Research Issues in Operating Systems for Reconfigurable Computing

Grant B. Wigley and David A. Kearney
Reconfigurable Computing Laboratory (RCL)
Advanced Computing Research Centre
University of South Australia
Mawson Lakes SA 5095
Grant.Wigley@unisa.edu.au David.Kearney@unisa.edu.au

Focus Session : Operating System Approaches for Reconfigurable Hardware

Abstract

As the number of system gates available on reconfigurable platforms increase beyond 20 million, the issue of the management of these resources and their sharing among may applications and users will become more of a concern. In this paper we describe the research issues for managing these resources in an operating system for a reconfigurable computer. We also detail a feasible set of components for the operating system and a feasible software architecture

We show there is no current operating system implementation with these components. We propose a number of performance metrics which we believe are important measures of the quality of an operating system implementation. These include fragmentation of area, algorithm performance and application performance. We complete the paper with a status report on our implementation of an operating system for a reconfigurable computer.

1. Introduction

As FPGA density increases with VLSI feature sizes below 0.15 micron, the need for time and space optimisation of reconfigurable designs will give way as the major focus of users to the need for tools to manage the complexity of systems incorporating in excess of 20 million gate equivalent designs. This will lead to demand for better design tools but will also open the way for the introduction of system software for the management of pre-designed reconfigurable cores. In the area of traditional computing the latter is the preserve of an operating system (OS). Therefore OS like software will be needed for reconfigurable computing (RC). This transformation is of course nothing new as it mirrors what has already occurred in the general software area, where many aspects of software implementation that were once the preserve of the application programmer have been transformed to the standard components of system software. The RC research community thus needs to further investigate the requirements and technical implementation of reconfigurable system software and in particular an operating system for reconfigurable computing.

This paper is organised as follows. Section 2 details previous work in the area of operating systems for reconfigurable computing and aims to show that no one has yet *implemented* all the components we identified as essential for an OS for reconfigurable computing. Section 3 will discuss the minimum services that must be provided by an operating system for a reconfigurable computer. Section 4 will cover the different components of the OS, including packing, partitioning and routing. Section 5 will discuss a number of abstractions that impact on an operating system for reconfigurable computing. Section 6 will discuss the number of different performance metrics that we will use to show the benefit of our system. Section 7 will detail our current implementation details and finally sum up our paper.

2. Existing Research on OS for RC

Operating systems traditionally provide run time support for applications. Surprisingly in view of the number of reconfigurable platforms (Wildforce, Splash2) and architectures (GARP, SCORE, and Morphosys) proposed and built, very few of these projects have included an investigation into run time support. Everybody who ever built a platform has seen the need for a single user loader [1], often in the guise of interface software between the RC platform and the host system. Some researchers have seen the need for a run time environment.

Brebner coined the term virtual hardware operating system [2]. He explores some of the fundamental issues that will influence the construction of any operating system for FPGA's with dynamic reconfiguration. He proposes that applications be designed into relocatable cores known as swappable logic units (SLU) [3]. He identifies the main responsibilities of an operating system to be placement of SLU's and providing for "bus addressable" registers for communication between SLU's. The whole system described by him was in fact simulated in C, including a simulation of the FPGA.

Shiraz [4] propose a reconfiguration manager which comprises a monitor which seems to receive interrupt like requests from applications already running, notifying the loader to place a new configuration at a particular place on the FPGA. This paper does not discuss how this new place is to be found (either in the initial load of the FPGA or subsequently). It is assumed that the monitor is really just a mechanism for swapping between a number of different cores in the same location. Thus there is no *allocation* in this proposal. The other elements of this proposal are a traditional loader and a configuration store.

Davis et. al. [5] they have developed a Java runtime environment for reconfigurable computing. At the top level is a hardware object scheduler which managers precompiled cores, in the middle is a place and route layer, and at the bottom is a virtualization of the FPGA to make the system portable. The authors miss the need for an allocator. There is also no evidence in their paper of the authors actually attempting to implement the layered architecture.

Rakhmatov [6] proposed a runtime environment for their architecture of multiple microprocessors with attached FPGA's. They make mention of many of the components we discuss in this review however they only deeply explore hardware clustering and I/O scheduling.

Jean [7] reported a resource manager for reconfigurable computing. There are difficulties in the published report of this manager. There are no details provided concerning the way the algorithm graphs for the applications are constructed, the allocation is on a per chip basis similar to our previous work [8]. There is no allocation on an area basis. The performance figures quoted are of questionable interest because the initialization time of the platform is included in the non RC data, even though it seems that a non RC version with a single initialisation could easily be constructed. If this were considered then the RC overhead would be much worse that quoted. In fact the overheads of the resource manager have not been fully investigated. The theses from which the paper was developed [9] indicate that there are significant overheads in data transfer for the applications concerned. So in summary this paper shows the possibility of a resource manager for RC but falls short of either a compelling and well documented implementation or a deep investigation of the issues involved.

Burns [10] have considered some of the design issues in a run time system. They point out the need for allocation of area and suggest that circuits be transformed to fit available space rather than wasting space to accommodate odd shaped circuits. They introduced the need for the circuit to interrupt the OS when it is to be swapped out. They do not consider any standard representation of the hardware in graph format and so do not have a concept of partitioning. They make the good suggestion that the choice of resources shared by the OS should be motivated by the applications.

In summary we far as we are aware one has actually tried to build an operating system for reconfigurable computing if the definition of operating system is to extend to allocation of area resources and not just to be a loader of applications. Some suggestions have been made about possible implementation of OS structures but no one has done an implementation. However there is agreement in the literature about some of the general tasks that an OS might need to perform. As we shall see later, in particular the key performance issues surrounding the relocating of hardware cores has received quite a bit of study [11]. We also note that the issue of resource sharing has been little touched in the reconfigurable OS research although this is often though to be a key issue in more traditional OS literature.

3. Services Provided by an OS for RC

Similar to a traditional OS, an OS for RC has a set of core services that must be provided. These set of minimum requirements include an application loader, FPGA area management, resource and application scheduling, application protection, and IO. Each of these requirements will be described in detail below.

3.1. Application Loading

A fundamental service provided by a traditional OS is the loading and initiation of execution of programs. In a reconfigurable computer this task is a little more complex because programs can consist of a combination of logic circuits and memory. Loading a reconfigurable computing "program" or application means placing the circuit and embedded RAM on the FPGA and then routing its external I/O interface to either neighbouring circuits or to a local or global communications bus. Note that when we use the words placement and routing we may not mean that this is done at configurable logic block (CLB) level as is common for FPGA place and route tools. Rather we assume that some placement and routing has been done at the low level and that the module is thus, in the sense of software technology, both a precompiled and relocatable module. Note also that the needs of placement and routing of the OS may be quite different to that normally associated with the tools used in FPGA logic application development. In particular, the time available for the place and route in the context of a reconfigurable OS is a significant constraint on the types of algorithms used and on the level of optimization that can be achieved. Another difference with operating systems for reconfigurable hardware is that loading an application onto an FPGA implies immediate execution whilst loading a software program into RAM does not.

3.2. Partitioning and Memory Management

The situation with virtual memory on a reconfigurable computer is similar to the traditional case. Applications can

be loaded beyond the capacity of the FPGA resources and the OS can select parts of applications that will be placed on the FPGA at any time. The difference is that splitting (called 'dynamic partitioning' in this paper) the reconfigurable application to fit into available "page frames" (area on the FPGA) which are not currently occupied by other circuits is non-trivial because circuits can not be arbitrarily partitioned. Making use of locality of reference and locality of time in reconfigurable applications has not been widely investigated. Applications of FPGAs are inherently two-dimensional although it is possible to imagine that they could be treated as one-dimensional by a restriction on the design, placement and routing. In this paper we concentrate on the initial issues of partitioning applications too big to fit on the existing resources and on loading multiple applications onto a single FPGA computer.

3.3. Scheduling

A next major issue in a traditional OS is scheduling. Scheduling reconfigurable applications is different because there is no obvious ways to pre-empt a reconfigurable application due to the typical absent of the instruction fetch, decode, and, execute cycle. Thus there is no predefined point of completion in a reconfigurable application unless the designer specifically provides this. As a consequence all multitasking of reconfigurable applications can be viewed as cooperative. This seems a serious limitation until it is realised that reconfigurable platforms are inherently multiprocessing at the granularity of each application and perhaps even at a finer granularity. A rogue application in a cooperative reconfigurable environment may not stop other applications from being loaded and executed. There is the ability on most FPGAs to delete an application pre-emptively. In the development of our OS we assume that the application designer provides a well-defined completion signal for their application or alternatively, where applications are automatically partitioned the partitioning algorithm inserts these signals at the cut point.

3.4. Protection and IO

Protection is an issue in all operating systems. On an FPGA, protection implies a bounding box beyond which the application circuit cannot be interconnected. This is the approach taken in this early version of the research described here. Another method of application protection is design rule checking. This would involve the OS having a set of rules that each application must conform to before loading it onto the FPGA surface. This type of protection is yet to be implemented.

DMA is a natural I/O mechanism for reconfigurable applications because the DMA hardware can be part of the application. As noted interrupts have little meaning in reconfigurable computing because there is no obvious processor cycle in most applications. Of course this is not to say that checkpoints could not be added to applications to make pre-emption possible but this will not be considered in this paper.

At the most basic level the resources that certainly need to be shared are the access ports to RAM directly connected to the FPGA.

4. OS Component Architecture

In this section we introduce what we believe are a minimum set of components of an OS for RC. Thus we believe that any implementation must include almost all the components listed in the next section.

An architecture containing these components is shown in figure 1. Applications requiring processing on the FPGA arrive as inputs to the OS. Each of application consists of a task graph comprising of pre-placed and pre-routed cores as nodes and communicating arcs representing dependencies. If the OS determines the application is going to fit somewhere on the unused FPGA area, then the nodes are placed in a rectangular bounding box and this box is allocated to a position on the FPGA. This gives rise to a 2 stage packing component comprising of allocation component and a placement component. If the OS determines the application wont fit it is partitioned before the 2 stage packing process.

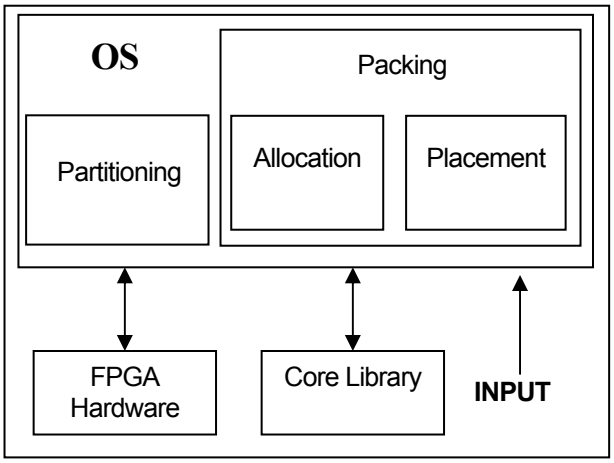

Figure 1 – Architecture of the OS

4.1. Two Stage Packing

In an OS for RC we define a two-stage process known as packing. Packing by our definition is a process of determining where on the FPGA to put the application and how to geometrically arrange the individual cores that make up the application, according to a set of rules.

Stage one of the packing process is known as allocation. We define allocation to be the process of determining where on the FPGA surface to place the application. This will

involve the algorithm to query the FPGA surface to determine where a specific amount of free area might be. Inputs to the allocator will be the geometry of the free space on the FPGA (including information about the global routing) and a pre-packed rectangular area from the placer. The output of the allocator is a position to place the pre-packed rectangle so that it does not intersect with any other exiting bounding box of a core on the FPGA or any global routing wire.

The second stage of the two-stage packing algorithm is placement. Again we define placement to be the process of determining what cores of the application are to be placed onto which cells of the FPGA. Inputs to the placer are a possible partial task graph (of pre-placed and pre-routed cores) and a target rectangular space slice. The placer algorithm arranges the nodes of the task graph inside the given rectangle allowing for their precedence by placing nodes that need to directly communicate next to one another (after allowing for the communication buffer area). The packing process is complete after both processes have executed. The detailed issues surrounding design choices for packing deserve more attention from the research community. Teich, Fekete, and Schepers [12] have made a good start in solving the placement problem. Whether the standard allocation algorithm can be used is yet to be determined.

4.2. Temporal Partitioner

Partitioning is the process whereby a large task graph is broken down into smaller components, similar to that of dividing standard programs up into pages. The inputs to the partitioner are the full task graph and a target node count. The partitioner returns a subset of the task graph which has area less than or equal to the target node count. The node count of the returned task graph may be less then the target if loops in the graph prevent a partition with the exact node count. There may be a degenerate case where the node count of the returned graph is zero. In this case further iterations of the upper levels of the algorithms may be needed or the task graph may need to wait for more area on the FPGAs. In [13] we show how a fixed sized temporal partitioning algorithm can be modified to suit the variable partition sized needs of an OS for RC. We the see the need for further research in efficient handling loops in task graphs and optimising communication between parts of a partitioned task graph.

4.3. Global Router

Routing is the process whereby electrical connections are made between two points by setting the appropriate routing switches. Traditional routing implies that the complete application is routed, i.e. at the CLB level. We define global routing to be the process of routing pre-placed and pre-routed cores together. Since the applications have been pre-routed, the only routing required is to externally route these blocks to either to other communicating modules or to external IO.

5. Key OS Abstractions

As in a traditional operating system, an OS for RC will require a number of different abstractions. These will include such items as a relocatable core library, responsible for the interface between each of the cores and an application architecture. Abstractions are also needed detailing how to communicate between the hardware applications and the software based operating system. Hardware hardware abstractions, defining how different application cores with each other and IO are required. More details of each are given below.

5.1. Relocatable Core Library

Pre-placed and routed cores will need to have a standard interface for use in an operating system. At the most basic level this means a standard format for ports on the hardware. An example of a static standard is the JBits core library standard. A more complex requirement is a standard protocol for each core to interchange with other cores.

5.2. Application Architecture Abstraction

As an OS is by its nature a general purpose tool it would seem to be unwise to commit its structure too heavily on one application architecture. However application architecture choices impact on performance. In the software area of course the OS runs on the same architecture as the applications and that architecture is a narrowly defined von Neumann processor. On an FPGA we expect that the OS and the applications might want use different internal structures so the question for the OS designer is just what is a minimum shared structure for these. The focus on the application architecture we believe should be identifying items that are shared between applications either because they must communicate using the shared resource or because the shared resource has limited capacity such as RAM interfaces. The whole issue of the relationship between application architectures and the OS performance and design is still an open question since there have been almost no implementations of an OS at all and what has been implemented has taken the route of being closely aligned to a well specified specific architecture.

5.3. Hardware/Software Communication Abstraction

Unless all the applications running under the OS and the OS itself are implemented in hardware there will be a need for a standardized interface between hardware cores and software threads. We do not expect from experience with applications that have been published (eg SAT solver) [14] that it would be wise from a performance and design complexity point of view to have all the application functions in hardware. Rarely used functions and complex control structures may not be the best choice for hardware implementation. The only published abstractions [8, 15] have a device driver with a message based socket interface to the software applications. It is well known that there are significant overheads in this as compared to an socketless interface which may be lead to a loss of performance for the hardware module. The other option is a method call interface but it is an open question whether this can be engineered with any better performance that the socket based one. As the performance of the software hardware interface is crucial to the any OS which involves software and hardware components further work on this is important for the OS research community.

5.4. Hardware/Hardware Communication Abstraction

Typically for performance reasons cores will need to communicate between each other using hardware only channels. In addition core access to memory will need to be in hardware. Whilst most of the application architectures suggested [16] have by necessity a notion of intercore communication they are very specific to the architectures which themselves are very prescriptive. Perhaps uniquely the RAW projects [17] has realised a compiler that generates interconnection structures. We view this compilation of interfaces as part of the application as distinct from the OS. We believe that the OS should also have its own structures for inter-application communication. This allows you to have different compilers for the same OS, a situation which is commonplace in the software arena. If a standardized core abstraction includes a fixed communication interface then the intercore communication could be implemented by abutment. Another possible option for the OS is to have a bus structure similar to common computing platforms and some SOC proposals [18]. However we argue that bus structures are probably a poor choice for reconfigurable computing since they serialize all communication between cores. The bus structure also unnecessarily constrains the layout of cores on the FPGA to bus interface locations. It would seem that the most promising area of investigation for interfaces is a paramaterisiable communication core that allows the OS to generate interfaces for heterogeneous application cores thus taking advantage of the reconfigurable nature of the platform and not unduly constrain the layout or serialize all communications.

Interfaces to fast memory attached directly to the FPGA are a special case of the hardware to hardware interface and are likely to play an even more important role in determining the performance of the many applications that need extensive off chip storage. There may be a case to have a fixed portion of this interface to ensure that this performance is achieved.

5.5. Global Routing Abstraction

We call the routing between hardware cores global routing. Whilst many FPGA platforms have hierarchical routing resources it would seem to be unnecessarily complex to have a routing abstraction that exposed these many levels. Thus a single level of routing could be assumed by the OS and it could be left to the tools to optimize the routing using the available resources.

6. Performance Measures

Like any operating system, performance can not be ignored. But what exactly is performance with an operating system for reconfigurable computing? As we have shown there are no implementations of an operating system for reconfigurable computing according to our definition. Therefore we are unable to benchmark it against other working models. We therefore have looked to a number of performance metrics that we can benchmark our implementation against.

6.1. Performance Overheads

Probably the most important metric for the OS to be tested against, is the performance overhead introduced by the OS. The introduction of any OS to an architecture will reduce the performance of the complete system. Clearly the OS on a von Neumann processor reduces the run-time of the applications. But most users accept this reduction in performance for an increase in ease of use of the machine. Computers wouldn't be in every home and office today if not for an easy to use OS. The key to the OS is to minimise the overhead introduced to gain this ease of use.

We intend to do this by selecting and developing fast algorithms that perform the packing and partitioning. Although there are a number of different types of algorithms that have already been developed to solve these problems, they mostly maximise the quality of result, at a cost of long run-time. This is the sort of overhead the OS can not afford to have. Therefore we are working on a number of algorithms that reduce the quality of result, but substantially reduce the run-time [13].

6.2. Application Performance

The other type of performance metric that must be considered is the reduction in application specific performance due to the introduction of the OS. Although closely related to performance overhead, application performance is the reduction in performance due to packing and or partitioning of the individual application. Since an OS has to accommodate multiple applications, the applications may have to be modified to use these algorithms. The use of these algorithms may introduce another overhead, which will have to be minimised. If an OS wasn't used, the designer would assume they have exclusive use of the hardware and would design their application in such a way to gain maximum performance.

6.3. Area Fragmentation

Fragmentation is usually associated in a traditional software OS environment with a loss of contiguous locations to store a particular application program. In the two dimensional environment of a RC there is a need to generalize this concept.

We introduced two types of fragmentation in [13]. Partition internal fragmentation was defined as the amount for free space located within one partition, caused by the placement and partitioning algorithms inability to utilize the free space, thus leaving gaps between cores. External fragmentation was the amount of free space not used compared with the space used on the FPGA surface. As applications arrive and leave the OS, the free space would become fragmented and potentially less useable as there would be more and more small free space partitions created. Clearly, the less fragmented the surface of the FPGA is the better the OS has performed. The worst case scenario would be every second CLB is used (in a checkerboard pattern), thus producing 100% fragmentation. The best case scenario would be all the applications be packed into a rectangle with no free space between them.

6.4. Other Metrics

Two other important metrics to benchmark the OS against are ease of application porting and ease of platform porting. Even if the OS doesn't reduce the performance of the application or introduce large overheads, it would be unworkable if application porting was made difficult. The OS has to have the ability to be able to take already designed applications and port them so they can be used by the system. This process doesn't have to be completely automated but certainly can not be a long and difficult process, otherwise designers wont use the system and see its potential benefit. Like a traditional OS, the system has to be easily ported to different architectures. This is especially true for an OS for RC as there are several different architectures of FPGAs. The OS has to have a small layer of FPGA dependent code which can be easily modified as to target a different architecture.

7. Our Implementation of an OS for RC

After a successful simulation of the operating system [19], we have decided to implement such a system using a real hardware platform. The reconfigurable platform chosen is an RC1000-pp reconfigurable development board from Celoxica [20]. It was chosen above other platforms because one, Xilinx completely supports the board with their software JBits, the board supports configuration via SelectMap, which results in being able to partially reconfigure the FPGA, and the board has a high speed PCI based connection and 4 banks of 2 Mb each of on-board RAM.

Our implementation of an operating system for a reconfigurable computer is Java-based and is integrated with Xilinx JBits. It has three major components, the loader, the packer and the partitioner. The loader accepts incoming applications from users and parses the input files into the OS format. The packer contains the allocation and placement algorithms and it modifies the bitstream according to its location on the FPGA. The final process is the partitioning and this divides the application into any number of partitions depending upon the area distribution of the FPGA. There is one final component of the OS which is responsible for the communication of the cores. When an application is partitioned the OS has to create routes between these cores in order to allow the data to correctly pass through the complete application. The OS also has a small component that interfaces with the target hardware which passes the bitstream and memory contents between the OS and the hardware. The OS is currently under development with the first complete prototype expected to be completed late 2002.

8. Conclusion

In this paper we have examined research issues of an operating system for a reconfigurable computer. We initially identified the previous work in this area and shown that there is no ready complete implementation of an OS for RC. We then detailed a set of core services that an OS must provide. These included application loading, partitioning and memory management, scheduling, protection and IO. We also listed a set of component for an OS. One such component is known as packing, defined as a process of determining where on the FPGA to put the application and how to geometrically arrange the individual cores that make up the application. Two, temporal partitioning defined as the process of dividing the application into smaller portions and three, global routing defined as finding a connection between two point by setting the appropriate routing switches. We

then introduced the concepts of abstractions including application architecture, hardware/software thread, and hardware/hardware intercore communication abstraction. As our OS is the first implementation of an OS for RC we have had to develop a number of performance metrics including performance overheads, application performance and area fragmentation. We then concluded with brief details on our current implementation of the OS.

9. Acknowledgements

The authors would like to acknowledge the support of the Sir Ross and Sir Keith Smith Fund.

10. References

[1] B. Gunther, "SPACE 2 as a Reconfigurable Stream Processor," In 4th Australasian Conference on Parallel and Real-time Systems (PART'97), Singapore, 1997.

[2] G. Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200," In 6th International Workshop on Field-Programmable Logic and Applications (FPL'96), Darmstadt, Germany, 1996.

[3] G. Brebner, "The Swappable Logic Unit: A Paradigm for Virtual Hardware," In IEEE Workshop on Field Custom Computing Machines (FCCM'97), Napa Valley, CA, USA, 1997.

[4] N. Shirazi, W. Luk, and P. Cheung, "Automating Production of Run-time Reconfigurable Designs," In IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98), Napa Valley, CA, USA, 1998.

[5] D. Davis, M. Barr, T. Bennett, S. Edwards, J. Harris, I. Miller, and C. Schanck, "A Java Development and Runtime Environment for Reconfigurable Computing," In Workshop on Parallel and Distributed Processing, (IPPS/SPDP'98), Orlando, Florida, 1998.

[6] D. Rakhmatov, S. Vrudhula, T. Brown, and A. Nagarandal, "Adaptive Multiuser Online Reconfigurable Engine," in *IEEE Design & Test of Computers*, vol. 17, 2000, pp. 53-67.

[7] J. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook, "Dynamic Reconfiguration to Support Concurrent Applications," *IEEE Transactions on Computers*, vol. 48, pp. 591-602, 1999.

[8] O. Diessel, D. Kearney, and G. Wigley, "A Web-based Multi-user Operating System for Reconfigurable Computing," In IPPS/SPDP'99 Parallel and Distributed Processing, San Juan, Puerto Rico, USA, 1999.

[9] V. Yavagal, "A Resource Manager for Configurable Computing Systems," Wright State University, 1998.

[10] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. Wit, "A Dynamic Reconfiguration Run-Time System," In IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97), Napa Valley, CA, USA, 1997.

[11] O. Diessel and H. ElGindy, "Run-time compaction of FPGA designs," In 7th International Workshop on Field-Programmable Logic and Applications (FPL'97), Berlin, Germany, 1997.

[12] J. Teich, S. Fekete, and J. Schepers, "Optimizing Dynamic Hardware Reconfigurations," University of Paderborn 97.288, 1998 1998.

[13] G. Wigley and D. Kearney, "The Management of Applications for Reconfigurable Computing using an Operating System," In Seventh Asia-Pacific Computer Systems Architecture Conference, Melbourne, Australia, 2002.

[14] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean Satisfiability with Configurable Hardware," In IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98), Napa Valley, USA, 1998.

[15] S. Guccione and D. Levi, "XBI: A Java-based interface to FPGA Hardware," In Configurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering, Bellingham, WA, 1998.

[16] S. Cadambi, J. Weener, S. Goldstein, H. Schmit, and D. Thomas, "Managing Pipeline-Reconfigurable FPGAs," In 6th International Symposium on FPGAs (FPGA97), Monetery, CA, USA, 1997.

[17] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor, "The RAW Compiler Project," In Second SUIF Compiler Workshop, Stanford, CA, 1997.

[18] B. Cordan, "An Efficient Bus Architecture for System-on-aChip Design," In IEEE Custom Integrated Circuits Conference, 1999.

[19] G. Wigley and D. Kearney, "The Development of an Operating System for Reconfigurable Computing," In IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'01), Napa Valley, 2001.

[20] Celoxica, "RC1000-PP Hardware Reference Manual," 2000.