Supporting Runtime Reconfigurable VLIWs Cores Through Dynamic Binary Translation

Simon Rokicki, Erven Rohou, Steven Derrien

Supporting Runtime Reconfigurable VLIWs Cores Through Dynamic Binary Translation

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

***Abstract*—Single ISA-Heterogeneous multi-cores such as the ARM big.LITTLE have proven to be an attractive solution to explore different energy/performance trade-offs. Such architectures combine Out of Order cores with smaller in-order ones to offer different power/energy profiles. They however do not really exploit the characteristics of workloads (compute-intensive vs. control dominated). In this work, we propose to enrich these architectures with runtime configurable VLIW cores, which are very efficient at compute-intensive kernels. To preserve the single ISA programming model, we resort to Dynamic Binary Translation, and use this technique to enable dynamic code specialization for Runtime Reconfigurable VLIWs cores. Our proposed DBT framework targets the RISC-V ISA, for which both OoO and in-order implementations exist. Our experimental results show that our approach can lead to best-case performance and energy efficiency when compared against static VLIW configurations.**

I. Introduction

With the end of Dennard scaling, heterogeneous multi-cores (e.g. mixing embedded CPUs and DSPs) have proven to be an attractive approach to explore better trade-off between performance and energy. However, such heterogeneity has also many drawbacks : (i) programming is more challenging, (ii) dynamic workloads balancing (using task migrations) is much less flexible.

To address those shortcomings, hardware vendors propose to hide architectural heterogeneity to programmers and runtimes through a homogeneous programming model by using the same ISA for all cores. This is the case of the ARM big.LITTLE architecture [1] which combines within a single platform high-performance Out of Order cores (big) with simpler, lower power in-order micro-architectures (LITTLE). Thanks to binary compatibility between cores, programming and runtime management are greatly simplified.

Although the combined use of heterogeneity and Dynamic Frequency Voltage Scaling enables subtle performance/energy trade-offs, such a platform does not really take advantage of the diversity encountered in workloads. For example modern embedded application workloads consist of many hotspots which can range from control-dominated kernels to compute-intensive ones. Whereas OoO cores are a perfect match for the former, the later could make a better target for a statically scheduled (i.e VLIW-based) micro-architecture.

In this work, we advocate the use of a new type of heterogeneous multi-core, in which we enrich the OoO/in-order heterogeneity by introducing VLIW cores following the NVidia's Denver philosophy. VLIW cores help processing compute-intensive workloads for a significantly lower energy budget than for their OoO counterparts. In this paper, we name our architecture FAT.Tall.skinny, as a tribute to the ARM big.LITTLE brand name.

Introducing a statically scheduled VLIW processor obviously breaks the single ISA property: in VLIW cores, Instruction Level Parallelism (ILP) must be explicit in the binary code. We alleviate this issue by resorting to Dynamic Binary Translation (DBT). In our context, we use DBT to translate from a host ISA (RISC-like) to a guest VLIW ISA as in Transmeta's Crusoe and NVidia's Denver processors. Our current prototype uses the RISC-V as a host ISA, and can target several variants of a VLIW core loosely based on the ST200 from STMicroelectronics.

To the difference of Transmeta and Denver, where the target VLIW guest architecture is fixed, we take advantage of a runtime configurable VLIW core. Using this core, and thanks to power gating, we can activate/deactivate execution units and/or grow/reduce the register file size. We build on this feature to expose even more performance/energy trade-offs to the runtime by considering several hardware guest VLIW configurations in our DBT framework.

More precisely, during the DBT process, we search for the VLIW configuration that offers the best performance/energy trade-off under power constraint (TDP). Thanks to the flexibility offered by DBT, we can operate at various levels of granularity (application, function or loop level). In addition, the combined use of a DBT cache (and hardware acceleration) makes this search phase easy to recoup. The contributions presented in this work are the following:

- The modification of a DBT back-end to handle different VLIW configurations;
- A dynamic exploration of VLIW configurations, using profiling information and scheduling results;
- A strategy to pick a configuration for a function under some external constraints;
- An experimental validation and a demonstration of the benefits of our approach.

The remainder of this paper is organized as follows: Section II introduces all the necessary background on Runtime Reconfigurable VLIWs and DBT; Section III describes our modification on the DBT framework and our strategy to select a VLIW configuration; the experimental study in Section IV demonstrates the benefits from the use of the dynamically adaptable VLIW and provide some comparisons of the dif-

ferent strategies. Finally, Section V compares our approach with previous methods and Section VI concludes this paper.

II. Background

In this section, we provide background on Single ISA multi-cores and DBT techniques, along with a short description of the runtime-adaptable VLIW core used in this work.

A. Single ISA heterogeneous multi-cores

As mentioned previously, the ARM big.LITTLE multi-core integrates, within a single system, two types of cores. ARM big cores consist of aggressive Out of Order super-scalar processors (e.g. A15) aimed at performance demanding applications, where energy efficiency comes at a secondary goal. On the contrary, LITTLE cores are built out of low-footprint in-order execution engines (e.g. A7), and are much more energy efficient. All these cores use the same ISA, and applications/threads can therefore seamlessly migrate from one type of core to the other. A similar approach is used by NVIDIA in the Denver processor, where big cores are based on VLIW architecture and where binary compatibility is achieved through DBT (see next paragraph).

All these systems are proprietary, and very little is known about their actual implementation. Besides, the fact that these platforms are closed strongly hinders research on the topic. The recent efforts around the RISC-V instruction set may be an answer to that issue. For example the open-source Rocket multi-core generator supports several types of cores (from simple 3 stage pipelines to aggressive OoO) making innovative research prototypes possible. Our work therefore naturally uses the RISC-V as a reference ISA.

B. Dynamic Binary Translation

Dynamic Binary Translation has mainly been used for portability purposes: fast simulation of an instruction set architecture (e.g. QEMU [2]), inter-generation portability (e.g. IBM Daisy [3]) or inter-ISA portability (e.g. Apple Rosetta).

DBT has also been used as a technique to improve performance and/or energy efficiency for legacy ISAs such as ARM and x86. In this context, the goal of DBT is to dynamically recompile a program from a host ISA (e.g. x86) to a more efficient guest ISA (e.g. VLIW based). Two commercial products are built on this idea. The Transmeta Code Morphing Software [4] was introduced in the early 2000s. The Crusoe processor was based on this technology, and the general purpose market (x86). More recently, NVidia introduced the Denver architecture [5] which follows the same principle, but uses the ARM ISA as host. Both enable the seamless execution of x86 (or ARM) binaries on a VLIW architecture, with the goal of improving energy efficiency. Of course, these are based on proprietary solutions for which very little information is available.

Dynamic compilation frameworks such as those used in Transmeta CMS and NVidia's Denver are decomposed in several optimization levels, which expose a trade-off between the DBT overhead and the optimization aggressiveness. The underlying idea is that each cycle spent optimizing code has to be recouped by the speed-up from the optimization. Consequently, the translation of cold-code[1] has to be as fast as possible. Conversely, it is expected that hotspots can benefit from very aggressive optimizations. Figure 1 depicts the different steps involved in a DBT translation and optimization flow targeting VLIW:

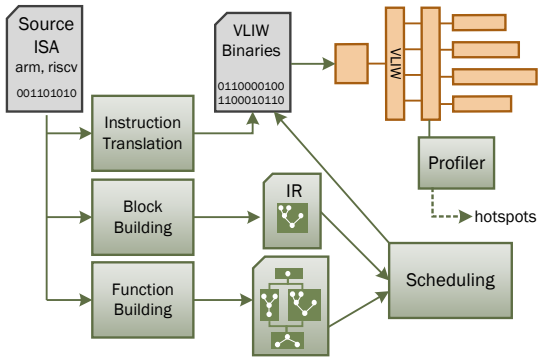

Fig. 1. General organization of a DBT framework targeting VLIW.

- **Instructions Translation** When a fragment of source-ISA binaries is executed, it will be translated one by one into VLIW binaries or sometimes interpreted by the framework. This first execution has to be done with the lowest overhead possible.
- **Blocks Building and Scheduling** When profiling information shows that a basic block is executed several times, it triggers the next optimization level. At this level, VLIW binaries are analyzed and translated into an Intermediate Representation (IR). This IR is then used to perform an instruction scheduling stage and generate VLIW binaries exploiting ILP.
- **Function Building and Scheduling** When profiling information indicates that a scheduled block belongs to a hotspot, the DBT framework performs a control flow analysis and builds the corresponding function. Then inter-block optimizations (e.g. trace-building, loop unrolling and function inlining) are performed to increase the available ILP.

To our knowledge, little academic work has been done on the topic, and the only open VLIW-DBT tool we are aware of is Hybrid-DBT[2]. Hybrid-DBT is an open-source DBT framework operating on the RISC-V host ISA and targeting VLIW architectures [6]. In this work, we built on this framework to handle and exploit a Runtime Reconfigurable VLIWswhich is described in the next subsection.

C. Power gating for VLIW cores

Power gating has been shown to be a very effective approach for reducing the energy dissipated by a processor datapath. The idea consists in deactivating parts of the processors that are not currently being used. Power gating can be controlled

[1] Code that has never been executed

[2] Available at https://github.com/srokicki/HybridDBT

directly by the processor control, or exposed in the ISA through specific instructions. The main challenge with this approach lies in deciding when and what to deactivate, with an additional difficulty stemming from the relatively long delays involved with activation/deactivation (Roy et al. [7] assume an overhead of 10 cycles).

For example, Roy et al. explored how to combine compiler and hardware support to exploit power gating of functional units in an OoO processor [7]. A similar idea was followed by Giraldo et al. [8] for VLIW processors. The technique consists in searching, at compile time, regions of code in which some functional unit is idle. Two additional instructions are then inserted to deactivate/reactivate the hardware block. Our approach somewhat differs from previous work, in the sense that we use power gating as a mean to expose several distinct VLIW configurations to the DBT framework, each of them with its own power/performance profile.

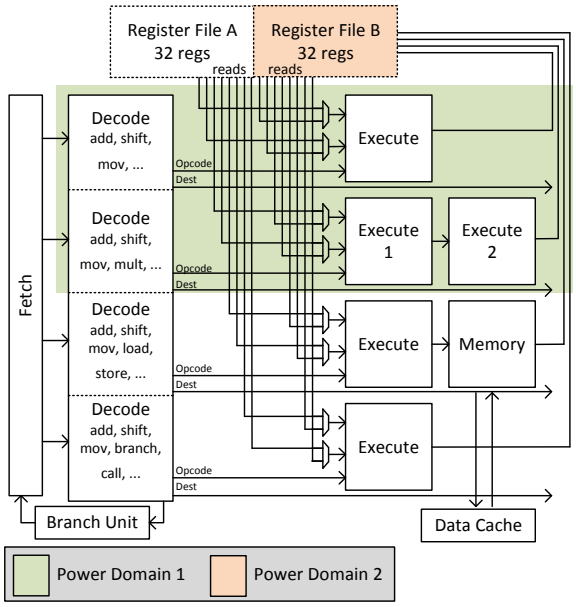

Fig. 2. Organization of the run-time reconfigurable VLIW. In this example, configuration can switched from 2 to 4 issues and from 32 to 64 register.

The VLIW processor we use in this work is loosely based on the ST200 processor. Figure 2 depicts our VLIW architecture, including its power domains (i.e. domains that can be switched off using power gating). The power domains used in this design offer the following configuration knobs:

- The VLIW can be set to an issue width of 2, 4, 6 or 8.
- Its register file may comprise 32 or 64 general purpose registers.
- Some specialized execution units may be deactivated: for example, we can expose a 6-issue VLIW with one memory pipeline and two multipliers or two memory pipelines and one multiplier.

Using these mechanisms, our VLIW core exposes 22 different hardware configurations to the DBT engine. The switch between hardware configurations is managed through specific machine instructions. Of course, these reconfiguration instructions are only exposed to the DBT runtime environment, to prevent any mismatch between the DBT output and the active configuration.

In this section, we introduced the reader to DBT techniques and presented our target architecture. The following Section details the main contribution of this work.

III. Proposed approach

In this section, we present how we use Dynamic Binary Translation techniques to take advantage of dynamically adaptable VLIW processors.

A. Integration on the Hybrid-DBT flow

As mentioned in Section II, Hybrid-DBT is a multi-staged translation flow. The various optimization levels operate at different scales (instruction, blocks, functions) and hence have very very different execution overheads. We recall that our goal is to dynamically adapt the VLIW configuration to the code being executed, and that reconfiguration is not instantaneous. As a consequence, we make the choice of applying this transformation at the function level.

We modified the Hybrid-DBT framework to handle different VLIW configurations: the instruction scheduler now generates binaries for various VLIW issue widths with more or less specialized execution units. It is to note that the size of the register file in the resulting configuration has some impact on the optimization performed by the flow: larger register files make unrolling and trace-based speculation more efficient.

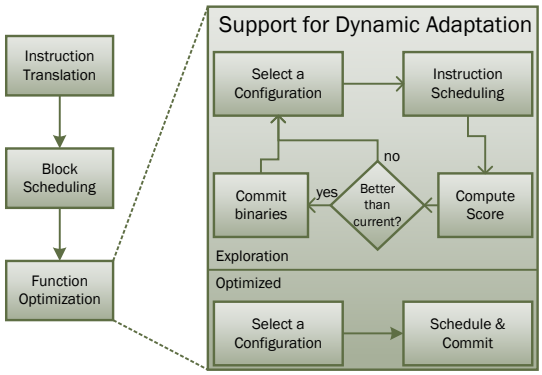

Fig. 3. Details on the flow handling dynamic adaptation

There are three different steps in the way dynamically adaptable VLIWs are used in the DBT framework:

- During the instruction translation, the DBT framework only translates source binaries one-by-one without trying to exploit ILP. When this code is executed, it will use a low-power configuration as it will not benefit from high issue-width. The size of the register file is fixed to 32 registers with a one-to-one mapping of the RISC-V registers.
- During the block scheduling, the framework has determined block boundaries and performed instruction scheduling to exploit ILP. In this step, the execution could benefits from higher issue-width. However, triggering VLIW reconfiguration for a single block would

be inefficient due to reconfiguration overhead. For this reason, the configuration is fixed to low-power mode.

- During the functions optimization, the DBT framework has detected hotspots and analyzed the control-flow to extract a function, several inter-block transformations are performed to merge blocks, build traces[3] and unroll loops. At this state of the execution, the DBT framework starts exploiting the adaptability. For each function found, the framework performs the two following steps:
  - First, it explores different configurations and analyzes the generated code to measure the efficiency of the configuration. This process is pictured in the `Exploration` part of Figure 3 and described in subsection III-B and III-C.
  - Then, when all configurations have been explored, the DBT framework will use the score of each configuration to dynamically decide which configuration to use for each function of the application. This choice will depend on external constraints which are given by the operating system. This mechanism is pictured in the `Optimized` part of Figure 3 and described in III-D.

In the next subsections, we will present the different steps involved in the dynamic exploration and selection of configurations.

B. Evaluating VLIW configurations

The DBT framework has to decide, based on runtime constraints, which VLIW configuration to choose for a given function. It does so by exploring several solutions to obtain the best trade-off between performance and energy consumption. We drive this search based on performance and energy models. Our performance model is based on a simple scoring function computed out of the generated binary, whereas the energy model is based on pre-computed data for each type of configuration.

Performance score: When a new configuration is tested and when the instruction scheduling has been done for all traces of a function, a score is computed to evaluate whether the configuration is a good match for the function at hand. The scoring function is the following:

$$score_c = \sum_{b \in B} \rho_b * size_c(b)$$

where $size_c(b)$ is the size of the schedule of block b using configuration c and ρ_b the rate of execution of b in the procedure. The score function is the sum of all schedule length multiplied by their weight in the procedure execution time.

Energy estimation: The DBT framework has access to the static estimation of the power consumption of each configuration. In our current prototype, we have built this table based on gate-level simulation of the architecture while running different benchmarks. By multiplying this average power consumption with the execution score (which can be seen as the average execution time of the procedure), we obtain a rough approximation of the energy consumed while executing the procedure on a given VLIW configuration.

[3]In the sense of Fisher's trace-based scheduling

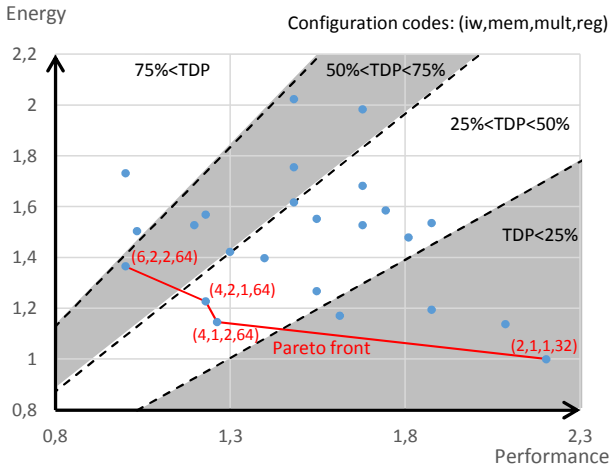

Fig. 4. Example of the trade-off available for matrix multiplication (values have been normalized)

As an illustration of the approach, Figure 4 shows all possible trade-offs for a matrix multiplication kernel (each point represents a distinct VLIW configuration). As we aim at optimizing both energy performance, there is no optimal operating point. It is however possible to derive Pareto optimums. In our example, one such optimum c is defined as follows: for any other point c' in the set, the performance of c' is greater than the one of c if and only if the energy consumption for c' is greater than the one of c. The configuration used for executing the function should always be among the Pareto optimums. Figure 4 highlights all Pareto optimums for the matrix multiplication as well as its associated configurations.

C. Dynamic exploration of configurations

Ideally, all configurations should be evaluated to be able to determine the Pareto front, the scores for different configurations are not independent of each other. In this subsection, we explain how the schedule for a given configuration can help choosing the next configuration to explore.

For example, when the framework schedules a block of 12 instructions with 8 memory instructions on a configuration having only 1 memory unit, the schedule will be at least 8-cycle long. The system can easily measure that the schedule is bounded by memory accesses and move to a configuration with more memory units. Adding multipliers or additional execution units will not increase performance but will increase energy consumption.

More precisely, for each type of instruction (normal, memory access and multiplier), we divide the number of instruction in the procedure by the number of available resources (size of the schedule multiplied by the number of execution units) to obtain a lower bound on the schedule length. Using these three scores, the system can determine the limiting factor and suggest a new configuration to explore.

Deciding whether we use 32 or 64 registers is more challenging. In our implementation, we consider that, when unrolling or when building execution trace, the use of additional registers will allow speculation. Moreover, when blocks have many arithmetic instructions, we can perform register renaming to remove false name dependencies.

D. Selection of a configuration

Once all different configurations have been explored and the Pareto front has been built, the system will still need to pick a configuration to use. As we said before, we cannot say that one configuration of the Pareto front is better than another, it is just a different trade-off. However, the system needs to be able to pick one and to determine if paying a certain amount of energy-consumption is worth the speed-up it brings.

In our current implementation, we decided that the DBT framework would receive two values from the system:

- The first value, **energy_ratio**, will define the ratio of energy consumption against performance. The framework will determine the point among the Pareto optimums that maximize the following formula $(1 - energy\_ratio) * score - energy\_ratio * energy$
- The second value, **max_tdp** defines a maximal power consumption for the system. If the first value is used to choose a point among the Pareto optimums, this one will exclude some point and may change the Pareto front. Figure 4 represents three domains corresponding to TDP limitations at 75%, 50% and 25% of the range of possible power consumption. We can see that while the limitations at 75% and 50% do not affect the Pareto front, the limitation at 25% add another point on the list of Pareto optimums.

These two values are left open to the OS which may use them to control the power management according to external constraints (battery autonomy, thermal information, high workload, critical task...).

IV. Experimental results

In this section, we present our experimental study to demonstrates the efficiency of the proposed approach. We first evaluate the improvement obtained from our approach against static VLIW configuration. We also determine the average efficiency of each configuration to determine if the VLIW design could be modified to remove unneeded configurations.

The experimental setup is the following: the VLIW was synthesized with Synopsys targeting ST Microelectronics ASIC 28 nm. Results indicate that the processor can run up to 750 MHz. To have a precise estimation of the power consumption, we performed a gate-level simulation of the design on several kernels using ModelSim. From this simulation, we obtained the average switching activity of each gate from which we derive an accurate estimation of the power consumption. This process has been followed for all the different VLIW configurations.

A set of benchmark applications taken from Mediabench suite was used to measure the efficiency of the proposed approach. These benchmarks are compiled into 64-bits RISC-V binaries using GCC 7.1. These binaries are then used as an input of our framework. The same binaries are used for all our experiments. For every benchmarks, the time and energy spent for the DBT process have been measured: because of Hybrid-DBT specialized hardware they are always below 1% of the total execution cost, even with the exploration of all configurations.

A. Impact on performance and energy consumption

The first experiment consists of evaluating the benefits offered by the combined use of Runtime Reconfigurable VLIWs and DBT. We define three baseline corresponding to the execution of binaries using Hybrid-DBT with a fixed VLIW configuration. These baselines correspond to a low-power 2-issue VLIW with 32 registers, a mid-range 4-issue VLIW with 64 registers and a high-end 8-issue VLIW with 64 registers. **With this experiment, we want to show that dynamic adaptation can lead to performance as high as the best static configuration while reducing the energy consumption.**

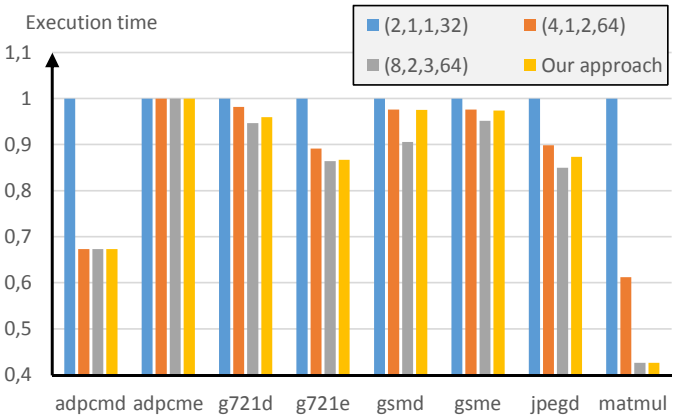

Fig. 5. Normalized number of cycles for the benchmarks application using different scenarios.

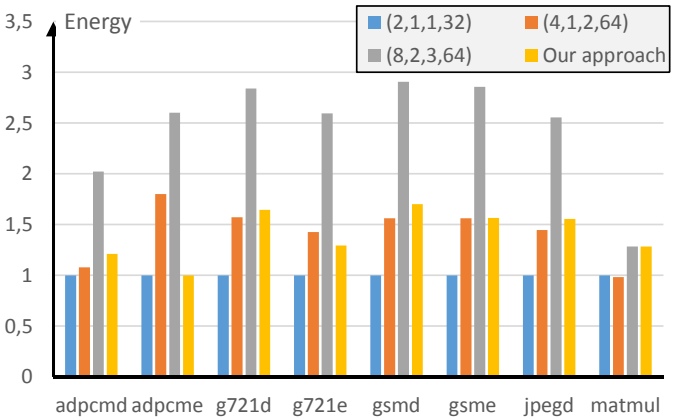

Fig. 6. Energy consumption for the benchmarks application using different scenarios (lower is better).

Figure 5 represents the number of cycles needed to execute each application on the four different scenarios. Each value

TABLE I
USAGE OF THE DIFFERENT VLIW CONFIGURATIONS.

Config	# use	Config	# use	Config	# use
(2,1,1,-)	35	(4,1,3,-)	3	(4,2,1,-)	26
(4,1,1,-)	8	(6,1,3,-)	1	(6,2,1,-)	2
(4,1,2,-)	35	(4,2,2,-)	10	(6,2,3,-)	1
(6,1,2,-)	3	(6,2,2,-)	11	(8,2,3,-)	7

has been normalized with the performance obtained with the 2-issue VLIW baseline execution. We can see here that our approach often performs as well as the best static VLIW configuration. The only exception is for `gsmd` application where analysis showed that current implementation did not detect the main hotspot.

Figure 6 pictures the energy consumed during the execution of these applications on each scenario. The energy consumption is obtained by multiplying the execution time by the average power consumption of the configuration used. Once again, these values are normalized against the value obtained with the 2-issue VLIW baseline. We can see that the energy efficiency is always higher than the one of the most-consuming configuration. **As it is expected, our approach does not reach the lowest energy consumption because it has been configured to favor performance in these experiments.**

From figures 5 and 6, we can also see that when high-issue configurations do not bring any speed-up (see `adpcme`), our framework will favor a low-energy execution resulting in a very high energy efficiency. Even when we reach same performance as for 8-issue VLIW (see `g721e`), our framework may find a different configuration which give the same performance while increasing the energy efficiency.

B. Utilization of the different configurations

In this subsection, we will measure how often each configuration is a Pareto optimum on all our experiments. This helps understands if some configurations are unused by the framework and to see if the VLIW design could be modified. Results are shown in Table I, where the configuration is named with 3 values: issue-width, number of memory units and number of multipliers. The last digit, which corresponds to the size of the register file, is left open here because we merged results for 32 and 64 registers. We can see that the framework tends to favor configurations with low-issue width. This comes from the lack of aggressive optimizations in the DBT framework. Another observation is that configurations with only one memory access and three multipliers are not often used. We could consider changing the architecture to remove this multiplier.

V. RELATED WORK

The Denver architecture from NVidia [5] shares many common points with our approach. Indeed, it will execute ARM binaries on an in-order processor. It is also used in a heterogeneous system environment but acting like a high-performance core. In this system, the Denver uses an ARM decoder for executing cold code and during the execution, another thread (which may run on one of the LITTLE cores) will optimize the code being executed. Even if few experimental studies were made on the performance of their in-order core against standard OoO cores, the architecture can execute up to seven instructions in parallel and may be more efficient than the usual OoO cores for single-thread applications.

Hybrid-DBT framework, on which we based our work, is a hardware accelerated DBT framework [6]. Three different accelerators have been designed: an instruction translator reduces the overhead of cold-code execution; a hardware instruction scheduler reduces the cost of continuous optimization as the scheduling step becomes cheap. In this work, we benefit from the reduced cost of instruction scheduling compared to software DBT: the exploration of different configuration was less energy-expensive and new version of the code were committed earlier in the execution.

We found few other tools for dynamic compilation targeting VLIW. We already mentioned NVidia's Denver architecture [5] and Transmeta CMS [4] which are completely closed. There is also the works from Dinechin and the one of Agosta et al. who developed Just-in-time compilers targeting VLIW [9], [10]. However, these tools are based on a bytecode (Java bytecode or CLI) and we favored the idea of single-ISA systems.

Brandon et al. developed Generic Binaries which were intended to enable the use of dynamically issue-width VLIW[11]. The idea is to generate VLIW binaries for an 8-issue while constraining the schedule to prevent read-after-write dependencies inside an instruction bundle. Consequently, each bundle could be split into two 4-instruction bundles or four 2-instruction bundles. However, this approach only supports changes in the issue-width and cannot handle changes in the size of the register file or to the number of specialized units (eg. multipliers or memory units). Moreover, their approach needs to recompile applications specifically for their purpose. The use of DBT allows to use legacy binaries and also offers the possibility of single-ISA heterogeneous platforms.

Finally, some works have been done to enable power gating of functional units in the context of OoO processors [12], [7]. However, these approaches require to statically analyze the code to find code regions where a functional unit is idle. Our approach can be applied transparently on existing binaries.

VI. CONCLUSION

In this paper, we presented how Dynamic Binary Translation could be used to handle Runtime Reconfigurable VLIWs. We modified the Hybrid-DBT framework and added support for different configurations in the instruction scheduler. The DBT framework will now explore different VLIW configurations at run-time, trying to optimize the energy consumption and the performance. Our experimental results show that our approach can lead best-case performance and energy efficiency when compared against static VLIW configurations. As far as we know, this approach is the first that enables the use of Runtime Reconfigurable VLIWs without requiring to recompile applications. This work enables the use of a new type

of heterogeneous multi-core using Runtime Reconfigurable VLIWs to execute compute-intensive workload with the best configuration possible.

As a future work, we plan to implement more optimization on Hybrid-DBT framework. This would increase the available ILP and increase performance of high-issue configurations. We also plan to study heterogeneous multi-core based on the proposed model of FAT.Tall.skinny. Indeed, a prototype of this architecture could be designed using existing open-source RISC-V cores [13], [14].

REFERENCES

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

[2] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX*, USENIX Association, 2005.

[3] K. Ebcioğlu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," ISCA '97, pp. 26–37, ACM, 1997.

[4] 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 *CGO*, 2003.

[5] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's First 64-bit ARM Processor," *IEEE Micro*, vol. 35, Mar. 2015.

[6] S. Rokicki, E. Rohou, and S. Derrien, "Hardware-Accelerated Dynamic Binary Translation," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1062–1067, Mar. 2017.

[7] S. Roy, N. Ranganathan, and S. Katkoori, "A Framework for Power-Gating Functional Units in Embedded Microprocessors," *IEEE Transactions on VLSI Systems*, vol. 17, pp. 1640–1649, Nov. 2009.

[8] J. S. P. Giraldo, L. Carro, S. Wong, and A. C. S. Beck, "Leveraging Compiler Support on VLIW Processors for Efficient Power Gating," in *ISVLSI'2016*, July 2016.

[9] B. Dupont de Dinechin, "Inter-Block Scoreboard Scheduling in a JIT Compiler for VLIW Processors," *Euro-Par'08*, 2008.

[10] G. Agosta, S. Crespi Reghizzi, G. Falauto, and M. Sykora, "JIST: Just-In-Time Scheduling Translation for Parallel Processors," *Scientific Programming*, 2005.

[11] 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*, 2013.

[12] S. Rele, S. Pande, S. Önder, and R. Gupta, "Optimizing Static Power Dissipation by Functional Units in Superscalar Processors," in *CC'02*, CC '02, (London, UK), pp. 261–275, Springer-Verlag, 2002.

[13] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," Tech. Rep. UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015.

[14] K. Asanović et al., "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.