On the Performance Effect of Loop Trace Window Size on Scheduling for Configurable Coarse Grain Loop Accelerators

Tiago Santos, Nuno Paulino, João Bispo, João M. P. Cardoso, João C. Ferreira
INESC TEC (Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência),
Faculty of Engineering of the University of Porto
Porto, PORTUGAL
{tiago.lascasas, nmcp, jbispo, jmpc, jcf@fe.up.pt}

2021 International Conference on Field-Programmable Technology (ICFPT) | 978-1-6654-2010-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICFPT52863.2021.9609868

Abstract—**By using Dynamic Binary Translation, instruction traces from pre-compiled applications can be offloaded, at runtime, to FPGA-based accelerators, such as Coarse-Grained Loop Accelerators, in a transparent way. However, scheduling onto coarse-grain accelerators is challenging, with two of current known issues being the density of computations that can be mapped, and the effects of memory accesses on performance. Using an in-house framework for analysis of instruction traces, we explore the effect of different window sizes when applying list scheduling, to map the window operations to a coarse-grain loop accelerator model that has been previously experimentally validated. For all window sizes, we vary the number of ALUs and memory ports available in the model, and comment how these parameters affect the resulting latency. For a set of benchmarks taken from the PolyBench suite, compiled for the 32-bit MicroBlaze softcore, we have achieved an average iteration speedup of 5.10x for a basic block repeated 5 times and scheduled with 8 ALUs and memory ports, and an average speedup of 5.46x when not considering resource constraints. We also identify which benchmarks contribute to the difference between these two speedups, and breakdown their limiting factors. Finally, we reflect on the impact memory dependencies have on scheduling.**

I. Introduction

Ever since CPUs stopped achieving significant single-thread performance gains, alternative ways to achieve higher performance in terms of execution time, such as adopting multi-core CPUs, have been sought [1]. At the same time, edge and embedded computing devices started to become widespread, which lead to increasing demands for energy efficiency. Finding hardware solutions that accommodate these performance targets is, therefore, of paramount importance. One of these solutions pertains to the usage of heterogeneous systems to accelerate applications, with System-on-a-Chip platforms (SoCs) comprised of a multi-core CPU and a a reconfigurable fabric (such as an FPGA) being of particular relevance. These SoCs allow for applications to achieve better performance in both metrics by utilizing the FPGA to accelerate performance-critical segments, achieving a lower execution latency at a fraction of the energy cost.

This adoption of generalized parallelism and heterogeneous computing, however, is difficult, as most applications are not developed with these characteristics in mind, and often cannot even be recompiled. Binary translation of applications arises as a solution to this problem by allowing a binary compiled for one specific Instruction Set Architecture (ISA) to be executed on another platform, be it a CPU with a different ISA or an FPGA accelerator. Binary translation can be either static [2] or dynamic (DBT) [3]. The former analyzes the program's executable, while the latter analyzes its execution trace.

This work focuses on DBT. We consider an heterogeneous system with a CPU and an FPGA, in which a program executes on the CPU and is dynamically accelerated by offloading trace segments to an accelerator on the FPGA. More specifically, we evaluate the translation of instruction trace windows, which capture multiple repetitions of a loop's basic block onto a Coarse-grained Loop Accelerator, which is a single row CGRA with a single fully connected row of ALUs and memory ports connected via crossbar, and with full access to the system memory [11]. We use a software model for this analysis, as this accelerator was already validated by an on-chip implementation [4], and focus only on the effects of scheduling different volumes of workload (by capturing larger instruction windows) onto the model, for different counts of available resources. We list schedule the repeated blocks onto our model, and vary both the number of repetitions and the model parameters (i.e., number of ALUs and memory ports). We evaluate the effect of an increasing number of repetitions on the total execution latency, as more repetitions of a loop's body may contain more operations independent from each other, which can execute in parallel. Our main focus is, then, to provide a study focused on how scheduling these multiple repetitions can increase the computational density on the accelerator, and how its functional units can scale for larger repetition values.

II. Related Work

Ansaloni et al. [7] explore how to optimize the mapping of computations onto a CGRA by mapping expressions rather than single operations. These so-called Expression-Grained Reconfigurable Arrays (EGRAs) have cells that can be reconfigured in order to implement a set of operations and their interconnects, and their study centers around exploring the granularity of these cells.

Lee et al. [8] focus on reducing the energy cost of modulo-scheduled CGRAs, which are also used to accelerate loops, by using compression in order to reduce the size of the CGRA's configuration. In particular, we note the simplifications performed over a data flow graph in order to prune data that is produced by some operations but never used by others.

Finally, Liu et al. [9] focus on accelerating loop nests with CGRAs, while using an automatic optimizer to find an appropriate resource configuration for a given application. That configuration is chosen through design space exploration using different properties retrieved from data flow graphs and the scheduling results.

III. Extraction and Scheduling of Repeated Basic Blocks

We retrieve instruction trace windows via our binary translation tools [5]. Decoded execution traces are fed dynamically into the framework's window extraction and analysis modules, wherein we implemented the instruction scheduling. For our analysis, the extraction of instruction windows is based on detection of loop iterations. The smallest valid window contains one loop iteration, i.e., basic block. Larger windows permit the capture of multiple repetitions, implicitly providing a given level of loop unrolling.

From a repeated basic block, we generate a data flow graph. This graph is comprised of vertices that represent arithmetic operations, memory accesses, registers and constants, and its edges represent the flow of data between the vertices. Figure 1 shows the data flow graph of a basic block with no repetitions. Our analysis steps compute the critical path length of the graphs (highlighted in red in Fig. 1), which is useful to establish an upper bound for an ideal scheduling scenario. By analyzing the basic block's inputs and arithmetic expressions, we are also able to establish guards to ensure that the iterations we repeat execute, or to safely abort if there is a violation.

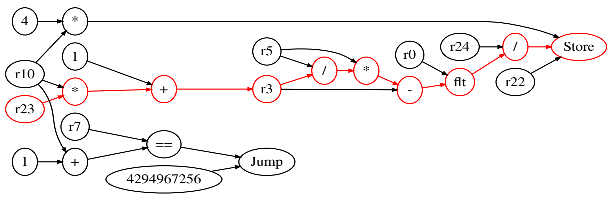

Fig. 1. Data flow graph of the first basic block detected in the 2mm benchmark, with the critical path highlighted in red

We apply several passes over the graph to remove information not relevant for scheduling, such as intermediary registers and constants, converting the data flow graph onto a dependency graph. The removal of these vertices can lead to disjointed subgraphs, as two separate chains of computations may occur completely in parallel, despite using the same input. We also remove the branching operations, as the checks previously determined help guarantee correctness. The final dependency graph contains only the vertices that incur on a latency cost and that can be scheduled onto the accelerator's resources (arithmetic operations and memory accesses), as well as the dependencies between these vertices.

Using the dependency graph, we can then schedule it onto a set of accelerator resources: Arithmetic Logic Units (ALUs), which can implement both integer and floating-point arithmetic operations, and memory ports, capable of implementing memory reads and writes. This accelerator has been described in our previous work [4]. Scheduling is done using a list scheduling algorithm, based on the definition provided by [6]. We order our vertices by their distance, in terms of total latency, to the roots of the graph, and with the number of successors as a tie-breaking heuristic. We have adapted the algorithm to use the two kinds of resources we considered (ALUs and memory ports), and accepts as parameters the different amounts of each resource, in order to generate different schedules.

IV. Exploration of Different Window Sizes

In order to perform our analysis, we select 18 benchmarks from the PolyBench benchmark suite, compiled for the MicroBlaze 32-bit softcore microprocessor, using GCC's optimization level *-O2* and floating-point datatypes. Execution streams are produced for each benchmark, and we find all basic blocks that repeat at least 5 times and with a maximum size of 200 instructions per repetition.

We evaluate the achievable speedups by list-scheduling up to 5 repetitions of each block for an optimistic case, where all memory accesses are considered to no alias, and for a conservative case, where all memory dependencies are enforced. We determine memory dependencies by analysis of the order that each memory operation should execute in order to preserve correctness in terms of Read-After-Write, Write-After-Read and Write-After-Write accesses.

Table I shows the result of this detection, using only a single repetition (i.e., the basic block itself). A total of 51 distinct basic blocks are detected, with an arithmetic mean of 11.9 instructions per block and a standard deviation of 4.5. Table I also shows that many benchmarks, such as *2mm*, *3mm* and *adi*, have multiple basic blocks representing loop iterations, while others, such as *trmm* and *doitgen*, only have a single basic block. On average, arithmetic operations outnumber memory accesses in a 3:1 proportion.

We build instruction sequences by repeating each basic block found up to 5 times (i.e., from 1 to 5 instances of the instruction sequence in each block). We then schedule the respective dataflow graphs.

For each dataflow graph ($51 \times 5 = 255$ in total), we perform scheduling under different resource constraints. All configurations are represented as $(ALUs, MemoryPorts)$ pairs. We use a configuration without resource limits as a theoretical upper bound, which is equivalent to the latency of the data flow graph's critical path. Finally, we choose five different configurations within these bounds, as shown in Fig. 2.

TABLE I

Summary of the basic blocks detected in each PolyBench benchmark

Benchmark	#Basic Blocks	Avg. Inst	Avg. Arith	Avg. Mem
2mm	5	12.20	9.80	1.40
3mm	5	12.20	9.80	1.40
adi	5	16.20	10.80	4.40
atax	3	8.33	5.67	1.67
bicg	3	11.33	7.00	3.33
covariance	3	8.33	5.33	2.00
doitgen	1	12.00	10.00	1.00
fdtd2d	1	19.00	15.00	3.00
gemm	2	10.50	6.50	3.00
gemver	4	11.25	6.50	3.75
gesummv	1	16.00	7.00	8.00
mvt	4	10.75	7.75	2.00
nussinov	3	6.67	4.33	1.00
symm	1	18.00	11.00	6.00
syr2k	3	15.67	10.00	4.67
syrk	3	11.67	7.33	3.33
trisolv	3	10.33	7.67	1.67
trmm	1	12.00	8.00	3.00
All	51	11.88	8.14	2.73

Avg. Inst Average Instructions per block, *Avg. Arith* Average arithmetic operations per block, *Avg. Mem* Average memory accesses per block

After scheduling the data flow graphs using all of these configurations, we then calculate the latency of an iteration, in cycles, by dividing the number of schedule stages by the number of repetitions. Then, we calculate the speedup on an iteration in relation to the results achieved by the original binary segment running on a MicroBlaze CPU with the same clock.

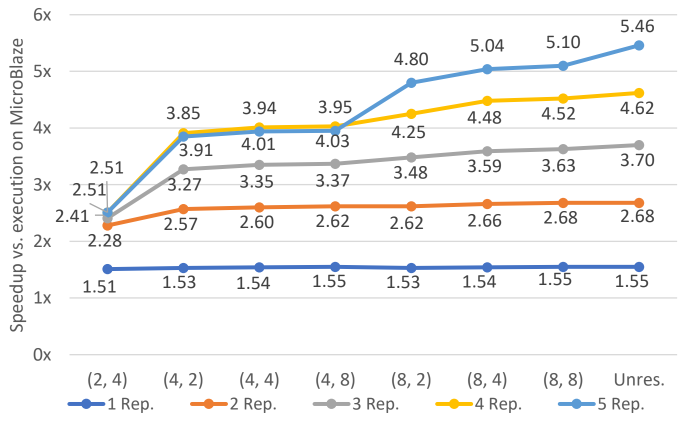

Fig. 2. Average speedup of a iteration across all benchmarks when using different repetitions and resource configurations

Figure 2 shows the average speedup achieved by all benchmarks, using each window size and each configuration, when measured against the latency of an iteration on a MicroBlaze CPU.

For a single iteration, additional resources do not lead to significant speedups. The largest speedup increase occurs between (2,4) and (4,2), for all repetitions. This can be explained by the 3:1 ratio between arithmetic operations and memory accesses we previously identified, as the (2, 4) configuration does not have enough ALUs to readily scale with so many arithmetic operations. By adding more ALUs using the (4, 2) configuration, arithmetic operations can be scheduled sooner, which reflects on the increased speedups. Increasing memory ports is not as critical, as our accelerator implements memory accesses with a latency of 2 cycles per access, while the latency of the ALUs may vary depending on the operation scheduled onto them.

TABLE II

Decrease in average speedup when considering all memory dependencies

Repetitions	(2,4)	(4,2)	(4,4)	(8,4)	(8,8)	Un.
1	4.0%	4.0%	4.3%	4.3%	4.4%	4.4%
2	19.2%	23.6%	23.9%	24.3%	24.4%	24.4%
3	22.1%	30.0%	30.8%	32.2%	32.4%	32.8%
4	23.5%	34.2%	35.2%	36.9%	37.0%	37.5%
5	23.5%	34.4%	35.1%	39.3%	39.5%	40.4%

Un. Unrestricted, *(ALUS, MemPorts)*

When it comes to the repetitions, we can observe that the speedup increases with a higher number of repetitions for these three configurations, but with a notable exception: using 4 repetitions leads to a slightly better speedup than 5 repetitions. While these configurations can schedule a DFG with 4 repetitions with good results, the number of resources on those configurations may not be enough to handle the extra operations and memory accesses of the additional repetition, which causes them to be scheduled later than ideal, and leading to a lower overall speedup.

As for configurations (8, 2), (8, 4) and (8, 8), there are still steady speedup gains for repetitions 3, 4 and 5. The extra resources present in these configurations successfully alleviate the restriction previously found when using 5 repetitions, as the number of available resources can now handle the added demand of having 5 repetitions. These configurations, however, may not be ideal, as an accelerator with these many resources starts to strain the FPGA's area and energy efficiency, while also adding additional complexity to the interconnects between different resources.

Focusing now on Tab. II, we can see the impact that considering all memory dependencies has on the achievable speedup gains. We compare our main approach, which assumes memory accesses do not alias, against a pessimistic approach where every Read-after-Write, Write-after-Read and Write-after-Write dependencies are assumed to alias. When considering these restrictions, we can see that speedups are, on average, between 20% and 40% lower than their optimistic counterparts.

At last, we notice that, for all repetition values except the first, the configuration with the highest amount of resources still cannot achieve an average speedup matching that of the unrestricted version, with the gap widening with higher repetition values. But since these speedups are an average between all benchmarks, we need to look at each benchmark individually: for 5 repetitions, we report that 36 basic blocks

TABLE III
BREAKDOWN OF THE RESOURCES NECESSARY FOR ALL BASIC BLOCKS TO REACH OPTIMAL SPEEDUP, WHEN CONSIDERING 5 REPETITIONS

Benchmark	BBID	(24, 20)	(30, 26)	Unrestricted
adi	BB3	6.04	6.04	6.04
adi	BB4	8.33	10.55	10.55
adi	BB5	10.37	10.37	10.37
bicg	BB3	10.23	10.23	10.23
covariance	BB3	5.31	5.31	5.31
fdtd2d	BB1	13.90	13.90	13.90
gemm	BB2	7.27	7.27	7.27
gemver	BB1	5.33	5.33	5.33
gemver	BB4	7.59	7.59	7.59
gesummv	BB1	10.00	10.48	10.48
symm	BB1	7.29	7.29	7.29
syr2k	BB2	8.29	8.29	8.29
syr2k	BB3	8.29	8.29	8.29
syrk	BB2	7.29	7.29	7.29
syrk	BB3	7.71	7.71	7.71

BBID Basic Block ID, *(ALUS, MemPorts)*

already achieve the optimal speedup with a (8, 8) configuration, with the remaining 15 basic blocks falling behind. We identify these basic blocks in Tab. III, and proceed to analyze how many more resources are needed to close the gap for all benchmarks. Using a configuration of (24, 20), we manage to stabilize the speedup of 13 basic blocks, with only the second basic block of *adi* and the only basic block of *gesummv* failing to do so by a small margin, and requiring a configuration of (30, 26) to finally converge.

V. CONCLUSION

This paper provides an analysis of how expanding our trace window to capture multiple repetitions of a loop's basic block can improve the efficiency of the hardware acceleration of those loops. We schedule these repeated basic blocks onto a Coarse-grained Loop Accelerator with different numbers of single-cycle ALUs and memory ports, in line with our previous hardware implementations. While previous work performed modulo scheduling over a single iteration of loop traces, in this work we briefly evaluate the list scheduling of a number of loop basic block repetitions, in practice unrolling the loop, and report on how the speedup evolves when increasing both the number of repetitions and resource configurations.

We identify three main takeaways. Firstly, high repetition values are not desirable if the number of available resources is not enough to meet the demand, which causes operations to wait in a ready state before being scheduled. For an accelerator with up to 4 resources of each type, which is feasible to implement, it is more advantageous to use 4 repetitions instead of 5.

Secondly, we note that accelerators should focus on providing as many ALUs as possible, as arithmetic operations usually far exceed the number of memory accesses, and being able to schedule these as soon as possible leads to the biggest speedup gains.

Finally, we have also looked into how close our schedules were to the optimal scenario in an accelerator without resource constraints: only 29% of our benchmarks failed to achieve that speedup using the highest resource configuration we allowed, (8, 8), which shows that our approach in terms of resource constraints provides a good compromise when it comes to obtaining a high speedup while keeping an efficient hardware design. We have also considered the impact that memory dependencies have on scheduling, as our approach assumed that no memory access alias. While the most pessimistic approach can still produce significant speedups, we consider ongoing work to develop ways to disambiguate some memory accesses during scheduling.

It is also worth noticing that this study is part of a larger project that aims to accelerate, in real time, different kinds of binary segments, such as Basic Blocks, Superblocks and Megablocks, detected on traces with different instruction sets, such as MicroBlaze, RISC-V and ARM. We see this study as a way to explore, at an early stage, the impact of one of the possible transformations we may perform over instruction traces, in order to achieve better speedups. Other types of analysis, such as real-time dependence tracking and detecting memory access patterns, are considered ongoing work.

ACKNOWLEDGMENT

This work was supported by the PEPCC project, "PTDC/EEI-HAC/30848/2017," financed by Fundação para a Ciência e Tecnologia (FCT).

REFERENCES

[1] Paulino, N., Ferreira, J. & Cardoso, J. Improving Performance and Energy Consumption in Embedded Systems via Binary Acceleration: A Survey. *ACM Comput. Surv.*. **53** (2020,2), https://doi.org/10.1145/3369764
[2] Paek, J., Choi, K. & Lee, J. Binary Acceleration Using Coarse-Grained Reconfigurable Architecture. *SIGARCH Comput. Archit. News*. **38**, 33-39 (2011,1), https://doi.org/10.1145/1926367.1926374
[3] Rokicki, S., Rohou, E. & Derrien, S. Hardware-accelerated dynamic binary translation. *Design, Automation & Test In Europe Conference Exhibition (DATE), 2017*. pp. 1062-1067 (2017)
[4] Paulino, N., Ferreira, J. & Cardoso, J. Generation of Customized Accelerators for Loop Pipelining of Binary Instruction Traces. *IEEE Transactions On Very Large Scale Integration (VLSI) Systems*. **25**, 21-34 (2017)
[5] Paulino, N., Bispo, J., Ferreira, J. & Cardoso, J. A Binary Translation Framework for Automated Hardware Generation. *IEEE Micro*. **41**, 15-23 (2021)
[6] Cooper, K. & Torczon, L. Chapter 12 - Instruction Scheduling. *Engineering A Compiler (Second Edition)*. pp. 639-677 (2012), https://www.sciencedirect.com/science/article/pii/B9780120884780000128
[7] Ansaloni, G., Bonzini, P. & Pozzi, L. Design and Architectural Exploration of Expression-Grained Reconfigurable Arrays. *2008 Symposium On Application Specific Processors*. pp. 26-33 (2008)
[8] Lee, H., Moghaddam, M., Suh, D. & Egger, B. Improving Energy Efficiency of Coarse-Grain Reconfigurable Arrays Through Modulo Schedule Compression/Decompression. *ACM Trans. Archit. Code Optim.*. **15** (2018,3), https://doi.org/10.1145/3162018
[9] Liu, C. & So, H. Automatic Soft CGRA Overlay Customization for High-Productivity Nested Loop Acceleration on FPGAs. *2015 IEEE 23rd Annual International Symposium On Field-Programmable Custom Computing Machines*. pp. 101-101 (2015)
[10] Guha, A., Vedula, N. & Shriraman, A. Deepframe: A Profile-Driven Compiler for Spatial Hardware Accelerators. *2019 28th International Conference On Parallel Architectures And Compilation Techniques (PACT)*. pp. 68-81 (2019)
[11] Paulino, N., Ferreira, J. & Cardoso, J. Trace-Based Reconfigurable Acceleration with Data Cache and External Memory Support. *Proceedings Of The 2014 IEEE International Symposium On Parallel And Distributed Processing With Applications*. pp. 158-165 (2014), https://doi.org/10.1109/ISPA.2014.29