hardware accelerators: the *first-pass translator* (④) is in charge of generating a first version of the VLIW code; the *IR builder* (③) generates an Intermediate Representation (IR) of the source binaries; and the *VLIW scheduler* (②) uses this IR to perform instruction scheduling and to generate code for the target VLIW instruction set.

These accelerators make the DBT process faster and more energy efficient compared to a software-only implementation. When no DBT is required, the system is able to shut-down this specialized hardware using power-gating, preserving the power efficiency of the platform.

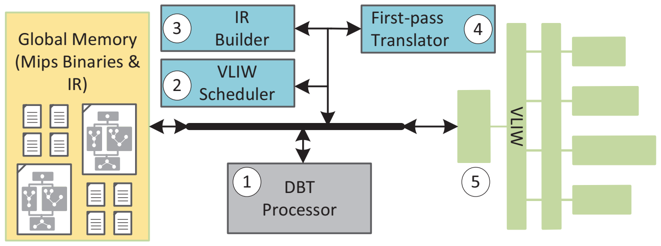

Fig. 1. Overview of the hardware DBT platform.

C. Run-time management

Dynamic Binary Translation needs to be both reactive and efficient: it needs to start executing the target program as soon as possible and needs to fully exploit the VLIW capabilities on heavy workloads. For this reason, our DBT framework is organized into several optimization levels. Lower optimization levels generate less efficient code, but execute much faster. Conversely, a frequently executed piece of program/instruction sequence will undergo more aggressive optimizations.

Currently, our framework uses two optimization levels, described below.

1) Cold-code execution: When a new section of binary code is to be executed by the processor, the system has no means to determine if the target code can benefit from aggressive optimizations (neither can it determine if the cost of these optimizations will be easily recouped). Moreover, to ensure the system reactivity, it has to start the execution quickly.

For this reason, the system will first perform a straightforward (i.e. syntactic) translation of MIPS binaries into VLIW binaries without trying to exploit instruction-level parallelism. For this, the DBT processor takes advantage of the *first-pass translator* which is fully implemented in hardware. The design of this *first-pass translator* is described in details in subsection III.

The use of an accelerator for the first-pass translator is a key feature to minimize the cost of cold code execution.

2) Heavily-used code: Cold code is instrumented to collect information about frequently executed fragments. Such instrumentation is hidden as much as possible in empty slots of the VLIW code. When hotspots are detected, they are considered for more aggressive optimization. We then build a higher-level Intermediate Representation (*IR*) of the fragment, appropriate for further processing.

Our main motivation while designing this IR was to make the scheduling step easier to implement in hardware, but also enabling a finer grain DBT process, which could operate at the basic-block level, without having to reconsider the whole procedure.

For each procedure, the IR contains a list of blocks as well as their successors in the control-flow graph. One block has a maximal size of 256 instructions (this impacts design choices in the accelerators, as we will see in subsection III-C). Instructions are encoded using a 128-bit format which also contains intra-block dependency information (e.g. IDs of the instruction predecessors and successors are directly encoded in the instruction). This information is then directly used by the instruction scheduler to extract ILP.

This IR is generated by the *IR builder* accelerator, whose operating principle is similar to that of the first-pass translator. However, while scanning instructions within a basic-block, it also keeps track of intra-block register-level def/use information. The accelerator then builds the extended instruction representation accordingly.

Once the IR is generated, the VLIW scheduler is called on each basic block independently. It operates on the block IR and schedules instructions according to available resources and instruction dependencies. More details about its implementation are provided in subsection III-C. The generated code can then be executed by the VLIW core.

It is important to note that IR generation can only be done when block boundaries are known. This information is collected by the *first-pass translator*. For this reason, when an indirect jump is met that invalidates boundaries, the system has to invalidate the generated IR and to start again with the correct boundaries.

Building the IR and extracting ILP in hardware make the overall DBT significantly faster, hence more reactive because there is less overhead to recoup in future invocations. As a consequence, application code benefits from higher optimization level earlier.

III. Hardware accelerators

A hardware/software co-designed DBT system is obviously a very complex design, which would be out of reach of an academic research project if designed at RTL level using HDL or similar tools. In this work, we therefore made the choice of designing our system (almost) entirely with Mentor Catapult-C High Level Synthesis technology [8]. This choice helped us focus our effort on key issues: minimizing the accelerator latency, while maintaining resource cost at reasonable levels.

We first developed the transformation in C and then performed the following modifications:

- Transforming array of structures into structures of arrays, and use array indexing as a replacement for pointers.
- Increasing memory bandwith through array and memory bank partitioning.
- Pipelining all loops with minimal Initiation Interval. This requires some refactoring to merge memory accesses and some explicit forwarding mechanisms along with pragma annotations to guide the HLS tool.

In the following we describe the three accelerators (*first-pass translator*, *IR builder*, *VLIW scheduler*) that are implemented in our DBT system. Our description addresses both hardware and algorithmic design choices, and their impact on performance and resource cost.

A. The First-pass translator

The role of the *first-pass translator* is to parse MIPS instructions and, for each of them, extract opcode, registers and immediate data. This information is then used to regenerate native VLIW instructions. Since not all MIPS instruction have a VLIW equivalent, they must be lowered into a sequence of native VLIW instructions. For example, `bltz` will be translated into `cmplt` and `brez` in the VLIW instruction set. In addition, whenever the accelerator identifies a jump or call instruction, it stores the source and destination addresses in a dedicated memory. The content of this memory is later used by the IR builder to identify block-boundaries. From a hardware point of view, this is easily implemented using Finite State Machines, and does not represent a real design challenge. For example our design takes between one and two cycles per MIPS instruction (a more detailed quantitative analysis is provided in Section IV).

B. The IR builder

The main difference between the *first-pass translator* and the *IR builder* is that the first one performs a straightforward translation while the second one has to construct a more complex representation including instruction level dependencies.

To obtain this information, we use a simple algorithm that keeps track of registers def/use relations as it iterates over the instructions of a basic-block. Our algorithm works as follows: for each of the 32 MIPS registers, the IR builder stores within a table the following information:

- `lastWriter` corresponds to the Id (its offset within the block) of the last instruction that wrote in that register.
- `lastReads` contains the list of IDs of the last three instructions that read that register, along with 2-bit value used to indicate how many actual reads are stored. The choice of the maximum number of stored reads is a trade-off: storing more reads may lead to more accurate dependencies but also increases the maximal number of dependencies added for a single instruction, which is the bottleneck of the IR builder, as we will see below.

Each analyzed instruction has at most one write in a register and two reads. The left part of Figure 2 shows different steps (labelled ① to ⑤) and accesses to entries (dashed arrows) needed to handle one read or one write.

When the accelerator analyzes an instruction which **writes in a given register**, it will gather the IDs of the last writer and the last three readers of the register (①). Then the current instruction is marked as the last writer of the register (②). The accelerator will then flag up to three dependencies in the IR: one from each previous reader, and if there is no reader, one from the last writer (② to ⑤). Note that adding a dependency requires reading the IR and updating the value on the next cycle, which involves two memory accesses.

When the accelerator decodes an instruction which **reads a given register**, it will gather the IDs of the last writer and the three last readers of the register (①). While handling a read, the accelerator only adds one dependency in the IR (② and ③). If the maximum number of readers in `lastReads` is reached, the oldest of the previous reader is evicted and will be the source of the dependency. Otherwise, the source of the dependency is the last writer. Finally, the current instruction is added in `lastReads` (④).

The accelerator also uses a similar mechanism to preserve the order of loads and stores on memory. However, as store instructions do not write registers and load instructions only read one register, they do not increase the maximal number of dependencies which are added for a given instruction.

Our design is heavily pipelined to minimize the execution time of the IR building phase. To eliminate the memory port conflicts that limit the pipeline through-put, we used distinct banks for all fields and for the IR. This is illustrated in Figure 2 along with the pipelined schedule of our accelerator (for two reads and one write). The right left part represent memory accesses (dashed arrows) required to handle a read or a write in a register. Using a dual-port memory for storing the IR and two single-ported banks for `lastWriter` and `lastReads` helped us reach an initiation interval of 5 (e.g. a new instruction is treated every 5 cycles).

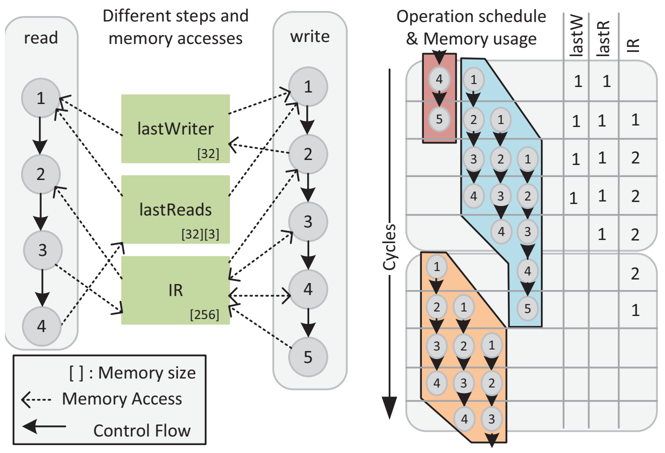

Fig. 2. High-level view of the IR builder. The left part represents steps to handle reads and writes in registers, the right part represents a schedule of these different steps.

C. The VLIW scheduler

The key component of our hardware accelerated DBT consists of the *VLIW scheduling* accelerator, which implements a greedy scheduling algorithm similar to list-scheduling. The scheduler operates on blocks exposed in the IR and produces a sequence of VLIW bundles.

Figure 3 depicts the internal memory organization within our accelerator, which consists of five distinct memory blocks:

- `IR`: memory containing the IR of the current instruction block to be executed,
- `sortedList`: sorted list containing all the instructions that are ready to be scheduled, sorted by priority,

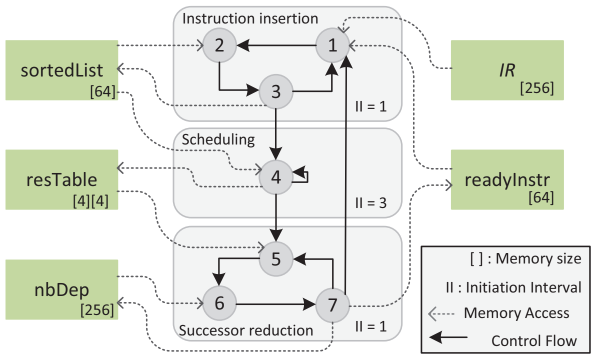

Fig. 3. Simplified view of our hardware VLIW scheduler.

- `resTable`: reservation table which keeps track of all pending instructions,
- `nbDep`: number of dependencies of each instruction,
- `readyInstr`: FIFO containing instructions which have no more dependencies and have to be inserted in the sorted list.

Scheduling is decomposed into three main stages:

1) The *instruction insertion* stage reads a ready instruction (Figure 3, ①), finds a location (start or end) in the list (②) and inserts the instruction (③) at this location.
2) The *scheduling* stage retrieves, from the sorted list, the instruction with the highest priority, and allocates it to an available functional unit.
3) Finally, the *successor reduction* stage considers every successor of all the finishing instructions in the reservation table (⑤), reads their current number of dependencies (⑥), decrements it and if this value reaches 0, inserts the instruction in the `readyInstr` FIFO (⑦).

All the data-structures used by the VLIW scheduler are stored in a custom memory block within the accelerator. Figure 3 indicates the size of each of these blocks. Their size is directly related to the design choices of the IR format. The size of `IR` and `nbDep` memory blocks corresponds to the maximal size allowed for blocks. If a block with more than 256 instructions is met, it will simply be split into two blocks, which may lead to lower performance. The size of the `resTable` memory block is related to VLIW structure: for example, with a 4-way VLIW with a 4-cycle pipeline depth, the reservation table stores up to 4×4 instructions. In contrast, the size of `sortedList` and `readyInstr` blocks are arbitrary and are therefore design-time parameters: reducing this size saves area but may also impact performance if they get full.

IV. Experimental results

We have evaluated our approach both in terms of cost and performance. In our context, performance refers to (i) the impact of DBT optimizations on application execution speed, and (ii) the DBT reactivity (i.e. DBT execution time). As far as cost is concerned, we considered both silicon area and energy efficiency. The section starts by a short description of our hardware platform and is followed by our performance and cost results.

A. Hardware prototype

As mentioned in Section II, we have developed a fully functional hardware prototype of our system. Since a detailed description of the platform is not possible given the current page limits, this document only provides a short description of its structure and its components.

TABLE I
Number of instructions and blocks of benchmarks

Benchmark	instr.	blocks	Benchmark	instr.	blocks
`dct`	257	3	`adpcm_e`	477	77
`matmul`	36	7	`adpcm_d`	462	68
`fft32x32s`	234	14	`crc`	167	27
`fir`	50	7	`bcnt`	137	7
`motion`	210	30	`blit`	159	23

Our platform is prototyped on a Altera based board (DE2-115 with Cyclone IV). The design uses a combination of Altera IPs (memory, Avalon bus interconnect, NIOS processor) and custom hardware components designed using Catapult-C HLS tool. All our custom hardware IPs communicate through small on-chip shared memory banks, and we used a NIOS-II processor as DBT processor for ease of development.

This prototype can operate at clock speed up to 41 MHz. The critical path of the design is caused by our VLIW processor IP (the DBT accelerators can run faster)[2]. We do not provide FPGA result since they are not relevant, and refer the reader to section IV-D where we discuss 65 nm ASIC area results.

B. Benchmarks

We validated our flow over a set of 10 kernels taken from the Mediabench suite [9]. These were compiled for a MIPS target with GCC 4.0.2 at `-O3` level. Table I provides the number of instructions and the number of blocks of these benchmarks.

C. DBT reactivity improvements

Our first set of experiments aims at quantifying the impact of hardware acceleration on the reactivity of our DBT system. To do so we used hardware monitors to obtain accurate timing information for each one of our two optimization levels. We collected results for two scenarios: one assuming that the DBT is executed directly on the VLIW core (this is our baseline, without dedicated processor and accelerators), the other one using our hardware accelerators.

Figure 4 summarizes the results that we obtained for a set of 10 benchmarks. Results show that the average execution time for the *first-step translator* is reduced by two orders of magnitude (between 115×–180× faster). These results are not surprising, given the type of operations (bitlevel manipulation and comparison) involved in this stage. For the *IR builder* stage, we also observe impressive speedup with performance improvements in the 30×–34× range. Interestingly, our *VLIW scheduler* brings only a “modest” speedup of 5×. This may question the relevance of using hardware acceleration for this stage. However, it is important to remember that scheduling represents more than 50% of DBT workload in pure software. According to Amdahl’s law, improving scheduling performance is therefore as important as for other steps.

In addition to performance improvements, we also propose to quantify the average number of cycles needed by the DBT system to translate a single MIPS instruction. For the optimization level 0, the *first-pass translator* requires only an average of 1.2 cycles per instruction. In other words,

[2]We believe this can be considered a good result given our choice of a (rather old) FPGA technology, and the fact that almost all the IPs (but the NIOS-II) were designed with HLS.

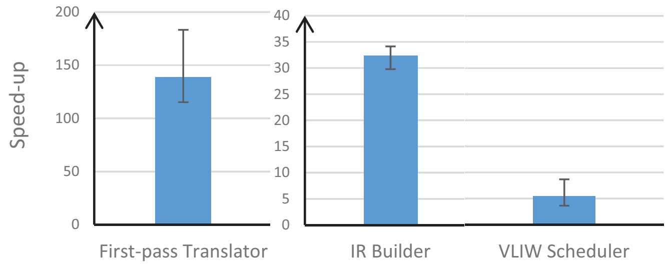

Fig. 4. Speed-up from accelerators (higher is better; log scale). First-pass (opt. level 0) benefits the most from hardware acceleration (over 100×), generating non-optimized code, but providing reactivity. Acceleration is less dramatic for VLIW Scheduler (opt. level 1), still very significant, in the order of 5.5×.

DBT can be very reactive if the user agrees to sacrifice performance. Results for the optimization level 1 are illustrated in Figure 5. They show that the *IR builder* and the *VLIW scheduler* require an average of 111 cycles per instruction, an improvement of almost one order of magnitude over a software only implementation. As mentioned in SubSection II-C, this is an important improvement, as the application code will benefit from higher optimization level much earlier.

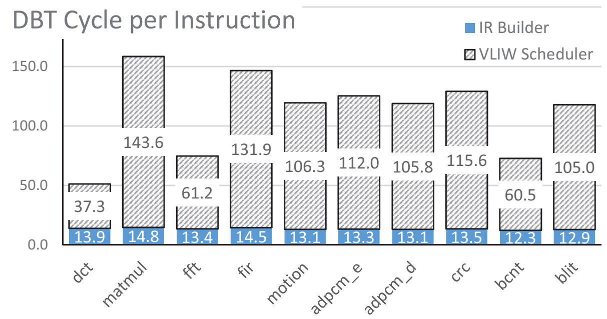

Fig. 5. DBT cycle count to reach different optimization levels

D. DBT area cost

Custom hardware accelerators obviously come at a price in terms of resource usage. Since FPGA resource costs are not really relevant, we have collected area costs for 65 nm ASIC using Design Compiler for all our IPs including the VLIW core. Our results, illustrated on Figure 6, show that the total area overhead of our DBT system is equivalent to that of a 4-issue VLIW core. Note that the DBT platform is able to generate code for wider issue VLIWs (up to 8-issue), so its relative cost can be even lower.

Also, to the difference of the VLIW core, the scheduler algorithm involves complex control structures (chained lists) and exhibits a highly data dependent behavior. It is widely recognized that for such algorithms, even modern High Level Synthesis tools still fail at offering quality of results comparable to hand written RTL code[3]. Our prior experiment with these tools suggest that an area-optimized RTL implementation is likely to reduce the area by a 2× to 4× factor. This would of course come at the cost of a development effort which is out-of-reach for a typical academic research project.

E. Energy efficiency improvement

In order to measure the improvement of our platform in the energy cost of a DBT, we run power and energy estimation for our DBT system (the DBT processor and the different

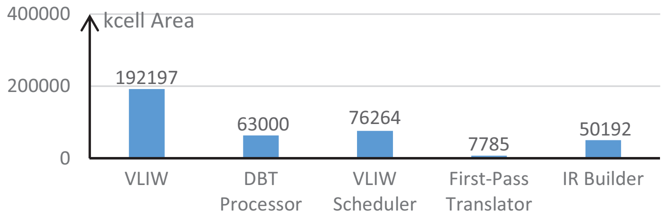

Fig. 6. Area utilization of different entities in the platform.

accelerators) assuming a 65 nm ASIC technology as target. The design was synthesized with Design Compiler with a target frequency of 250 MHz. Power estimations were obtained from Gate-level simulations of the whole system.

To quantify the energy benefits of our hardware assisted DBT, we compared hardware and software energy cost of our three DBT stages. Figure 7 represents the energy efficiency improvement offered by the hardware accelerators. We observe that, when executed using custom hardware, the first-pass translation (resp. the IR building) uses 250× (resp. 78×) less energy than their VLIW software counterparts. Similarly, the hardware accelerated scheduling step is 11× more energy efficient.

Since one of the motivations behind the use DBT based processor architecture is energy efficiency, reducing the energy cost of DBT itself is of prime importance.

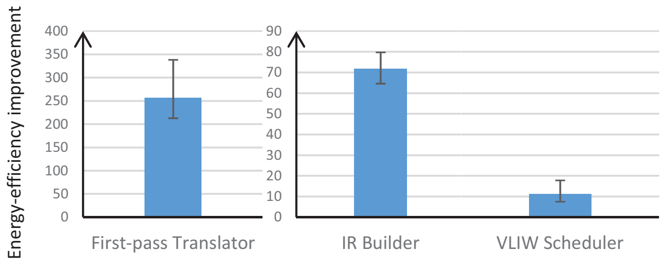

Fig. 7. Energy-efficiency (ratio of consumed energy non-accelerated vs. accelerated, higher is better; log scale) improvement from accelerators. Behaviors are similar to speedups (Figure 4): first-pass benefits from a dramatic improvement, making initial code generation very cheap. Further optimizations levels still benefit from significantly improved energy efficiency.

F. Performance of DBT generated code

Since the aim of our optimizing DBT system is to improve performance, we also evaluated how much performance improvement we could obtain by taking advantage of the VLIW core. The results that we obtained are provided in Figure 8. They show that an average of 80 % performance improvement can be achieved against the non-optimized DBT (i.e. with only first-pass stage activated) for our 4-issue VLIW core.

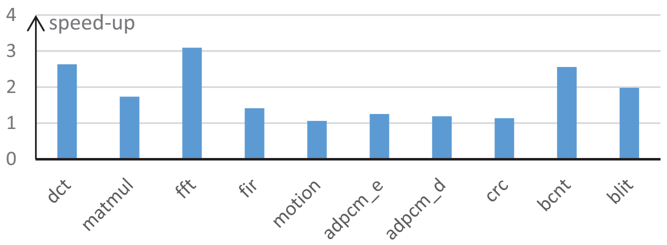

Fig. 8. Speed-up with optimization level 1 compared to level 0

Our DBT engine at optimization level 1 has the capability to deliver speed-ups up to 3× the naive code generator. Several benchmarks do not benefit such performance, for two main reasons: first, good performance on VLIW requires large basic

[3]This is especially true for area, modern tools are more competitive when it comes to latency and clock speed.

blocks; second, the MIPS binary we take as input has fewer registers, which introduces false dependencies. In future work, we will perform register allocation and apply optimizations such as loop unrolling and superblock formation. Thanks to the flexibility provided by the DBT processor, these optimizations will be rather straightforward to implement.

V. Discussion and Related Work

Transmeta Crusoe [3] and NVidia Denver [4] architectures are the two most similar works we found. They both aim at a transparent execution of legacy binaries on a VLIW architecture. Their hardware/software co-designed system is customized to handle efficiently main challenges of the DBT.

Transmeta co-designed the Crusoe architecture and the Code Morphing Software. It proposes hardware mechanisms for speculation and recovery, precise exception management, handling memory mapped I/O and self-modifying code [3].

NVidia's Denver architecture uses a 7-issue VLIW-like architecture. In order to execute ARM binaries, the processor includes an ARM ISA decoder that can decode up to two instructions per cycle. These decoded instructions are translated into an extended Internal Representation (IR) used as an input to the optimizer. This IR is stored in an internal cache and can be modified by another process in charge of continuous optimization. Whenever the code is executed again, optimized code is used instead of ARM instructions, resulting in a better utilization of all seven execution units available [4]. Our approach shares common points with Denver architecture: the use of an intermediate extended IR and the use of customized hardware for handling cold-code execution.

One of the main challenges of DBT is its startup time (e.g. performance on cold-code). For example, IBM Daisy [2] requires up to 4k instructions to translate a single source instruction. The solution to reduce start-up time is to define several optimization levels: cold-code is optimized only once, while hotspots go through upper optimization levels, to produce more efficient code, at higher price. The same idea drives JIT compilers [6]. However, there are several approaches on how to handle the first optimization level, which will be critical for cold-code execution.

A common approach, is to start execution with code interpretation [10], [6]. This removes translation overhead when executing cold code but leads to inefficient execution. Moreover, this loss of efficiency will never be recouped because future execution of the same code will still require interpretation.

Another common approach is to generate a first naive translation of the binaries [3], [11], [12], [5]. Contrary to interpretation, this approach requires a translation overhead which is as small as possible. However, this overhead may be recouped because future execution of the same piece of code will use this translated code.

Finally, several approaches choose to use (low-performance) native execution on their first optimization stage [4], [13]. Their proposed architecture offers ways to execute natively the binary code with low performance and to translate and optimize hotspot for another more efficient microarchitecture (here VLIW).

In our work, we generate a first naive translation of the code for cold-code execution. However, our main contribution is to go further by offering a second cheap optimization level, which produces much more efficient code at reasonable price, thanks to dedicated hardware accelerators.

Our approach also has some connections with earlier works on JIT compilers targeting VLIW machines, which aimed at defining very fast, yet efficient scheduling heuristics [14], [15]. These two approaches identified instruction scheduling as one of the main bottlenecks when dynamically generating binaries for VLIW. They both use greedy algorithms to schedule instructions. Transmeta's Code Morphing Software also used greedy instruction scheduling on its first optimization level but switched to a smarter algorithm when generating more efficient code [11].

Last, VLIW support in DBT was also addressed by Michel et al. [16], who studied extensions needed to handle VLIW processors. Their goal largely differs from us since they aim at the dynamic translation of VLIW ISA (e.g. TI C6x DSPs) with x86 ISA as a target (for fast simulation purposes).

VI. Conclusion

We propose a hardware-accelerated DBT platform targeting VLIW architectures. Our approach exploits customized hardware to reduce the overhead of generating scheduled code and make DBT more reactive. Translation of a binary section is $8\times$ faster and consumes $18\times$ less energy than an equivalent pure-software approach. The FPGA prototype of the platform as well as all C/C++ sources are available on GitHub to ease reproducibility and encourage future work/experimentation on hardware/software co-design DBT systems. Future work will focus on implementing several basic optimization passes to extract more ILP from MIPS binaries and study how they interact with hardware accelerators.

References

[1] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX ATC, FREENIX Track*, pp. 41–46, 2005.

[2] K. Ebcioğlu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," in *ISCA*, 1997.

[3] 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 *International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2003.

[4] D. Boggs, G. Brown, N. Tuck, and K. Venkatraman, "Denver: NVIDIA's First 64-bit ARM Processor," *Micro*, 2015.

[5] S. Hu and J. E. Smith, "Reducing Startup Time in Co-Designed Virtual Machines," in *ISCA*, 2006.

[6] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot Server Compiler," in *JVM Research and Technology Symposium*, 2001.

[7] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW Approach to Architecture, Compilers and Tools*. Elsevier, 2005.

[8] T. Bollaert, "Catapult Synthesis: A Practical Introduction to Iterative C Synthesis," in *High-Level Synthesis: From Algorithm to Digital Circuit*, 2008.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *MICRO*, 1997.

[10] H.-S. Kim and J. E. Smith, "Dynamic Binary Translation for Accumulator-Oriented Architectures," in *CGO*, 2003.

[11] D. Ditzel, "Experience with Dynamic Binary Translation." ISCA AMAS-BT Keynote, 2008.

[12] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 Execution Layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium®-based Systems," in *International Symposium on Microarchitecture*, MICRO 36, 2003.

[13] Y. Wu, S. Hu, E. Borin, and C. Wang, "A HW/SW Co-designed Heterogeneous Multi-core Virtual Machine for Energy-Efficient General Purpose Computing," in *CGO*, 2011.

[14] B. Dupont De Dinechin, "Inter-Block Scoreboard Scheduling in a JIT Compiler for VLIW Processors," in *Euro-Par Parallel Processing*, 2008.

[15] G. Agosta, S. Crespi Reghizzi, G. Falauto, and M. Sykora, "JIST: Just-In-Time scheduling translation for parallel processors," *Scientific Programming*, vol. 13, no. 3, 2005.

[16] L. Michel, N. Fournel, and F. Pétrot, "Fast Simulation of Systems Embedding VLIW Processors," in *CODES+ISSS*, 2012.