Fast Shared On-Chip Memory Architecture for Efficient Hybrid Computing with CGRAs

Jongeun Lee, Yeonghun Jeong and Sungsok Seo
School of ECE
Ulsan National Institute of Science and Technology, Ulsan, Korea
jlee@unist.ac.kr

***Abstract*—While Coarse-Grained Reconfigurable Architectures (CGRAs) are very efficient at handling regular, compute-intensive loops, their weakness at control-intensive processing and the need for frequent reconfiguration require another processor, for which usually a main processor is used. To minimize the overhead arising in such collaborative execution, we integrate a dedicated sequential processor (SP) with a reconfigurable array (RA), where the crucial problem is how to share the memory between SP and RA while keeping the SP's memory access latency very short. We present a detailed architecture, control, and program example of our approach, focusing on our optimized on-chip shared memory organization between SP and RA. Our preliminary results demonstrate that our optimized memory architecture is very effective in reducing kernel execution times (23.5% compared to a more straightforward alternative), and our approach can reduce the RA control overhead and other sequential code execution time in kernels significantly, resulting in up to 23.1% reduction in kernel execution time, compared to the conventional system using the main processor for sequential code execution.**

I. Introduction

While Coarse-Grained Reconfigurable Architectures (CGRAs) are very good at regular, compute-intensive loops, they may not be able to deal with other variants very efficiently, such as those that contain irregular memory accesses, function calls (including basic math functions like exponential and trigonometric functions), and deeply nested conditionals or loops. Even perfectly nested loops cannot be directly executed by CGRAs, leading to a hybrid mode of execution [1], where the inner-most loops are executed by a CGRA while the outer loops are carried out in software by the main processor. Borrowing the power of main processor for every small unsupported operation of a CGRA can make programming and optimization cumbersome. More critically, the speedup by the CGRA can be greatly offset by the control overhead on the main processor. Our study, agreeing with [2], shows that the control overhead of a CGRA can be as high as 40% of kernel execution time. It is such limitations of CGRAs that motivate our work in this paper.

In this work we explore a different way to reduce the overhead—by adding a small dedicated processor for a CGRA. A dedicated processor can be placed right next to the CGRA, without going through multiple interconnections unlike a main processor in today's System-On-Chip (SOC). Moreover, a dedicated processor can be customized for the CGRA accelerator, including special instructions and custom memory interface. Further, by delegating the management of the CGRA, the main processor can work on more important work with less interruption, or be put into low power mode. We call this approach *SPIRA* (Sequential Processor Integrated Reconfigurable Array), where Sequential Processor (SP) and Reconfigurable Array (RA) refer to the dedicated processor and the CGRA, respectively.

One pitfall of adding a processor is the (newly added) communication overhead between SP and RA. Therefore we consider it essential for them to have a shared memory, preferably all of the RA's local memory being accessible to the SP, and the SP can use the RA's local memory as its primary data memory to minimize the hardware cost. However, this creates a new problem of increasing the memory access latency of the SP. The local memory architecture of the RA is typically optimized for throughput and bandwidth, and therefore can deal with latency, which is mostly due to a crossbar switch between multiple PEs and multi-bank memories [3]. If the SP has to share the same memory interface (e.g., by extending the crossbar switch with a port for the SP), the memory access latency could be increased so much as to negate the potential advantage of adding a dedicated processor.

In this paper we present a novel memory architecture for SPIRA, which enables the sharing of the data memory between the SP and the RA, while simultaneously providing for the SP very fast access latency for RA's memory as well as RA's registers. This is made possible by our novel memory interface and a set of hardware controllers ensuring that SP and RA are never simultaneously active. This *exclusive execution* semantics between SP and RA can also help reduce the synchronization delay as well as conserve the energy.

Our experimental results from detailed system-level simulation using applications from computer vision and multimedia benchmarks demonstrate that our novel memory architecture can reduce kernel execution times by up to 38.9% (23.5% on average) compared to simply extending the crossbar switch. Also despite the slower speed and the simpler architecture of the SP over the MP, our SPIRA approach can reduce the RA control overhead and other sequential execution time in kernels by 25.3% and 39.5% on average, respectively, resulting in significant reduction in total kernel execution time—up to 23.1% (12.7% on average)—compared to the conventional system using the main processor for sequential code execution.

978-3-9815370-0-0/DATE13/©2013 EDAA

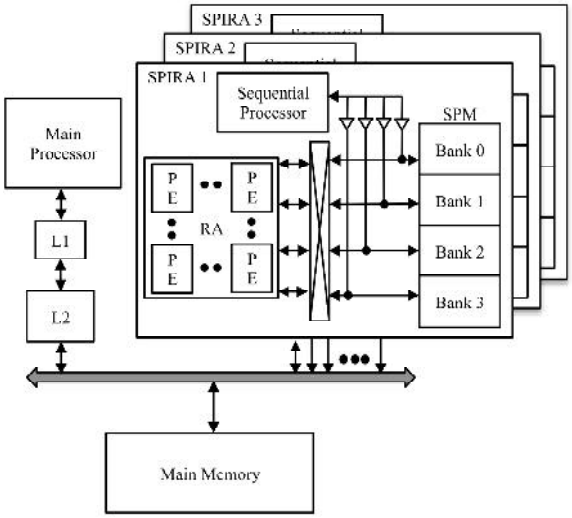

Fig. 1. System architecture employing multiple SPIRA accelerators.

II. Related Work

Coupling a sequential processor to a reconfigurable architecture is not new. Many CGRAs [4]–[6], as well as other types of coprocessors, typically assume a main processor. However, the main processor of those coprocessors is very different from our SP, since the main processor is not typically customized for a coprocessor, and has its own memory hierarchy including data caches and even address translation, none of which is shared with the RA. Our SP processor uses a scratchpad memory as its only data memory (no data cache or address translation), which is fully shared with the RA.

In particular, the ADRES reconfigurable processor [6] supports the idea of tightly-coupled coprocessor, which has a design-time option of making part of the CGRA (typically 4 PEs) reconfigurable as a VLIW processor. However, the VLIW is very different from our SP from the memory architecture's perspective, and more similar to the main processor mentioned above. Moreover, unlike the SP, the VLIW "mode" cannot be used to reconfigure the RA.

Some recent FPGAs include a hard or soft processor in the fabric. A hard processor is a physical processor core built into the FPGA silicon, such as PowerPC embedded in Xilinx Virtex-4 and ARM922T in Altera Excalibur, and therefore has little flexibility. On the other hand, soft processors are built using the FPGA's general-purpose logic, and therefore can be easily customized [7], [8] for the target application. Our SP processor is designed to make runtime reconfiguration faster, which however is not the primary focus of soft processor due to the much slower reconfiguration with current FPGAs.

When a loop is accelerated using a reconfigurable architecture, the portion of the sequential code becomes larger than in the original sequential execution. Especially when there are one or more outer loops, the iteration control in the outer loops may be even higher than that of the innermost loop in terms of the execution time [2]. The overhead can be very high especially if there are one or more outer loops. To remedy this problem, [1] proposes a hardware solution for 2-deep nested loops only, whereas our approach is more versatile and can be applied to any arbitrary loop level. [9] applies polytope-based loop parallelization to CGRAs, which can handle arbitrary loop levels; however, it is not known how to automatically generate such implementations for arbitrary loops including imperfect ones.

III. SPIRA Approach

A. Hardware Architecture

We base our RA architecture on ADRES [3], which is a 4x4 array of PEs coupled with a full crossbar switch and a multi-banked scratchpad memory. The SP is a single-issue, in-order, pipelined processor whose primary role is to control the RA and, when profitable, to complement the RA by performing control-intensive code or other operations that are not supported by the RA. As such, the SP does not have cache or address translation for data memory. Making the SP simple (viz., single-issue, in-order, no cache or address translation) helps not only minimize its power consumption, but also optimize the memory access latency, which is critical for the SP's main job.

1) Expanding the Crossbar: A straightforward way to create a shared memory between SP and RA is to connect the SP's address and data buses to an input port of the crossbar switch located between the RA and the scratchpad memory (SPM). However, this approach requires adding a new port, and can increase the SP-to-SPM path latency significantly. In the ADRES architecture, the crossbar can add to the latency by 4 to 8 cycles depending on how well bank conflict is avoided [3]. Moreover, this can affect not only the SPM access latency but also the latency to the RA's registers.

2) Proposed Memory Architecture: Fig. 1 illustrates our proposed architecture, where the SP and the RA are connected to the SPM through one set of address and data buses. These buses are primitive wires without arbitration or address decoder, as they were originally one-to-one connections. We now need arbitration and address decoder because of the sharing of the same buses with the SP. Thus we add them only on the SP's data paths as illustrated in the figure. Address decoder's output, when the SP is running, controls which SPM bank should be enabled, and all the banks are fed with the same address lines from the SP. Arbitration is done by running the SP and the RA exclusively, as explained in Section III-B.

B. Software and Control

Initially the MP Code runs on the main processor. Soon the main processor invokes SPIRA by writing to one of its memory-mapped registers. The written value is the start address for the SP, which the SPIRA controller receives and sends it through a dedicated set of wires to the SP, along with a wakeup signal. At the same time, the controller wakes up the SP, which starts executing the SP Code. In addition to executing sequential code outside of the kernel, the SP performs a series of store instructions to initialize certain

registers of the RA and to set up the pipeline, which is followed by the invocation of the RA.

C. Memory Access Latency

Minimizing the memory latency for SP is critical, since the SP performs control-intensive computations including controlling the RA. While the SP can access the SPM, RA registers, or even external memory or devices, the most important paths are those reaching the SPM and RA registers.

The SP-to-SPM path latency is determined by the SPM latency plus the mux or decoder delay. Both the mux and the decoder have extremely small delay, because their sizes are very small, and their control input, coming from the ECU, is kept constant except when the SP or RA changes the power state. Using Cacti 6.5 [10], we estimate that the SPM of 256 KB on a 65 nm techniques has 3.52 ns latency and 1.58 ns cycle time. This can give a fast 2-cycle load latency for SP running at 520 MHz. Similarly the SP-to-RA-register latency is dominated by the RA register access time, which we assume to be 2 SP cycles.

The RA has one memory access path, which is from memory-accessing PEs to SPM banks via the crossbar switch. This path is almost identical to that of the original RA, and so should be the latency.

IV. Experiments

A. Experimental Setup

To evaluate the effectiveness of our SPIRA architecture we use applications from San Diego Vision Benchmark Suite (SD-VBS) [11], MiBench [12], and MediaBench [13]. We select 1∼3 kernels from each application that are the most important in terms of execution time, and map them to SPIRA. Only the inner-most loops are mapped to the RA using a modulo scheduling algorithm [14] while outer loops are mapped to the SP. For comparison we also map kernels to a system with the RA accelerator only; in this case, the inner-most loop is still mapped to the RA but the outer loops are mapped to the main processor.

We extend the SimpleScalar simulator [15] integrated with DRAMsim 1.2 [16] to model the target system, which includes a main processor, a SPIRA accelerator, a system interconnect, a DMA engine, and a main memory. The main processor is modeled after ARM Cortex A8 processor (dual-issue, in-order) running at 720 MHz [17] with separate level-1 caches, each with 16 KB, and a unified level-2 cache of 256 KB. The SP processor is modeled after ARM11 processor (single-issue, in-order) [18] running at 520 MHz with no cache or address translation. The RA has an array of 4x4 PEs (with the exception of MPEG2 kernels which we were able to map only to a 6x6 PE array with 6 SPM banks), including four load-store PEs that are connected to four 256 KB SPM banks via a crossbar switch. The RA runs at 520 MHz and supports bidirectional mesh-plus-diagonal PE-to-PE interconnection. The system interconnect is 32-bit wide, runs at 166 MHz, and supports multiple outstanding transactions with pipelining similar to the ARM AXI protocol. The DMA engine is 32-bit wide and multi-threaded. The main memory is 32-bit wide DDR-333 SDRAM.

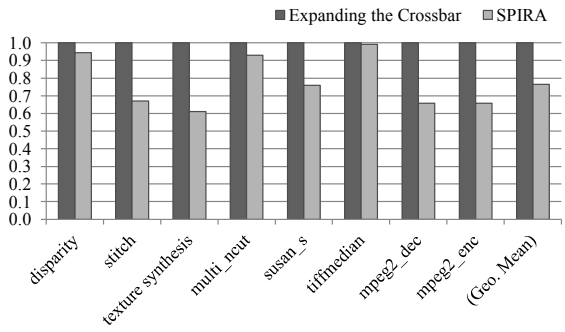

Fig. 2. Our proposed memory architecture vs. expanding the crossbar (kernel execution time, normalized).

B. Comparison against Expanding the Crossbar

While our SPIRA architecture connects the SP directly to the SPM without going through the crossbar switch, a more straightforward solution would be to simply expand the crossbar. In this experiment we compare kernel execution times of using our proposed memory architecture vs. the alternative (i.e., expanding the crossbar).

First we note that there are side effects to expanding the crossbar, that the latency of the crossbar itself is increased (because instead of 4-to-1 muxes, 8-to-1 muxes must be used now), and that the crossbar's latency must also be added to the SP's memory access latency. We ignore the former and only evaluate the latter, which seems much more critical. Considering the differences in the technology, frequency, and memory size, we estimate that the latency is increased to 6 cycles from 5 cycles reported in [3]. Fig. 2 shows our simulation results. As expected, our SPIRA architecture can reduce kernel runtime consistently in all the applications. Overall, by using our SPIRA architecture kernel runtime is reduced by up to 38.9% (in *texture synthesis*) and on average 23.5%, as compared to the alternative. The SP runtime alone (outer loops and RA control) is reduced by 40.0% on average, and 42.8% at most (not shown in the graph).

C. Comparison against Using Main Processor

In the next two sets of experiments we compare our SPIRA approach with more conventional system architecture. Without SPIRA, one has to use the main processor to complement the RA. Thus our first comparison is against using the main processor instead of the SP. Note that though SPIRA requires more resources due to the SP and the small controllers, the SPIRA case doesn't utilize the main processor at all for kernels; therefore, it can be seen as a fair comparison as far as the kernel performance is concerned.

Fig. 3 compares kernel execution times normalized to that of the MP+RA case. The first four applications are from SD-VBS, the next two from MiBench, and the last two from MediaBench. For applications with multiple selected kernels, we report the sum of kernel execution times, weighted by their

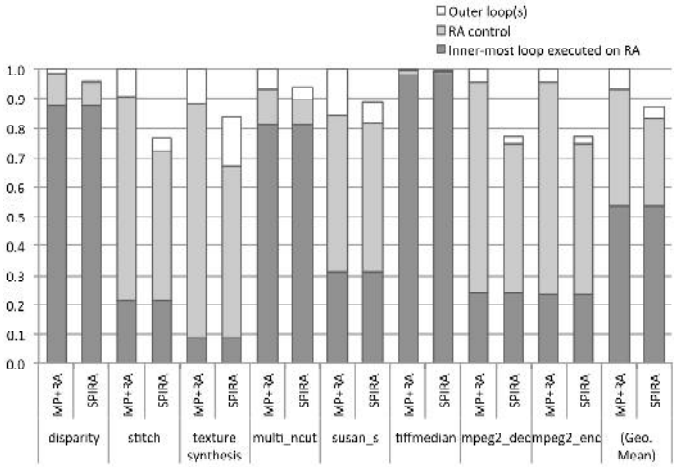

Fig. 3. SPIRA vs. MP+RA (kernel execution time, normalized).

execution frequencies. We assume that DMA for configuration and array data is finished before kernel execution.

In the graph the execution time of each application is broken down into three parts: outer loop(s), RA control for the inner-most loop, and the inner-most loop. Since the inner-most loop is executed on the RA, this part is the same regardless of the architecture. The other parts vary, but are mostly reduced in the SPIRA's case.

First we observe that the RA control part is reduced consistently in the SPIRA's case. This is not surprising, considering that the SP is tightly-coupled with the RA, and therefore has an edge over the MP, in quickly accessing RA registers as well as synchronizing with it. However it is unexpected that the outer loop execution time is also reduced in the SPIRA's case—often significantly—with one exception of *texture synthesis*, even though the MP is more powerful than the SP (viz., 38% faster clock frequency and dual-issue). This can be attributed to the following factors: most of outer loop codes are simple; the dual-issue feature of the MP couldn't be utilized very effectively due to control statements (i.e., branches); and the SP has a fast level-1 memory access time—even faster than the MP with the clock speed difference taken into account. Lastly the degree of kernel speedup by our SPIRA approach is strongly correlated with how much portion of the kernel execution time is spent on non-RA execution issues. In summary our SPIRA architecture can reduce the outer loop execution time and the RA control overhead by 39.5% and 25.3%, respectively, resulting in 12.7% reduction in the kernel execution time, all on average.

V. Conclusion

We presented the SPIRA approach and its fast, shared memory architecture for SP, which can reduce the control overhead in hybrid computing involving main processor and CGRA processors. We find that optimizing the memory architecture is crucial to realize the potential advantage of the SPIRA approach. Our experiments using applications from computer vision and multimedia benchmarks indicate that combining all three processors—main processor, SP, and RA—can result in the best performance in most situations, than using either pair of the processors only. Our preliminary results demonstrate that not only can our proposed memory architecture significantly reduce the kernel runtimes compared to a more straightforward alternative, our proposed technique has significant benefit in terms of performance over the conventional system employing the main processor for sequential code execution. While we expect the cost of adding a SP and the small controllers to a CGRA to be small compared to that of a CGRA due to the SP's simple architecture (no data cache or address translation), accurate evaluation of it remains for future work.

Acknowledgment

This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology, under grant 2010-0011534.

References

[1] Y. Kim *et al.*, "Improving performance of nested loops on reconfigurable array processors," *ACM Transactions on Architecture and Code Optimization*, 2012.

[2] N. Kapre *et al.*, "VLIW-SCORE: Beyond C for sequential control of spice FPGA acceleration," in *IEEE International Conference on Field-Programmable Technology (FPT 2011)*, 2011.

[3] B. Bougard *et al.*, "A coarse-grained array accelerator for software-defined radio baseband processing," *IEEE Micro*, vol. 28, no. 4, pp. 41–50, 2008.

[4] J. Hauser and J. Wawrzynek, "Garp: a mips processor with a reconfigurable coprocessor," in *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, apr 1997.

[5] H. Singh *et al.*, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, 2000.

[6] B. Mei *et al.*, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," *Lecture Notes in Computer Science*, vol. 2778, pp. 61–70, 2003.

[7] P. Yiannacouras *et al.*, "The microarchitecture of FPGA-based soft processors," in *Proc. CASES*. ACM, 2005, pp. 202–212.

[8] C. H. Chou *et al.*, "Vegas: soft vector processor with scratchpad memory," in *International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. ACM, 2011.

[9] F. Hannig *et al.*, "Mapping of regular nested loop programs to coarse-grained reconfigurable arrays - constraints and methodology," in *Proc. of IPDPS*, april 2004, p. 148.

[10] N. Muralimanohar. *et al.*, "Cacti 6.0: A tool to understand large caches," 2009. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.147.3834

[11] S. Venkata *et al.*, "Sd-vbs: The san diego vision benchmark suite," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, oct. 2009, pp. 55 –64.

[12] M. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, dec. 2001, pp. 3 – 14.

[13] C. Lee *et al.*, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO-30*, Washington, DC, USA, 1997, pp. 330–335.

[14] H. Park *et al.*, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proc. PACT*. New York, NY, USA: ACM, 2008, pp. 166–176.

[15] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," *Computer*, vol. 35, 2002.

[16] D. Wang *et al.*, "Dramsim: a memory system simulator," *SIGARCH Comput. Archit. News*, vol. 33, pp. 100–107, November 2005.

[17] *OMAP3530/25 Applications Processor*, Texas Instruments.

[18] *ARM1176JZF-S Technical Reference Manual*, ARM.