Journal of Systems Architecture 75 (2017) 107–119

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Increasing the efficiency and feasibility of configurable computing units

Anita Tino[*], Kaamran Raahemifar

Department of Electrical and Computer Engineering, Ryerson University, 350 Victoria St., Toronto, ON, Canada

ARTICLE INFO

Article history:
Received 13 June 2016
Revised 21 November 2016
Accepted 12 January 2017
Available online 13 January 2017

Keywords:
General purpose processors
Configurable datapaths
Task execution
Microarchitecture
Computer architecture

ABSTRACT

Multicore processors are customary within current generation computing systems. The overall concept of general purpose processing however remains a challenge as architects must provide increased performance for each advancing generation without solely relying on transistor scaling and additional cache levels. Although architects have steered towards heterogeneity to increase the performance and efficiency for a variety of workloads, the fundamental issue of how a single core's architecture may be improved and applied to the multiprocessor domain remains. This work builds upon the concept of Configurable Computing Units (CCU) - a nuanced approach to processor architectures and microarchitectures, employing reconfigurable datapaths and task-based execution. This work improves upon the efficiency of CCUs by applying various new design techniques including branch prediction, variable configuration, an OpenMP programming model, and Berkeley Dwarf testing. Experimental results using Gem5 demonstrate that a single CCU core can achieve dual-core performance, with a 1.29x decrease in area overhead and 55% of the power consumption required by a conventional CPU.

1. Introduction

From the dawn of the first in-order microprocessor, every advancing computing generation has continued to provide its users with increased performance and enhanced design features. The key challenge within the current heterogeneous multicore generation is to exploit both parallel and sequential thread performance with efficiency. For instance, invoking a system with numerous simple cores delivers high thread-level parallelism (TLP) with energy and area efficiency, whereas employing heterogeneous cores may increase TLP and data-level parallelism (DLP) for computationally intensive application phases at the additional cost of area and power consumption. Although an abundance of TLP and DLP may be extracted using additional on-chip resources, computing architectures are still fundamentally limited by the overall Instruction Level Parallelism (ILP) and DLP a core may provide [1,2]. Accordingly, the problem of how a single core's organization and design may be improved and applied to the multiprocessor domain remains [3,4].

A common response to increasing general purpose processor performance has been to invoke dataflow-like execution hardware within a datapath. Many unconventional computing models have addressed such issues by redesigning the processor completely [5–8]. These dataflow-like processor models are able to increase single core ILP and DLP, while eliminating unnecessary broadcasts to improve energy efficiency. In order to support dataflow-like execution however, these processors require custom compilers and/or languages which lead to several software compatibility issues. Majority of these processors also possess massive and distributed grid-like structures causing dependency and communication issues, while proving problematic for memory accesses which lie on the outer bounds of a processor.

An FPGA's bit-level configurable approach requires very fine-grained application customization leading to significant increases in design effort and long compilation times, proving problematic for general purpose computing. Coarse-Grained Reconfigurable Architectures (CGRA) have been proposed to mitigate FPGA effects, raising configurability to the word-level and reducing the amount of configuration information necessary for dynamic customization in computing systems. These architectures act as accelerators and/or coprocessors to monolithic CPUs, encompassing several internal computing elements that are interconnected for dataflow-like execution [9,10]. CGRAs increase performance however at the expense of ISA, compiler, and microarchitecture modifications to support custom instructions which redirect applicable code phases to the backend CGRA unit(s). Certain CGRAs also require custom programming languages, design flows, and OS support for compatibility with current computing systems [9,11,12].

[*] Corresponding author.
E-mail addresses: a2tino@ee.ryerson.ca (A. Tino), kraahemi@ee.ryerson.ca (K. Raahemifar).

http://dx.doi.org/10.1016/j.sysarc.2017.01.006
1383-7621/© 2017 Elsevier B.V. All rights reserved.

To address general purpose single-core issues while maintaining computing compatibility, previous work has presented the concept of Configurable Computing Units (CCU) [13,14,22]. As opposed to using CGRAs with power hungry monolithic cores and additional reconfigurable support, CCUs redesign the processor and datapath completely by invoking underlying reconfigurable hardware for enhanced single-thread performance, ILP, DLP, and energy efficiency. Specifically, CCUs are configurable processors that execute tasks which are generated through *logical* and *physical* compilation. Unlike previous work in dataflow based processors, CCUs employ a *logical* compiler to maintain compatibility with current software, programming languages, and compilers, while using a *physical* compiler (PhysC) to exploit information of the underlying hardware architecture and overcome restricted dataflow. This technique allows CCUs to adapt to a variety of workloads in an efficient manner, acting as a middle layer to maintain compatibility and capture the needs of the underlying hardware. This work builds upon the concepts presented in [14] by invoking:

- OpenMP support with a Gem5 simulator implementation
- Branch prediction techniques
- Variable configuration methods for configuration overhead mitigation
- Revisions and optimizations to PhysC
- Berkeley Dwarf benchmark testing to determine the advantages and limitations of CCU architectures
- Effects of load/store unit scalability on the memory system (and other scalability analyses)
- RTL modeling for full processor prototypes

The remainder of this paper is organized as follows: Section 2 provides background information on CCU processors. Section 3 expands upon the concepts of logical and physical compilation. Section 4 discusses details and revisions to the CCU architecture. Section 5 presents and discusses the experimental testing and results obtained using the new model, along with the effects of various CCU configurations and optimizations. Previous work is overviewed and compared in Section 6, with a conclusion provided in Section 7.

2. Background

This section provides a brief overview of basic CCU terminologies in Section 2.1, and overviews the proposed processor's functionality in Section's 2.2 and 2.3 using a design flow and execution example, respectively.

2.1. Definitions

A CCU is a general purpose processing core consisting of variable sized engines. Each engine comprises of unique functional units (FUs) that are connected through a registerSwitch (rS) interconnect as shown in Fig. 1. The interconnect consists of configurable rS units which provide *distributed storage* and single-cycle multi-hop data communication between dependent instructions. These characteristics allow the rS interconnect to avoid constant access to centralized (or outer grid-bound) register files, bypass networks, and tile-based hotspots, while preventing unnecessary broadcasts. The engines configure to a general purpose application's communication patterns using the programmable rS interconnect and configuration data generated by the PhysC. Therefore, each engine is able to temporally configure itself on every clock cycle during execution to support various data transfers and storages as required by the application's task (as opposed to replicating issue, dispatch, bypass and writeback stages similar to previous works [5–8]). A banked memory setup holding configuration data

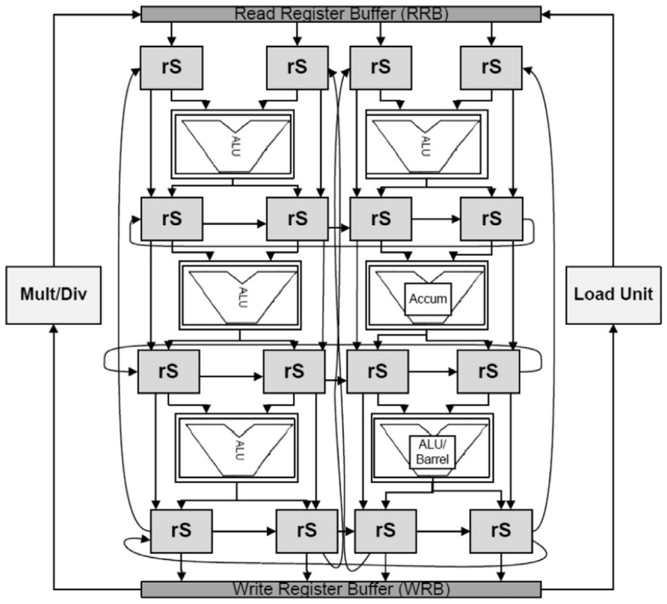

Fig. 1. Six functional unit (Single) engine CCU architecture.

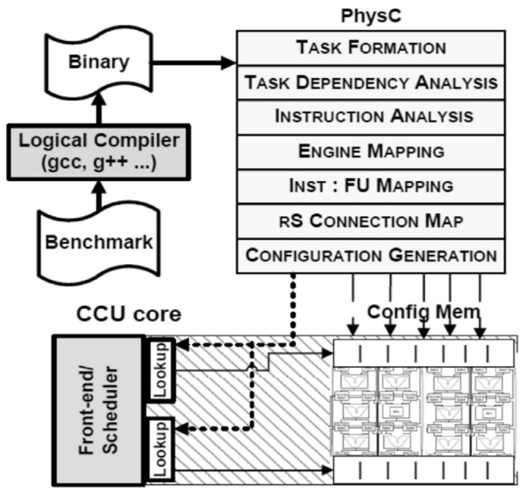

Fig. 2. CCU design flow.

is also used in the backend to reduce configuration times and allow for reconfigurability in general purpose processors.

2.2. General CCU design flow

The overall CCU design flow is presented in Fig. 2. As typical of general purpose processors, the first step requires that a benchmark be input to the *logical* compiler (i.e. any standard compiler) which generates a binary. The binary is then sent to the PhysC for further processing. Using the binary and information of the underlying architecture, the PhysC performs **1)** task formation, **2)** task and instruction analysis to obtain statistical data and eliminate data hazards, which is then used for the **3)** extraction of producer-consumer dependencies, **4)** selection of the most suitable engine for task execution, and finally the **5)** generation of configuration data to be used by CCU engines for execution. The generated configuration data is then stored to its respective engine's banked configuration memory (Fig. 2). Similarly, the addresses used to store the configuration data are saved to lookup tables in the CCU backend, based on task IDs and respective execution engines.