Available online at www.sciencedirect.com

SCIENCE DIRECT®

Advances in Engineering Software 37 (2006) 561–582

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

Parallel algorithms development for programmable logic devices

Issam W. Damaj

Electrical and Computer Engineering Department, Hariri Canadian Academy of Sciences and Technology, Meshref P.O. Box: 10 Damour, Chouf 2010, Lebanon

Received 21 July 2005; received in revised form 7 January 2006; accepted 19 January 2006
Available online 22 March 2006

Abstract

Programmable logic devices (*PLDs*) continue to grow in size and currently contain several millions of gates. At the same time, research effort is going into higher-level hardware synthesis methodologies for reconfigurable computing that can exploit *PLD* technology. In this paper, we explore the effectiveness and extend one such formal methodology in the design of massively parallel algorithms. We take a step-wise refinement approach to the development of correct reconfigurable hardware circuits from formal specifications. A functional programming notation is used for specifying algorithms and for reasoning about them. The specifications are realised through the use of a combination of function decomposition strategies, data refinement techniques, and off-the-shelf refinements based upon higher-order functions. The off-the-shelf refinements are inspired by the operators of communicating sequential processes (*CSP*) and map easily to programs in *Handel-C* (a hardware description language). The *Handel-C* descriptions are directly compiled into reconfigurable hardware. The practical realisation of this methodology is evidenced by a case studying the matrix multiplication algorithm as it is relatively simple and well known. In this paper, we obtain several hardware implementations with different performance characteristics by applying different refinements to the algorithm. The developed designs are compiled and tested under *Celoxica's RC-1000* reconfigurable computer with its 2 million gates *Virtex-E FPGA*. Performance analysis and evaluation of these implementations are included.
© 2006 Elsevier Ltd. All rights reserved.

Keywords: Formal models; Gate array; Methodologies; Parallel algorithms

1. Introduction

The rapid progress and advancement in electronic chips technology provides a variety of new implementation options for system engineers. The choice varies between the flexible programs running on a general-purpose processor (*GPP*) and the fixed hardware implementation using an application specific integrated circuit (*ASIC*). Many other implementation options present, for instance, a system with a *RISC* processor and a *DSP* core. Other options include graphics processors and microcontrollers. Specialist processors certainly improve performance over general-purpose ones, but this comes as a quid pro quo for flexibility. Combining the flexibility of *GPPs* and the high performance of *ASICs* leads to the introduction of reconfigurable computing (*RC*) as a new implementation option with a balance between versatility and speed.

Generally, reconfigurable computing is computer processing with highly flexible computing fabrics. The principal difference when compared to using ordinary microprocessors is the ability to make substantial changes to the data path itself in addition to the control flow. In the last decade, there was a renaissance in the area of reconfigurable computing research with many proposed reconfigurable architectures developed both in industry and academia such as, *Matrix*, *Garp*, *RAW*, *DPGA*, *RaPiD*, *PRISM*, *Pleiades*, and *Morphosys* [1]. Such designs were feasible due to the relentless progress of silicon technology that allowed complex designs to be implemented on a single chip.

Field programmable gate arrays (*FPGAs*), nowadays are important components of *RC*-systems, have shown a dramatic increase in their density over the last few years. For example, companies like *Xilinx* [2] and *Altera* [3] have

E-mail address: damajiw@hariricanadian.edu.lb

0965-9978/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2006.01.009

enabled the production of *FPGAs* with several millions of gates, such as in *Virtex-II Pro* and *Stratix-II FPGAs*. The versatility of *FPGAs*, opened up completely new avenues in high-performance computing. These reconfigurable digital electronic hardware circuits can be combined with high-level software and design methodologies to form a powerful paradigm for computing.

The traditional implementation of a function on an *FPGA* is done using logic synthesis based on *VHDL*, Verilog or a similar *HDL* (hardware description language). These discrete event simulation languages are rather different from languages, such as *C*, *C++* or *JAVA*. Many *FPGA* implementation tools are primarily *HDL*-based and not well integrated with high-level software tools. Furthermore, these *HDL*-based *IP* (intellectual property) cores are expensive and they have complex licensing schemes [4]. These obstacles had caused some blockage to the infiltration of *FPGAs* as the main platform solution for hardware engineers. An interesting step towards more success in hardware compilation is to grant a higher-level of abstraction from the point of view of programmer. Designer productivity can be improved and time-to-market can be reduces by making hardware design more like programming in a high-level language. Recently, vendors have initiated the use of high-level languages dependent tools like *Handel-C* [5–8], *Forge* [9], *Nimble* [10,11], *SystemC* [12] and *Viva* [13] (an object-oriented graphical development environment for programming *FPGAs*).

With the availability of powerful high-level tools accompanying the emergence of multi-million *FPGA* chips, more emphasis should be placed on affording an even higher level of abstraction in programming reconfigurable hardware. Building on these research motivations, in the work in hand, we extend and examine a methodology whose main objective is to allow for a higher-level correct synthesis of massively parallel algorithms and to map (compile) them onto reconfigurable hardware. Our main concern is with behavioural refinement, in particular the derivation of parallel algorithms. The presented methodology systematically transforms functional specifications of algorithms into parallel hardware implementations. It builds on the work of Abdallah and Hawkins [14–17] extending their treatment of data and process refinement.

This paper is divided so that the following section introduces the adopted development methodology. Section 3 presents the theoretical background. In Section 4, we put some emphasis on the approach to develop different implementations of the matrix multiplication algorithm. The following section details the development steps. Section 7 demonstrates selected implementations. In Section 8, we analyze and evaluate the performance of the suggested implementations. Finally, Section 9 concludes the paper.

2. The development method

Although compilers can expose parallelism through data flow analysis [18], imperative languages are perhaps not ideal as a starting point. This is because imperative programs already incorporate design decisions (concerning control flows and data structures), preconditions (that can be assumed), post-conditions (that must be achieved), and invariants (that must be maintained). The direct manipulation of state makes it both difficult to prove that any two pieces of code are equivalent, and to perform substitutions, modify and rewrite the algorithm. Functional languages [19], such as *Haskell* [20], however, do not manipulate state directly, and as such gain the property of referential transparency. Any sub-expression of an algorithm can be substituted for any other that is provably equivalent. This is aided by an effective set of laws given to us by such reasoning frameworks as *Bird-Merteen* formalism (*BMF*) [21], along with a wealth of other work in the functional programming and parallel processing fields [22–27].

Although, many hardware development methods still use the powerful data flow analysis, such as Viva [13], the attractions for using the functional paradigm has incited many researchers. This triggered many investigations in this area, such as *Lava* [28], *Hawk* [29,30], *Hydra* [31], *HML* [32], *MHDL* [33], *DDD* system [34], *SAFL* [35], *MuFP* [36], *Ruby* [37], and *Form* [38].

The suggested development model adopts the transformational programming approach for deriving massively parallel algorithms from functional specifications (see Fig. 1). The functional notation is used for specifying algorithms and for reasoning about them. This is usually done by carefully combining a small number of higher-order functions that serve as the basic building blocks for writing high-level programs. The systematic methods for massive parallelisation of algorithms work by carefully composing an "off-the-shelf" massively parallel implementation of each of the building blocks involved in the algorithm. The underlying parallelisation techniques are based on both pipelining and data parallelism.

Higher-order functions, such as *map*, *filter*, *foldl*, and *foldr*, provide a high degree of abstraction in functional programs [20]. Not only they do allow clear and succinct specifications for a large class of algorithms, but they also are ideal starting points for generating efficient implementations by a process of mathematical calculation using *BMF*. Over the past decade, there have been attempts to apply *BMF* for generating data-parallel programs from abstract specifications using the skeleton approach [24,15]. The main attraction of this approach is the potential for increasing reusability of parallel programs without sacrificing too much performance. The essence of this approach is to design a generic solution once, and to use instances of the design many times for various applications. Accordingly, this approach allows portability by implementing the design on different parallel architectures.

In order to develop generic solutions for general parallel architectures, it is necessary to formulate the design within a concurrency framework such as *CSP* [15,8]. Often parallel functional programs show peculiar behaviours which are only understandable in the terms of concurrency rather