Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Hardware design methodology using lightweight dataflow and its integration with low power techniques

Tiziana Fanni[a,*], Lin Li[b], Timo Viitanen[c], Carlo Sau[a], Renjie Xie[c], Francesca Palumbo[d], Luigi Raffo[a], Heikki Huttunen[c], Jarmo Takala[c], Shuvra S. Bhattacharyya[b,c]

[a] *University of Cagliari, Dept. of Electrical and Electronic Engineering, Italy*
[b] *University of Maryland, ECE Department, College Park, MD 20742, USA*
[c] *Tampere University of Technology, Finland*
[d] *University of Sassari, PolComIng - Information Engineering Unit, Italy*

A R T I C L E I N F O

Article history:
Received 30 December 2016
Revised 3 May 2017
Accepted 12 June 2017
Available online 15 June 2017

Keywords:
Dataflow
Deep neural networks
Digital systems design
Low power design
Signal processing
Clock gating
Globally asynchronous locally synchronous

A B S T R A C T

Dataflow models of computation are capable of providing high-level descriptions for hardware and software components and systems, facilitating efficient processes for system-level design. The modularity and parallelism of dataflow representations make them suitable for key aspects of design exploration and optimization, such as efficient scheduling, task synchronization, memory and power management. The lightweight dataflow (LWDF) programming methodology provides an abstract programming model that supports dataflow-based design of signal processing hardware and software components and systems. Due to its formulation in terms of abstract application programming interfaces, the LWDF methodology can be integrated with a wide variety of simulation- and implementation-oriented languages, and can be targeted across different platforms, which allows engineers to integrate dataflow modeling approaches relatively easily into existing design processes. Previous work on LWDF techniques has emphasized their application to DSP software implementation (e.g., through integration with C and CUDA). In this paper, we efficiently integrate the LWDF methodology with hardware description languages (HDLs), and we apply this HDL-integrated form of the methodology to develop efficient methods for low power DSP hardware implementation. The effectiveness of the proposed LWDF-based hardware design methodology is demonstrated through a case study of a deep neural network application for vehicle classification.

1. Introduction

Dataflow models of computation are highly suitable to describing the functionality of digital signal processing (DSP) applications. In dataflow graphs, nodes represent computational components, called *actors*, and edges model channels for point-to-point communication between actors. An actor can be executed whenever it has sufficient data on its input ports and sufficient empty space on its output ports.

Dataflow models present an intrinsic modularity that facilitates the composition and portability of computational components (e.g., see [1,2]), being effective in terms of retargetability of design processes across different platforms (e.g., see [3]). Furthermore, dataflow models facilitate exploration of optimization techniques to achieve efficient implementations of DSP systems (e.g., see [4–6]).

Lightweight dataflow (*LWDF*) is a programming methodology that allows designers to systematically integrate and experiment with dataflow modeling approaches in the context of existing design processes [1]. LWDF is “lightweight” in the sense that the programming model is designed to be minimally intrusive on existing design methodologies and processes, and requires minimal dependence on specialized tools or libraries.

Previous work on LWDF techniques has emphasized their application to DSP software implementation (e.g., through integration with C and CUDA, as presented in [7,8]). In [9], to the best of our knowledge, we presented the first study that deeply integrates LWDF techniques with hardware description language (HDL) programming, and that provides a complex application study involving LWDF-based digital hardware design and optimization.

There is a wide variety of model-based design methodologies and corresponding tools for digital hardware implementation. For

* Corresponding author.
E-mail addresses: tiziana.fanni@diee.unica.it (T. Fanni), lli12311@umd.edu (L. Li), timo.2.viitanen@tut.fi (T. Viitanen), carlo.sau@diee.unica.it (C. Sau), renjie.xie@tut.fi (R. Xie), fpalumbo@uniss.it (F. Palumbo), raffo@unica.it (L. Raffo), heikki.huttunen@tut.fi (H. Huttunen), jarmo.takala@tut.fi (J. Takala), ssb@umd.edu (S.S. Bhattacharyya).

http://dx.doi.org/10.1016/j.sysarc.2017.06.003
1383-7621/© 2017 Elsevier B.V. All rights reserved.

example, the Compaan and Laura tools for HDL code synthesis from Kahn process networks (KPNs) are presented in [10]. The CAL programming language and the Orcc toolset provide a novel environment for implementing dataflow programs in hardware [11,12]. The CAPH language and framework represent another recent effort to generate HDL from a dataflow language [13]. In [14], methodologies for hardware synthesis of pipelined dataflow graph structures are developed to enable high-level, hardware-oriented dataflow graph optimization. Nezan et al. [15] present an integrated design flow and corresponding tools that automatically optimize dataflow specifications to generate HDL designs. The Multi-Dataflow Composer (MDC) tool, a framework for the automatic creation of multi-functional reconfigurable platforms, performs a complete design space exploration, evaluating the trade-off among resource usage, power consumption and operating frequency [16].

Some relevant works have explored the deployment of multiple clock domains and clock gating techniques in conjunction with dataflow-based digital hardware implementations: Brunet et al. propose a design methodology to partition streaming applications onto a multi-clock-domain architecture [17], and Xronos, a high-level synthesis tool for FPGA platforms, adopts a coarse-grained clock gating strategy [18]. The MDC tool has the capabilities of identifying disjointed logic regions, and automatically applying coarse grain clock and power gating techniques to these regions [19,20].

Generally speaking, these methodologies and tools are limited by the language used to describe the adopted dataflow description or by the generated HDL, which can be target dependent. Furthermore, these tools support the user-friendly application of existing design optimization techniques, rather than the rapid prototyping of new techniques. Automatically targeting a new language, platform or design optimization technique can require significant effort in development and maintenance of graph analysis and code generation functionality. Such effort can sometimes be justified for models and design approaches that are relatively mature. However, for experimental methods or methods that are highly specialized to a particular application, such effort is costly.

LWDF helps to address this gap by providing a compact set of APIs that can be used to incorporate advanced dataflow techniques in a manner that does not require development and maintenance of automation tools. Rather than being focused on automation, LWDF is designed to help the designer architect an efficient dataflow-based implementation and iteratively experiment with it. This capability has been highly useful in the work that we are reporting on in this paper, as it has allowed us to rapidly incorporate and experiment with advanced power optimization techniques in the framework of a systematic dataflow-based design methodology.

At the same time, because the LWDF APIs are based on formal dataflow principles, LWDF-based implementations can be well-suited as a target for automated synthesis and code generation tools. For example, LWDF APIs for CUDA and C have been targeted in the DIF-GPU tool for automated synthesis of hybrid CPU/GPU implementations [21]. Indeed, development of automation support for the models and methods introduced in this paper is a useful direction for future work.

In summary, the work in this paper emphasizes the application of DSP-oriented dataflow methods in terms of compact and retargetable APIs, and also emphasizes the rigorous integration of power-management within the proposed APIs. While the design techniques developed in this paper are demonstrated concretely in the context of the Verilog HDL, their formulation in terms of hardware extensions to the abstract LWDF methodology makes them readily retargetable to other HDLs, such as VHDL and SystemC.

A preliminary version of this paper has been presented in [9]. This paper goes beyond the contributions presented in [9] by optimizing the dataflow edge module (DEM) design, and providing an improved clock gating mechanism that we implemented in the deep neural network (DNN) application, where the clock gating technique is applied not only to actors but also to synchronous first-in first-out (FIFO) channels. Details on these extensions are presented in Section 4, and in Section 5, the simulation results of the four DNN designs are updated (compared to the corresponding results in [9]) based on incorporation of the new DEM design, and the new clock gated application design. Additionally, we have extended the presentation of our DNN case study with details on hardware actor profiling, which provides further insight into the characteristics of alternative low power solutions.

The remainder of this paper is organized as follows. Section 2 provides background on dataflow models of computation. In Section 3, we describe the LWDF-V methodology and its implementation in the lightweight dataflow environment (LIDE). In Section 4, we present the proposed integration of low power techniques with the LWDF-V methodology. In Section 5, we demonstrate a DNN application for vehicle classification that is designed and optimized using this new low power form of LWDF-V.

2. Background

In dataflow models of computation for embedded signal processing, DSP systems are modeled as directed graphs, which are composed of nodes (actors) representing computational functions and edges representing communication channels between actors (e.g., see [4]). Each actor executes as a sequence of discrete units of computation, called *firings*. During each firing the actor, according to the adopted dataflow model, consumes one or more *data tokens* from its input edges, performs the computations associated with the firing, and produces one or more *data tokens* onto its output edges. Edges buffer data in a first-in first-out (FIFO) fashion. Some well-defined amount of data is encapsulated in a *token* as it passes from the output edge of one actor to the input edge of another. The conditions under which an actor can fire are called the *firing rules* of the actor (e.g., see [22]). Alternative forms of dataflow in general differ in the classes of firing rules that actors employ.

2.1. Overview of dataflow models

Enable-invoke dataflow (*EIDF*) is a general dataflow model of computation that supports dynamic dataflow behavior in actors [23]. In EIDF, each actor is divided into a set of *modes*, where each mode, when it executes, has static dataflow rates — i.e., each mode has a fixed *consumption rate* and *production rate* associated with each input and output port, respectively. Dynamic dataflow behavior can be achieved by switching among different modes of the same actor that have different dataflow (production or consumption) rates associated with the same port. The firing rule for a given EIDF actor is dependent on the current mode M of the actor. Intuitively, this mode-dependent firing rule is that there must be sufficient data on the actor input buffers (as determined by the fixed consumption rates associated with M), and sufficient vacant space on the actor output buffers (as determined by the production rates associated with M). For more details on the semantics of EIDF, we refer the reader to [23].

The specification of an EIDF actor includes a method called the *enable method*, which checks whether there is sufficient data available on the actor's input ports and sufficient data available on its output ports to fire the actor in its current mode. The enable method returns a Boolean result that is true-valued when the aforementioned availability conditions are met. Each EIDF actor also has an *invoke* method, which executes the actor operation