Reconfigurable Computing in the New Age of Parallelism

Walid Najjar and Jason Villarreal

Department of Computer Science and Engineering
University of California Riverside
Riverside, CA 92521, USA
{najjar,villarre}@cs.ucr.edu

Abstract. Reconfigurable computing is an emerging paradigm enabled by the growth in size and speed of FPGAs. In this paper we discuss its place in the evolution of computing as a technology as well as the role it can play in the current technology outlook. We discuss the evolution of ROCCC (Riverside Optimizing Compiler for Configurable Computing) in this context.

Keywords: Reconfigurable computing, FPGAs.

1 Introduction

Reconfigurable computing (RC) has emerged in recent years as a novel computing paradigm most often proposed as complementing traditional CPU-based computing. This paper is an attempt to situate this computing model in the historical evolution of computing in general over the past half century and in doing so define the parameters of its viability, its potentials and the challenges it faces.

In this section we briefly summarize the main factors that have contributed to the current rise of RC as a computing paradigm. The RC model, its potentials and challenges are described in Section 2. Section 3 describes the ROCCC 2.0 (Riverside Optimizing Compiler for Configurable Computing), a C to HDL compilation tool who objective is to raise the programming abstraction level for RC while providing the user with both a top down and a bottom-up approach to designing FPGA-based code accelerators.

1.1 The Role of the von Neumann Model

Over a little more than half a century, computing has emerged from non-existence, as a technology, to being a major component in the world's economy. It has a profound impact on the daily life of a large number of this planet's inhabitants. Such a rapid evolution of a technology is unprecedented in human history. Probably, the single most important enabling factor of this emergence has been the von Neumann, or stored program, model of computation where a single storage unit holds both instructions and data. Prior to the proposal of this model by Alan Turing and John von Neumann "computers" relied on a fixed program architecture where re-programming involved re-wiring or the re-setting of switches.

K. Bertels et al. (Eds.): SAMOS 2009, LNCS 5657, pp. 255–262, 2009.

In this model a program is expressed as a sequence of instructions stored in memory. A Central Processing Unit (CPU) fetches the instruction from memory and executes them. The execution of an instruction implies a predetermined order of operations. Each instruction is assumed to be atomic, i.e. uninterruptible, and sequential, i.e. no instruction is started before the previous one is completed. The program dynamically determines the ordering of instructions.

The stored program model provides the common conceptual framework upon which all the elements of a modern computer systems are built: architectures, micro-architectures, languages, compilers, operating systems, applications, I/O devices, all refer to this common framework. Instruction Set Architectures (ISAs), reflecting this model, quickly defined the boundary between the software and hardware realms. Later, micro-architectures provided a variety of structural implementations within the same ISA. The model is implicit in the definition of all imperative languages (e.g. FORTRAN, Algol, Pascal, C, C++, Java etc.) as evidenced by expressions such as I = I+1, where I points to a memory location rather than a mathematical variable.

It is impossible to over-emphasize the role this common framework has played in the evolution of computing as we have experienced it.

However, the stored program model has its limitations:

- *The von Neumann bottleneck* refers to the limited bandwidth between the memory and CPU. This limitation has given rise to cache architectures and virtual memory.
- *The sequential execution* severely limits the potential parallelism in programs. Efforts to overcome this limitation include instruction level parallelism in the micro-architecture, macro-level parallel architectures such as SIMD, MIMD, vector machines, SPMD etc.

These limitations have provided the main impetus to the vast and very fruitful research efforts in computer architecture, micro-architecture, parallel architectures, compilers, language designs, algorithms, operating systems etc. This work has provided a tremendous insight into the nature of computing using this paradigm, which was translated into a tremendous improvement in performance.

1.2 The Role of Moore's Law

Over the past 50 years, the achievable computing performance increased by more than 10 orders of magnitude!

This was the VLSI (Very Large Scale Integration) revolution. It was driven by two factors: (1) Moore's Law that stated that the *number of transistors on a die* would double approximately every two years, and (2) the shrinking of the feature size of transistors which resulted in a dramatic increase of the affordable clock frequency. This dramatic performance increase was accompanied by a comparable decrease in energy expanded, cost and volume.

The side effect of the VLSI revolution was that the thriving research in parallel computing models, algorithms, programming languages, parallelizing compilers and parallel architectures that had flourished in the 70s, 80s and early 90s came to a stand still and died. A very large number of companies, and an even larger number of research projects, that had developed platforms on various parallel models (SIMD,

MIMD, SMP, message passing and SPMD) folded because it was impossible to compete with Moore's Law.

Today, VLSI technology has reached a point where the shrinking of the feature size of transistors faces major technological barriers. The number of dopant ions in a single device has become so small that it poses a risk to the stability of the device over time. Furthermore, the sheer cost of new fabrication lines has become prohibitively high. However, the number of transistors on a die keeps on increasing in the form of multi-cores and many cores. This phenomenon is ushering the *New Age of Parallelism.* In this new age of parallelism we see the same, or very similar, topics being addressed in the context of multi-core computing and networks on a chip.

1.3 Field Programmable Gate Arrays

Gate arrays started life as devices intended for use as "glue logic" to replace the large numbers of LSI chips that played this role with one single device per board. As their size and speed grew, FPGAs evolved into platforms used for functional verification and rapid prototyping and then for rapid time to market. With the widening range of FPGA applications came new features in the FPGA architectures: embedded DSP cores, CPU cores, on chip Block RAM etc.

Even though research in FPGA-based hardware code acceleration has been carried for over 10 years, this application is a relatively new comer and has not yet had any major impact on the overall market for FPGA and hence on their internal architectures.

2 Reconfigurable Computing, Potentials and Challenges

There is no formal definition of reconfigurable computing. However, the generally agreed upon definition is that of a hardware structure whose functionality is defined by the user. One can argue, and some have, that an ALU is a reconfigurable structure as it can be an adder or a multiplier etc. These functionalities, however, have been defined at design time and cannot be changed.

In the late 50s and early 60 Gerlad Estrin of UCLA proposed the fixed plus variable model where a programmable hardware structure acted as a co-processor to a fixed datapath CPU [1,2,3]. This work is sometimes recognized as the early version of a reconfigurable computing structure.

2.1 The Role of FPGAs

There is no doubt that without the very rapid increase in size and speed of available FPGA devices, riding the curve of Moore's Law, there would not be any discussion of reconfigurable computing. They have provided the components used in the very early reconfigurable computing machines such as the Splash 1 and Splash 2 [4].

It has often been argued that the architecture of FPGAs is too fine grained and hence their use in reconfigurable computers implies a significant amount of overhead as opposed to coarser granularity architectures. This is true. However, FPGAs have two fundamental advantages (1) they are available *now*, and (2) any coarser granularity must target a specific subset of applications and hence would suffer similar, if not larger, overhead on other applications. There have been a number of academic and

industry projects that have set out to develop coarse-grained reconfigurable architectures. They faced two major obstacles:

1. Competing with the VLSI revolution and Moore's Law. During the time it takes to conceive of a design, develop it and have it fabricated, the performance of CPUs and DSPs has increased at least by a factor of two and the size and speed of FPGAs have increased by a similar factor.
2. Developing a suitable tool chain for their programmability. This is the most challenging obstacle. These tools have to parallelize a source code written in a sequential language (C/C++), partition it and map it on an array of processors and schedule the execution. While some very significant breakthroughs have been achieved these remain daunting obstacles.

However, the main challenge faced by these efforts is the *application specialization*. It is clear that for a narrowly defined set of applications one can develop a coarse-grained architecture that outperforms FPGAs. However, as the set of applications gets broader, the efficiency advantages of this architecture diminish.

2.2 Applications of Reconfigurable Computing

Applications of the reconfigurable computing, today, can be divided in two broad categories:

- **Embedded systems.** Including high-performance embedded systems, air or space borne computers, telecommunication machinery etc. In these systems the vendors develop the applications and the end user is generally not expected to develop new applications.
- **High performance computing.** These systems attempt to leverage the tremendous potential of high-end FPGAs to deliver a speed-up, over traditional processors, of several orders of magnitude.

While this distinction is not always clear cut these two application domains have greatly varying requirements. In the first, the cost and power consumption are primary considerations. In the second, the speed-up must be large enough to justify the added costs of a large number of high-end FPGA devices.

2.3 Potentials

A survey of all the applications that have been tried on reconfigurable computing platforms and the achievable speed-ups reported is beyond the scope of this paper. However, we will discuss why and how such speed-ups are achieved to have a better insight in developing solutions to the challenges facing this model. In doing so we will report on the analysis done in [5].

In [5] the authors present an analysis of the speed-up that can, and often is, achieved by mapping computations on an FPGA through which data is streamed. The analysis points to the main sources for the speed-up:

- The elimination of *overhead* operations. These are instructions in a loop body that manage data movement to/from the memory, index calculations and control operations.
- The *parallelism* that can be achieved on the FPGA. Typically done by unrolling loop bodies.

Other factors, not addressed in that paper, include the folding of constants into the computation, the tailoring of the data bit width, the distribution of data storage on the data path, deep pipelining, multiple clock domains etc.

The first factor of the speed-up is inherent in the von Neumann model. Decoupling the fetching of the data from the computation proper eliminates it. This advantage is mitigated or eliminated when the data must be accessed randomly in a pattern determined dynamically at run time. The streaming of the data also eliminates a number of pipeline bubbles due to cache misses, data and instructions, control operations etc. Reported experimental measurements show this *inefficiency factor* to be about one order of magnitude. The clock frequency that can be achieved on an FPGA is typically about an order of magnitude smaller than that of a CPU. It is compensated, however, by the inefficiency factor itself.

The parallelism that can be achieved on an FPGA is due, primarily, to the sheer size of these devices (FPGAs are the largest chips being fabricated) and is very substantially enhanced by optimizations that shrink the size of the circuits such as constant folding (i.e. using logic instead of registers), variable bit width, distributed storage etc. The degree of parallelism reported in research papers is typically measured in orders of magnitude. It is the primary factor of the measured speed-up over a microprocessor.

In spite of their tremendous potentials, FPGAs and reconfigurable computing in general are not yet in the main stream as computing platforms. The adoption of this new paradigm hinges on the mitigation of a number of challenges. One non-technical obstacle is the perception of FPGAs as *exotic* devices. Compared to microprocessors and DSPs, FPGAs are relative newcomers. The research into their use for general applications is less than a decade old. They are not covered or used in traditional computer science curricula. However, addressing the other challenges can lift this obstacle.

2.4 Challenges

Programmability and Tool Chain. The programming of FPGAs is typically done using HDLs (Hardware Description Languages) and requires a solid background in logic and circuit design. Furthermore, the programming tools chain is long and complex when compared to the simple compilation step of traditional languages. HDLs rely on a radically different paradigm than imperative languages: they describe reactive entities, support timing and concurrency.

A wider acceptance of FPGAs as code accelerators requires the existence of a programmability model that leverages the popularity of widely used programming languages such as C/C++ and Java as well as a programming tool chain that is capable of abstracting away the details and intricacies of FPGA architectures.

Algorithms and Applications. Ideally, the porting of an application code to an FPGA accelerator would involve the compiling of the frequently executed code segments (typically a loop nest) to hardware. However this is hardly the case. Most high-performance application codes have been thoroughly optimized to perform well on sequential machines and do not lend themselves to such a simple porting. Most often a complete re-working of the algorithm and application code is necessary to achieve a speed-up that justifies the added costs of an FPGA platform.

FPGA acceleration is ideally suited for applications where large amounts of data can be streamed through a circuit. Furthermore, the computation that is implemented on the circuit cannot rely on a large data structure, as is often the case in on a Von Neumann platform. Very large hash tables are a most typical example. Access to such a table from an FPGA would eliminate all potentials for parallelism on the device as memory accesses would have to be serialized. What is required, therefore, is either a structuring of the existing algorithm or the development of a new one that supports (1) the streaming of data and (2) relies on a relatively small state space for the computation.

3 Generating Hardware Accelerators – ROCCC 2.0

The Riverside Optimizing Compiler for Configurable Circuits (ROCCC) was designed to provide an alternative to hand coding hardware for FPGAs without major loss of performance. ROCCC is an optimizing compiler designed to create hardware accelerators of software systems by translating kernels of C code into highly parallel and optimized hardware circuits. ROCCC performs many high level and parallelizing optimizations on C code including all standard compiler transformations on a much greater scale than a compiler for a Von Neumann machine. The objectives of these optimizations are the maximization of the parallelism in the generated circuit, the maximization of the clock rate of the generated circuit, and the minimization the number of off-chip memory accesses. ROCCC is not designed to compile complete systems but instead create highly efficient hardware accelerators for use in large systems.

3.1 Philosophy of ROCCC 2.0

For every high level algorithm and system there are a near-infinite number of ways to express the execution in C. Although each C algorithm can be expanded and converted into hardware in numerous ways, the best software algorithm is rarely the best hardware implementation. An inefficient hardware implementation may slow down total system execution, as described in [6] while exploring the hardware acceleration of molecular dynamics implementations. When a good hardware implementation is known, the C system code has to be structured differently and the compiler requires heavy guidance on what to generate. This varies for every application and made the implementation and use of the ROCCC compiler overly complex.

Additionally, FPGAs exist on a wide variety of platforms. When the compiler has full knowledge of the target platform, the code generation and optimization ordering become completely intertwined and unmanageable. A new platform would require not only a new port, but also a complete retuning of each optimization to match the available resources.

These observations led to the development of ROCCC 2.0. The main idea behind the development of ROCCC 2.0 is the description of hardware circuits in a bottom-up fashion through the creation of modules while still preserving the optimizations that generated efficient circuits. Each module can exist either as software or hardware. Designers describe modules using C, compile them with ROCCC, and have access to a library of previously compiled modules. IP cores may also be imported directly into the module library and then can be accessed by other modules and systems with a function call. When building complete systems, ROCCC is able to parallelize and replicate modules as needed and places each module in the correct location in a large pipeline.

ROCCC 2.0 also separates the platform specifics from a standard abstraction layer that interfaces with different memory models, vastly simplifying both the implementation and interface of each module.

3.2 Example

Describing a module is done using standard C code without any new keywords. The interface to the module is described with a *struct* detailing all of the inputs and outputs. The implementation of the module is described in a function that takes and returns an instance of this *struct*. Figure 1 shows a 5-TAP FIR filter module. The FIR filter has 5 inputs and one output specified in the interface *struct*. The implementation performs multiplication and addition against a constant array, which is propagated and eliminated through high-level transformations.

```
typedef struct
{
  int A0_in ;
  int A1_in ;
  int A2_in ;
  int A3_in ;
  int A4_in ;
  int result_out ;
  } FIR_t ;

FIR_t FIR(FIR_t f)
{
 const int T[5] = {3,5,7,9,11};
 f.result_out = f.A0_in * T[0] +
       f.A1_in * T[1] +
       f.A2_in * T[2] +
       f.A3_in * T[3] +
       f.A4_in * T[4] ;

    return f ;
  }
```

Fig. 1. A 5-TAP FIR Filter Module in C

```
#include "roccc-library.h"

void FIRSystem()
  {
    int A[100] ;
    int B[100] ;
    int i ;
    int output ;
    for (i = 0 ; i < 100 ;++i)
    {
      FIR(A[i], A[i+1], A[i+2],
  A[i+3], A[i+4], output) ;
      B[i] = output ;
    }
  }
```

Fig. 2. A Complete System in C That Includes the FIR Module

Once compiled with ROCCC, the FIR filter may be used by other C code either as a C procedure (by calling the function in Figure 1) or as a hardware module. Figure 2 illustrates both the creation of a complete system in hardware, which includes accessing streams of data, and the inclusion of a hardware module. All modules available

for use are exported in a library format as both hardware descriptions in VHDL and C function declarations for use at the C level.

The arrays *A* and *B* in Figure 2 are detected and analyzed by ROCCC to be input and output streams of data, and the appropriate abstracted interface is created. The stream interface is defined by ROCCC and not platform specific; instead it merely is a generic fifo interface with storage for data reuse. By including *roccc-library.h*, C code has access to call any previously compiled module directly. This is accomplished by calling a function with the name of the module and passing parameters that correspond to the inputs and outputs listed in the original *struct* in the same order in which they appear in the declaration.

The code in Figure 2 consists of ROCCC 1.0 code with the addition of modules. This allows us to leverage all of the previous parallelizing transformations and provide good performance of the generated hardware. The compilation process is user-directed and with the addition of modules allows for greater control of the final circuit's architecture.

4 Conclusion and Future Outlook

Reconfigurable computing shows great potential in improving the execution performance of a large class of applications. The main difficulty in utilizing this potential, however, is in the programmability of the platform and the translation of sequential systems into concurrent streaming engines. Recognizing this need, ROCCC 2.0 begins to bridge the gap between the high level sequential languages developers are familiar with, and the streaming circuit structure required for good performance.

The lowering of this barrier would allow for a wider adoption of this model, the development of application codes that are well suited for RC and pave the way for the evolution of architectures specifically designed to support reconfiguration.

References

1. Estrin, G., Viswanathan, C.R.: Organization of a "fixed-plus-variable" structure computer for eigenvalues and eigenvectors of real symmetric matrices. Journal of the ACM 9(1), 41–60 (1962)
2. Estrin, G., Turn, R.: Automatic assignment of computations in a variable structure computer system. IEEE Transactions on Electronic Computers EC-12(5), 755–773 (1963)
3. Estrin, G., Bussell, B., Turn, R., Bibb, J.: Parallel processing in a restructurable computer system. IEEE Transactions on Electronic Computers EC-12(5), 747–755 (1963)
4. Buell, D., Arnold, J., Kleinfelder, W.: Splash 2: FPGAs in a Custom Computing Machine. IEEE CS Press, Los Alamitos (1996)
5. Guo, Z., Najjar, W., Vahid, F., Vissers, K.: A Quantitative Analysis of the Speedup Factors of FPGAs over Processors. In: Symp. Field-Programmable Gate Arrays (FPGA), Monterey, CA (February 2004)
6. Villarreal, J., Najjar, W.: Compiled Hardware Acceleration of Molecular Dynamics Code. In: Int. Conf. on Field Programmable Logic and Applications (FPL 2008), Heidelberg, Germany (Septermber 2008)