Computers and Electrical Engineering 39 (2013) 2145–2160

Contents lists available at ScienceDirect

Computers and Electrical Engineering

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

Hardware implementations of software programs based on hierarchical finite state machine models ☆

Valery Sklyarov *, Iouliia Skliarova

Department of Electronics, Telecommunications and Informatics/IEETA, University of Aveiro, Aveiro 3810-193, Portugal

ARTICLE INFO

Article history:
Received 7 January 2013
Received in revised form 20 July 2013
Accepted 22 July 2013
Available online 19 August 2013

ABSTRACT

Advances in microelectronic devices have dissolved the boundary between software and hardware. Faster hardware circuits that enable significantly greater parallelism to be achieved have encouraged recent research efforts into high-performance computation in electronic systems without the direct use of processing cores. Standard multi-core processors undoubtedly introduce a number of constraints, such as pre-defined operand sizes and instruction sets, and limits on concurrency and parallelism. This paper suggests a way to convert methods and functions that are defined in a general-purpose programming language into hardware implementations. Thus, conventional programming techniques such as function hierarchy, recursion, passing arguments and returning values can be entirely implemented in hardware modules that execute within a hierarchical finite state machine with extended capabilities. The resulting circuits have been found to be faster than their software alternatives and this conclusion is confirmed by numerous experiments in a variety of application areas.

1. Introduction

Nowadays, the development of software and hardware becomes more and more interrelated. The emphasis has significantly shifted from general-purpose to application-specific products in the form of embedded processing modules in various areas such as communications, industrial automation, automotive computers, and home electronics. To support application-specific computations, a number of new engineering solutions and technological innovations have been proposed. There is a tendency to integrate components on a chip that not so long ago were separated and implemented as autonomous ASICs (application-specific integrated circuits) or ASSP (application-specific standard products). A few years ago, individual ASICs/ASSPs were assembled together with the surrounding logic, often implemented in autonomous FPGAs (field-programmable gate arrays); today all these components are coupled within the same micro-chip. For example, the Zynq-7000 [1] extensible processing platform (EPP) incorporates a processing system (PS) that combines the industry-standard ARM dual-core Cortex™-A9 32-bit RISC processor and a number of peripherals such as memory controllers, USB (universal serial bus), Gigabit Ethernet, and UART (universal asynchronous receiver/transmitter). The same micro-chip contains a built-in gate array (programmable logic – PL) from the Artix-7 or Kintex-7 FPGA families that is linked with the PS through the AXI (advanced extensible interface).

EPPs like Zynq [1] can run software that interacts with parallel processing elements (PE) that have been mapped to hardware. The main objective of any PE is to provide greater performance than an equivalent software component with similar functionality that is typically composed of a set of functions in C, or methods in Java. The relative effectiveness

☆ Reviews processed and recommended for publication to Editor-in-Chief by Associate Editor Dr. Rene Cumplido.

* Corresponding author. Tel.: +351 234401539.
E-mail addresses: skl@ua.pt (V. Sklyarov), iouliia@ua.pt (I. Skliarova).

0045-7906/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compeleceng.2013.07.019

(*e.g.* performance) of software modules that have been mapped to hardware PEs needs to be tested, analyzed and compared. Thus, it is important to be able to create the functionality of typical software constructions directly in hardware circuits. This paper addresses the provision of modularity, hierarchy (including recursion), and parallelism in hardware. Modularity and hierarchy are very widely used techniques in general-purpose programming. They are supported by the majority of application-specific development systems for the design of software in single/multi-core autonomous and built-in microcontrollers, mainly originating from specifications in C, and less frequently in Java. In many practical cases, there is a need for hardware accelerators to achieve higher performance by parallelizing the most critical parts of the programs in hardware circuits. Thus, mapping such processor-intensive software fragments to hardware by applying potential parallelism becomes very important.

There are many known methods that allow modularity, hierarchy and parallelism to be realized in hardware and a survey of some of these is presented in [2]. Our approach is based on a hierarchical finite state machine (HFSM) model, which is less constrained than potential alternatives [2], can easily be implemented in hardware, and is very consistent with the corresponding software technique. The model is also supported by known templates that are fully synthesizable [2] in commercial computer-aided design (CAD) systems. The next sections provide additional details of the model, review and compare existing alternatives, and explicitly indicate innovations proposed in the paper through the following contributions:

1. A new HFSM model *with datapath* based on *optimized stacks* built *from memory blocks* (see Sections 4 and 5).
2. A regular technique permitting *the values* of signals (as well as pointers) to be supplied to the invoked HFSM modules and *the returned values* (pointers) to be accessed after terminating the modules, which makes easier to generate hardware from software procedures (see Sections 4 and 5).
3. Concurrent execution of HFSM modules permitting *broad parallelism* to be supported (see Section 6).
4. Examples of practical applications clearly demonstrating benefits that are gained from the proposed extended HFSM capabilities through experiments and comparisons (see Section 7).

The remainder of the paper is organized in seven sections. Section 2 analyzes the related work aimed at the acceleration of software through hardware, but not necessarily based on the technique proposed in this paper. Section 3 describes the HFSM model and demonstrates how this enables software methods and functions to be mapped to electronic circuits. Special attention is paid to the advantages and distinctive features of this model. Section 4 discusses the tradeoffs between software and hardware and presents in detail the proposed novel technique for converting methods and functions from general-purpose languages to hardware. Section 5 suggests a method for the optimization of HFSM memory. Section 6 considers regular solutions for parallel implementations. Section 7 is dedicated to practical applications, experiments, and comparisons. The conclusion is given in Section 8.

2. Related work

Combining the capabilities of software and hardware permits many characteristics of existing applications to be improved. The earliest work on this was done at the University of California in Los Angeles [3]. The idea was to create a computer with a *fixed + variable* structure by augmenting a standard processor with an array of reconfigurable logic, assuming that this logic can be utilized to solve some processor tasks faster and more efficiently. Such a combination of the flexibility of software and the speed of hardware was considered to be a new way to evolve higher performance computing from any general purpose computer. The level of technology in 1959–1960 prevented this idea from being put in practice. Today a very similar technique has been implemented on a chip that combines multi-core processors, embedded blocks, and advanced reconfigurable logic. For example, the Xilinx Zynq xc7z020 EPP [1] permits the implementation and testing of: (1) systems requiring the development of software and invoking the on-chip PS; (2) application-specific hardware in PL using embedded blocks such as DSPs (digital signal processors) and memories, and arbitrary logic composed of FPGA slices; and (3) a *fixed + variable* structure computational system combining the PS and the PL with high-speed data exchange between them.

Let us discuss a potential scenario for interactions between the PS and the PL. The PL implements a set of *modules* that are activated from the PS. A *module* is a hardware circuit that executes a dedicated task. It is an entity in a hardware description language (HDL) such as VHDL that potentially invokes other entities. Data exchange is provided either directly between the PS and the PL, or through a shared window in memory that is accessed from the both the PS and the PL. As soon as the PS needs to initiate accelerated operations, it sends a request to the PL and either transfers data associated with the operations to the PL, or indicates an address and size for the shared memory area in which the data are stored. The PL executes the operations and informs the PS as soon as the operations have been completed. Finally, the results are transferred back, either directly or through the shared memory window. Clearly the PS and the *modules* implemented in the PL can work in parallel. To accelerate software running on the PS, we need to be able to replace time-consuming *software procedures* (*functions* in C) with functionally equivalent *hardware modules* that are faster and to apply parallelism and pipelining. Thus, we need fast mechanisms that enable selected *software functions* to be converted to *hardware modules* that provide the equivalent functionality, and thus execute exactly the same operations as the *software functions*, but faster. This problem has been widely investigated and several common techniques have been applied. A direct approach is to take an entire software program and apply an automatic conversion to hardware. The program can be written in a general-purpose language (GPL) (most often in