High-level Programming for Reconfigurable Computing

Computation Structures Group Memo 391
November 25, 1996

James C. Hoe

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Area Examination at the Massachusetts Institute of Technology

High-level Programming for Reconfigurable Computing

James C. Hoe

November 25, 1996

Abstract

With strong efforts behind hardware development, the reconfigurable computing (RC) community is now producing powerful reconfigurable hardware with up to one-million gate capacity. In comparison, programming for RC systems has received less emphasis. Current approaches to RC programming often involves separate development for software, reconfigurable hardware, and their interaction. This paper investigates three RC programming approaches, each with increasing effort to unify the specification of the software and the reconfigurable hardware components. Arguments for an abstract programming environment, without the exact details about the underlying RC implementation, is put forth. The advantages and practicality of an unified, high-level programming environment is demonstrated by a hypothetical RC specifically designed to support a multithreaded programming model and a dynamic runtime resource manager.

1 Introduction

In a von Neumann stored-program architecture, the instruction set architecture (ISA) provided an important abstraction to help separate the tasks of the architects and programmers. On occasions, with major obstacles or innovations in software, this abstraction is allowed to break, and software concerns can trickle down to affect the hardware design. The static nature of the von Neumann hardware limits such opportunities, but this kind of feedback invariably leads to break-throughs in the overall system performance.

Reconfigurable computing (RC) describes a revolutionary architecture that is more conducive to this hardware-software coupling. Instead of spending the entire hardware resource to support the universality of a general-purpose ISA, RC reserves some of the resources as customizable, reconfigurable hardware. The objective of a RC compiler is no longer just targeted at a fixed ISA. Instead, on an application by application basis, the compiler can also gain additional performance through the custom hardware.

1.1 Programming Methodology for Reconfigurable Computing

Numerous performance records have been set by existing RC hardwares. However, a high-level of expertise in low-level hardware design, as well as an intimate knowledge of the RC platform, has been required to achieve the reported performance. The current norm of RC programming maintains a strongly separated model for software and reconfigurable hardware. An application development starts by manually separating the design into hardware and software components, and the two components are developed separately in their respective and drastically different environments.

Development of a suitable high-level programming model and language is crucial to the continued success of RC. Developing a RC application should not require the equivalent flow for custom hardware design. As RC hardware grows beyond the one-million gate mark, a diminishing number of "programmers" will have the ability to utilize this power without the aid of high-level programming abstractions. Attempts have been made to simplify the specification of hardware, but little progress have been made toward an abstract high-level RC programming model/language for hardware-software co-design. As in parallel programming, we find the RC community is also unsure about how to program the hardware they have created.

1.2 Assumptions about RC

This paper does not use the model where RC is purely a piece of reconfigurable hardware. Such a lop-sided resource only functions as a metastable point between hardware and software. A successful and important application in this model will migrate toward cheaper, more efficient production in ASICs, or become surpassed by the software on the next microprocessor. Other applications simply become forgotten. In this paper, we will define a RC to be a general purpose system containing both standard von Neumann processors for flexibility and reconfigurable hardware for acceleration. Advancements in technology may shift the balance between the contribution of reconfigurable hardware and software, but there will always be value added from their cooperation.

1.3 Paper Outline

This paper studies the programming of RCs. In particular we would like to investigate the suitability of various programming models and languages in the context of hardware-software co-design. We also want to better understand the relationship between the programming features and the necessary support in RC hardware design. To serve as a reference in our investigation, an overview of existing RC hardware technologies and architecture is prepared separately in Appendix A. In Section 2 through 4, three current RC programming approaches are surveyed. The methodologies are described and are evaluated for their suitability to RC programming. The first methodology covered in Section 2 is hardware-oriented, behavioral synthesis from a hardware description language (HDL). Next, the practice of adapting a high-level programming language syntax in place of HDL is discussed. In Section 4, we will study a restricted case of applying a software programming paradigm to unified RC programming. In Section 5, issues for RC programming is summarized, and a hypothetical RC system to address these issues is proposed. Section 6 compares the proposed system to another unified hardware-software co-design framework. A final summary is given in Section 7.

2 High-Level Behavioral Description

A high-level behavioral synthesis is the technology at the foundation of all high-level RC programming methodologies. To understand the power and limits of this technology, we start by examining the current state of high-level synthesis as described in a tutorial paper by McFarland, Parker and Camposano[18][1].

2.1 Ideal High-Level Behavioral Synthesis

A high-level behavioral description allows a hardware design to be specified by the behavior of its output signals relative to its input signals, without giving the implementation details. The description is typically conducted with a high-level sequential language with constructs for expressing hierarchical design and concurrency. Behavioral synthesis currently does not have the sophistication to make design choices such as partitioning a single description into pipelined or parallel modules. Given a description, the only objective of behavioral synthesis is to produce an equivalent structural representation in the form of a network of hardware macros like registers, memory, ALUs, etc. Behavioral synthesis needs to be followed by structural and logic synthesis prior to producing a final fabrication-level representation.

Like software compilation, behavioral synthesis begins by compiling the high-level description into a collection of operations in a dependency graph. Then, within the given design constraints, the synthesis optimally schedules and allocates the operations to hardware, ending with a structural description. Current optimization techniques use heuristics and artificial constraints to contain the otherwise unmanageable problem.

2.2 Reality of High-Level Synthesis

2.2.1 Yorktown Silicon Compiler

A quantitative assessment of a large behavioral synthesis exercise using the Yorktown Silicon Compiler is given by Camposano in [3]. The exercise involves synthesizing a 70,000-transistor IBM 801 RISC processor core from a 1460-statement behavioral description. High-level expressions (binary, arithmetic) and flow controls (case, if-then-else) are used, and 801 architectural registers are specified using variables and arrays. However, the partition into single-cycle pipelined modules and the design of their interfaces had to be explicitly specified. The number of register file read/write ports and Harvard memory architecture was also manually chosen. The behavioral synthesis of the entire design required 3.6 hours on an IBM 3090-200, approximately half of the overall synthesis time required to produce the final transistor-level output. The behavioral synthesis, instructed to optimize timing and to ignore area, matched the clock cycle time achieved by an unoptimized manual structural design[2]. Compared to the same unoptimized manual design, the synthesized design required 45% more combinational and 11% more state elements for 26% more transistors overall. The estimated design time savings is 80%. However, no claim is made about the correctness of this synthesized output.

[1]Area Exam paper #3

[2]Both cycle times are 49% longer than an optimized manual design.

2.2.2 Synopsis Behavioral Compiler

To update and reaffirm the previous assessment, I experimented by compiling simple designs, captured in Verilog[27] HDL, for Xilinx 4000 Field Programmable Gate Arrays (FPGAs)[31] using the Synopsis FPGA Compiler[24]. Synopsis and Verilog represent the current industry-standard tools for HDL-driven ASIC design. The compiler is efficient in optimizing finite-state machines and localized combinational logic. The compiler identifies and maps certain operations to optimized library modules. However, the synthesized datapath often included redundant elements and showed little global optimizations.[3] The compiler relied on hints given by rigidly-formed code sections and Synopsis-specific directives to infer sequential elements like registers and latches. As a result of this inference process, the efficiency of an output is highly sensitive to the exact format used in the behavioral description. In my experience, a properly coded Verilog design, containing a 32-bit non-arithmetic datapath and finite-state machine controls, can comfortably exceed 33MHz on the Xilinx 4000 FPGAs of "-3"[4] speed rating. Nevertheless, the critical path can usually be sped up significantly by replacing behavioral descriptions with structural descriptions.

Besides efficiency, I further experimented to find out how far we are from compiling a true algorithmic description into hardware. The two weaknesses identified are in the area of compiler inference for clocking and the treatment of common algorithmic constructs like loops and recursions. For example, Synopsis by default produces combinational circuits. The inclusion of sequential elements must be stated explicitly. No scheduling as described in [18] is performed[5]. Descriptions requiring scheduling (combinational feedback and some loop structures) are simply not allowed. For input into Synopsis, Verilog loop constructs must be stated as executing once per cycle so the required registers can be inferred. (An exception is a loop with compile-time constant loop bounds, which are implemented combinationally.) Recursion, supported by the Verilog **task** construct, is also rejected by Synopsis during synthesis.

2.3 Relevance to RC

A traditional hardware design process has been slow and tedious only because a design must work correctly and efficiently the first time. This rigid discipline is not necessary for programming reconfigurable hardware. Although most textbook algorithms probably are not directly synthesizable today, ideal behavioral synthesis can allow RC programmers to specify a design at the algorithmic level with a high-level of abstraction. The simplified specification process leads to a reduction in both design time and human error. Moreover, by speeding up the design process, behavioral synthesis allows trial-and-error and incremental design techniques to become a normal part of RC application development.

With the current level of technology, high-level behavioral synthesis cannot compete with manual efforts. However, these inefficiencies arise directly from the trade-offs we made to in-

[3]At least part of this inefficiency could be a manifestation of targeting FPGA's logic structures. It could also be because Xilinx's proprietary back-end tools perform its own optimizations.

[4]The number roughly corresponds to the logic delay (in nanoseconds) of Xilinx's basic structure. "-2" parts are becoming available. "-4" and "-5" parts are the most common today.

[5]Synopsis does perform allocation to allow sharing of combinational blocks to optimize area saving.

crease programmability. High-level behavioral descriptions are easier for humans because the structural and technology-dependent details, which aid computer synthesis, are removed. As an immediate solution, Camposano suggests using behavioral synthesis in conjunction with a manual system-level design. Other works suggest the use of mix-level entries, comparable to using both C and assembly code[4].

In the future, as system designs exceed the critical size that humans can process, the programming focus will switch from efficiency to manageability. However, even today, a synthesized output represents a significant improvement for a large number of novice system/hardware designers. In the short-run, certain restrictions and manual intervention has to be accepted into our normal mode of operation. Since behavioral synthesis is a hardware synthesis technique, this discussion has been based on a direct application of behavioral synthesis to the specification of reconfigurable hardware. A more interesting use requires integrating behavioral synthesis in an even higher-level framework unified for hardware and software co-specification.

3 High-level Programming Syntax for Hardware Description

The difference between the approaches in this section and the basic behavioral synthesis is the use of a popular software programming language in place of conventional HDL. In these frameworks, only the language syntax has been adopted. In other words, the description generally cannot be compiled into a meaningful program using the language's original compiler. This section focuses on the use of declarative languages in the Programmable Active Memory (PAM) effort by Vuillemin, et. al.[28][6].

3.1 PAM and PERL1DC

PAM DECPeRLe-1, with a four-by-four array of FPGAs, serves as a single, large (200K gates) reconfigurable resource attached to the host system (see Section A.3.1). The FPGA array and the host interact through two FIFOs that connects one edge of the array and the host's peripheral bus. The architectural model is explicit in the application development. A typical application involves the user software on the host streaming *continuous* data to and from PAM via the two FIFOs. On the other end, the user specified hardware operates on the host input and returns the processed results. An application development requires a manual partition into hardware and software components. The PERL1DC language supports the high-level description of the hardware portion of the application. The software portion is developed normally and uses library functions to access the interface FIFOs.

PERL1DC adopts the object-oriented C++ syntax instead of HDL to capitalize on the popularity of the root language. Technically, PERL1DC is only a structural netlister because the language only describes an interconnection of logic blocks and has no semantics for execution. In a design specification, C++ boolean operations provide shorthands for describing combinational logic. Sequential elements are instantiated using a supplied register primitive.

[6]Area Exam paper #1

The hardware interface to external SRAMs and host FIFOs are also encapsulated in library primitives. Procedure, loop, and array constructs convey hierarchical and structural relationships of modules. A feature unique to PERL1DC is the mechanism to specify the relative layout of related modules and suggest multi-FPGA partition boundaries.

By adopting a structural framework, PERL1DC avoids the difficult behavioral synthesis issues posed in the previous section. In PERL1DC's structural framework, clocking is specified by connecting the appropriate net to the clock input of the register primitive. In contrast, other behavioral-based frameworks in this category has had to rely on ad hoc rules to place implicit clock edges at procedure and/or loop boundaries[11, 22].

3.2 Relevance to RC

The choice to use the C++ syntax for the design entry provided nothing more than syntactic sugar on top of conventional HDLs. The real value-added from PERL1DC is the level of abstraction provided over hardware details. Novice users can work entirely within the abstraction of an uniform sea-of-gates, leaving the multi-FPGA design partitioning to automatic tools. Synchronous primitives insulate the user design from the difficult, and sometimes asynchronous, circuits required to access external SRAMs and host FIFOs. The second main benefit arises from PAM's full suite of support tools. A collection of proprietary and commercial tools enable a fully-automated synthesis from the user's design down to the final FPGA configurations. An extended simulation infrastructure allows hardware design to be co-simulated transparently with the application's software. The technology for final debugging on the actual hardware is even more impressive. Interactive graphical and textual source-level debugging is made possible using the FPGA's state read-back facility and software controlled clock generation. All of the above contribute to the "programmer-friendliness" of a RC system.

In consideration for performance, PAM made a few concessions on programmability. Since behavioral synthesis technology was deemed inadequate, the much more mature structural synthesis is used instead. Furthermore, implementation-specific layout and partition annotations are allowed in the description to improve the quality of synthesis. However, the biggest obstacle in supporting an abstract programming environment in PAM is set by its restrictive design for hardware-software interaction. The FIFO-based interface prevents PAM from easily supporting unified programming models which may require more general interactions.

In summary, PAM and other research in this area attempted to reduce the gap between hardware and software design by adopting a familiar software programming syntax for hardware descriptions. However, the real improvement actually comes from raising the level of abstraction and automation for hardware design. A stronger unification of hardware and software specification in this framework is difficult because the language syntax has been adopted without the semantics.

4 Software Programming Paradigm for Hardware Description

This section presents a special case where a software programming paradigm is adopted for unified RC programming, specifying both hardware and software. Furthermore, the description is conducted with an existing software programming language, so the same program could be compiled for *standard* platforms by the *standard* compiler. The discussion is based on dbC, a data-parallel programming language for Splash 2 presented by Gokhale and Minnich[12][7].

4.1 Splash 2 and dbC

Splash 2 is a linear array of sixteen Xilinx FPGAs with additional interconnectivity through a configurable full-crossbar (see Section A.3.2). Nearest-neighbor connections and crossbar connections are 36-bit wide. A special scatter-gather network also exists. This is the case where an architecture is specifically designed with a data-parallel SIMD programming model in mind.

The dbC programming model consists of a front-end element (FE) that coordinates the SIMD execution of some number of processing elements (PEs). When targeting Splash 2, the FE execution is mapped onto the host computer while the PE nodes are mapped to the FPGAs. When the number of PEs exceeds sixteen, multiple small PEs can share a single FPGA, or multiple Splash 2 units can be cascaded together.

A dbC program, similar to MPL or C* programs, is encoded with an ANSI C superset extended to support data-parallel nearest-neighbor and scatter-gather communication. Only integral data types are supported due to the limited floating-point capability of FPGA-based PEs. dbC also allows the width of integer types to be specified to conserve FPGA resources. During synthesis, a dbC program is first compiled to an intermediate C program containing operations for an abstract memory-to-memory SIMD machine. The intermediate program is then compiled to produce the FE software for the host and PE configurations to be replicated on every FPGA of the array. There was no mention of support for debugging dbC programs on the Splash 2 hardware, but presumably a program can be fully developed using one of the other supported platforms (CM-2, Terasys, or simulation on sequential systems like SUNs and Crays).

4.2 Relevance to RC

With similar needs for expressing concurrency and interaction, RC programming can be thought of as a special case of parallel programming. With dbC, the simple and intuitive data-parallel model allows the users to quickly specify a large amount of RC hardware at a high-level of abstraction. Partition and interactions between the host and FPGAs are implied in the programming model and are automatically handled by the compiler. Furthermore, since the programming model reflects the hardware model closely, compilation

[7]Area Exam paper #2

remains relatively straight-forward and efficient. Nevertheless, these advantages become disadvantages when an application falls outside of the data-parallel model.

The data-parallel paradigm is a large trade-off between generality and simplicity. To support the data-parallel model, the Splash 2 hardware had to be specifically designed, thus limiting its capability to support other models. The data-parallel programming model is also overly-specific to allow optimizations by a compiler analysis. dbC's program semantics give the users explicit control over the work assignment to FE and PEs, which in turn, is statically mapped to the host and FPGAs. With this explicit mapping, the compiler does not have the freedom to balance the workload between the host and FPGAs, and thus, the applications are strictly limited to those whose PE fits within a single FPGA. In general, the data-parallel paradigm is too restrictive to take full advantage of RC.

5 Issues and Discussion

None of the methodologies visited provide complete solutions, but each one demonstrated some of the desirable attributes. This section will summarize the desirable attributes for RC programming and will propose a system to captures them.

5.1 What to Look for in a RC Programming Environment

5.1.1 Abstract Programming Model

A familiar and intuitive abstract model reduces the human design effort. However, when a model can closely reflect implementation reality, compilation is easier, and there is a better chance to produce the "expected" output with good performance. For a given level of compiler and implementation technology, there is a trade-off between programmability and performance. For programmability, one would like a programming model to hide the architecture details of hardware and software. However, equally important, the programming model should be general enough so that a wide range of algorithms can be specified efficiently.

5.1.2 Multiple Levels of Abstractions

The inefficiencies resulting from a high-level programming model can be recovered by allowing multiple levels of programming abstractions with increasing hardware details. At the highest-level, full support for programming constructs like loops and recursion is necessary for algorithmic, not just behavioral, description. The availability of the library modules allows even novice users to quickly produce efficient results. However, an interface should also be provided so critical designs can be encoded at the appropriate low-level abstraction, where implementation details can be specified to increase synthesis efficiency.

5.1.3 Ease of Use

A popular software programming model and syntax will increase user-friendliness. Furthermore, the ability to compile the same source code for both standard and RC platforms assists

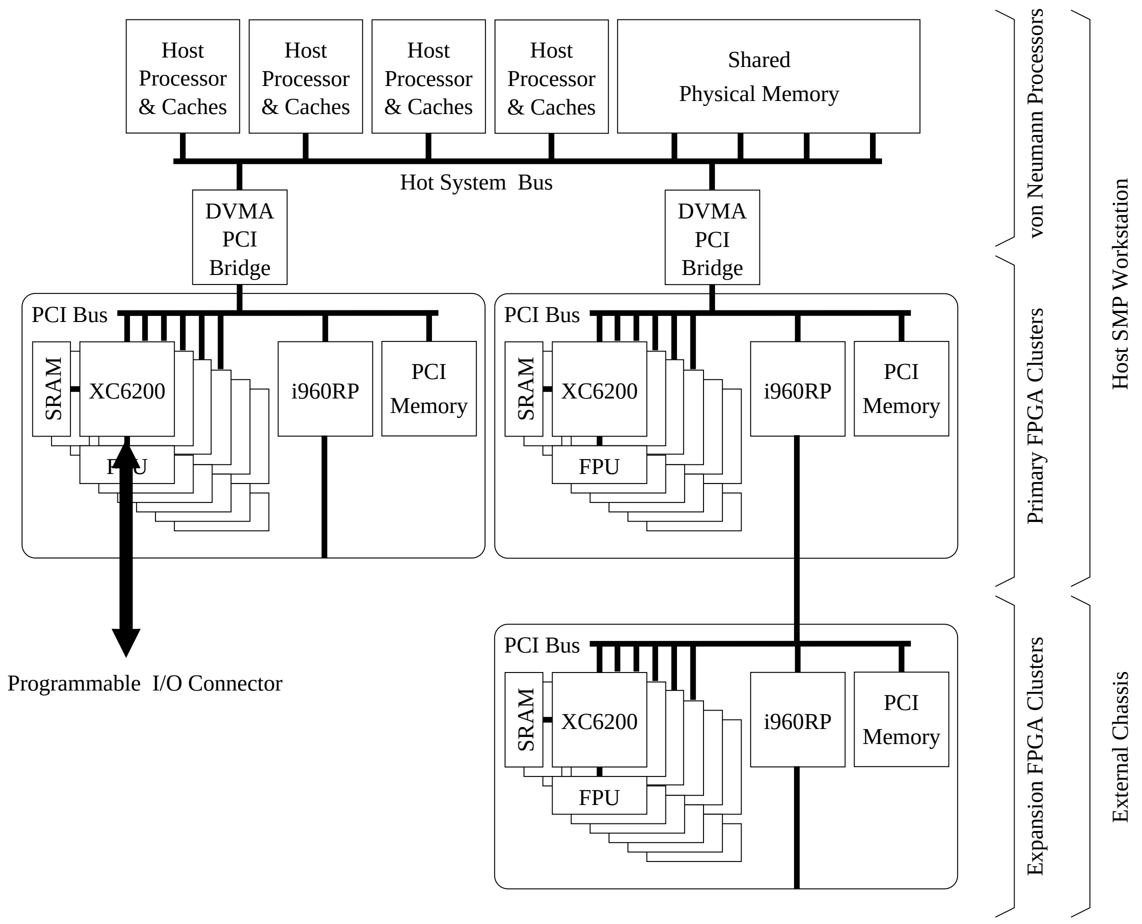

Figure 1: An Hypothetical RC Architecture

in program development and debugging. The integration and automation of RC development tools need to achieve the same convenience enjoyed in software compilations. Because of the complexity of hardware synthesis, RC compilations can be several orders of magnitude slower than software compilations. This re-emphasizes the importance that an application should be debuggable and verifiable without having to complete the full synthesis steps.

5.2 RC Architecture for a Multithreaded Programming Model

A RC system with a multithreaded programming model and dynamic hardware resource management is proposed to capture the attributes discussed above. The hardware architecture is presented first. Justification for the architecture is given along with the presentation of the programming model and the runtime system.

5.2.1 Hypothetical RC Architecture

The proposed hardware includes a standard von Neumann host system, possibly a SMP (See Figure 1). To support the reconfigurable aspects, instead of grouping many FPGAs into a single resource, six large (100K gates, 400+ I/O pins) FPGAs are individually connected to a standard bus architecture like PCI[20]. Xilinx 6200 [32] FPGAs, with built-in hardwired circuitry for a bus interface, are ideal for this application. Each FPGA has interfaces to private SRAM banks and a floating-point coprocessor. Some of the floating-point coprocessor interfaces can also optionally accept a generic cable connector to interface with special hardware. The remaining FPGA I/O pins provide the nearest-neighbor interconnections between FPGAs in a linear topology. Using Xilinx 6200's partial reconfiguration capability, the perimeter of a FPGA contains a fixed configuration to support the various interfaces, whereas the core of a FPGA is reconfigured according to the user's application.

The primary FPGA-cluster bus is connected to the host-system bus through a bus bridge with memory-management capability. In other words, similar to SUN's SBus[10] architecture, virtual addresses are used by FPGAs on the cluster bus to DMA to the shared host memory. The bridge uses a translation table to provide both address translation and protection during DMA. The bridge also ensures cache-coherence of DMA to the host memory. Besides DMA, FPGAs are individually accessible as slave devices of memory-mapped I/O from the host processors and other FPGAs. Depending on the host systems, several FPGA clusters can be attached to expand the system horizontally. An Intel i960RP[16] embedded processor on the cluster bus also allows vertical expansion through the processor's integrated PCI-to-PCI bus bridge.

5.2.2 Multithreaded Programming for RC

Rather than reinventing the wheel, the relatively intuitive multithreaded execution model with shared memory, such as the one supported by Cilk[23], is selected. The Cilk general-purpose multithreaded language captures most of the desirable attributes for RC programming. High-level abstraction and ease of use are provide by Cilk's ANSI C-based programming language. Cilk keywords, **spawn** (fork) and **sync** (join), provide the expression for multithreading concurrency. Cilk is widely supported on many platforms (both parallel and sequential) to facilitate application development and debugging.

In the Cilk programming model, each Cilk procedure specifies a sequence of threads[8]. Within a procedure, a sub-procedure call annotated with **spawn** starts a new thread for concurrent execution. For synchronization and communication between threads, shared-memory data structures can be used. Using a join-counter, **sync** provides synchronization where a parent thread blocks until all child threads have terminated. The thread execution is managed by a runtime scheduler which dynamically dispatches the threads from a ready queue to the available resources. Threads entering **sync** are enqueued into a sleep queue until they are awaken by a join-counter event.

When compiling a Cilk application for RC, in addition to the fully-executable software normally generated, procedures are selectively compiled into FPGA configurations. FPGAs are equipped with a floating-point coprocessor and DMA capability to increase the range of programs that can be supported. A hardware procedure call provides entry into a hardware thread. Protocols for inter-FPGA procedure calls, using either the bus or nearest-neighbor connection, can also be explored. Hardware thread termination is signaled by an interrupt or by a direct update to the join-counter data structure in shared memory. Shared program

[8]In Cilk jargon, a thread is an atomic sequence of execution surrounded by procedure entrance, exit, **spawn** or **sync**.

variables are mapped into the host physical memory, whereas local program variables are allocated in the private SRAM banks or internal FPGA registers. To simplify hardware management, an entire FPGA is the only unit of hardware allocation. Thus, a large thread must be sub-divided into multiple FPGA-sized threads by the compiler at the source code level. On the other hand, a subtree of small threads should be in-lined to occupy the same FPGA.

5.2.3 Runtime System

During runtime, a configuration representing a thread is dynamically loaded into one of the FPGAs. A demand-paging management scheme can be adopted to improve the utilization. When a hardware-supportable thread is spawned, the system has several options:

- If no FPGA exists or none is available, the normal software thread is spawned on the host processor.
- If the desired hardware thread is already loaded and not executing, then the new thread of execution is started using a hardware procedure call.
- If some FPGAs are idle but the desired hardware thread is not loaded, the new configuration is loaded and started on a free FPGA.
- In a more aggressive implementation, a busy FPGA could be preempted for high priority threads. This requires saving and restoring of program states on the preempted FPGA.

Loading a new hardware thread requires the system to:

- Use demand-paging heuristics to select a candidate FPGA.
- Establish the necessary DVMA address translation in the host-PCI bus bridge.
- "Link" the FPGA configuration to reflect the dynamic location of data structures and the selected FPGA.
- Download and start the thread on the selected FPGA.

FPGA configurations can be cached in i960RP's PCI memory, and runtime system tasks can be performed with the help of the i960RP embedded processors.

5.2.4 Reality Check

Although the original Cilk fully supports high-level algorithmic descriptions, we will have to observe additional restrictions due to the limits of behavioral synthesis. Recursive function calls is problematic. One could attempt a brute-force mapping of recursion to hardware by building stacks in the SRAM banks or make inter-FPGA procedure calls, but good efficiency probably requires manual algorithmic transformation. Although the presence of a floating-point coprocessor and DVMA increases the range of supportable problems, some operations still should not be compiled for hardware execution. Thus, the hardware speed-up will be dependent on the style and operations in the source code.

This is acceptable as long as the variation from the programming style is in performance and not in functionality. A poorly written program must still run. Just like carefully crafted C code produces better compiled binary, a Cilk program written to avoid the shortcomings of the system will result in better performance. Occasionally, to achieve the same goals as assembly programming, even hand-crafted HDL or schematic capture may be necessary. Expert-crafted library functions should be supplied to assist novice programmers.

The runtime support for dynamic spawning and the demand-paged management of hardware threads is probably the most fantastic notion in the whole proposal. However, these are not unexplored ideas. The support for hardware procedure calls has been studied by [2, 6]. Demand-based reconfiguration of hardware has been studied in the context of reconfigurable functional units in microprocessors[30, 29]. The multi-context FPGA configuration scheme, proposed in DPGA[7], can also facilitate rapid hardware thread swapping. Stock knowledge accumulated on multithreading and demand paging in the traditional contexts should also speed progress.

The proposed hardware architecture is an integration of standard technology which makes it highly feasible. In fact, a prototype can be constructed from stock commercial parts like a SUN workstation with SBus, SBus extenders and EVC1 or Pamette boards (see Section A.3.4 about EVC1 and Pamette). However, an important issue, as suggested in many studies, is the cost-effectiveness of FPGA-based RC in comparison to a RC architecture equipped with an array of embedded DSPs or media processors. An embedded processor-based design overlaps many of the same advantages as current FPGA-based approaches, but embedded processor systems does not require hardware synthesis technology. In any case, the multithreaded programming approach developed in this section should also be applicable to partitioning a single task description dynamically among the embedded processors.

Given the dynamic nature of execution, application debugging on the actual RC hardware will be extremely difficult. As supported in most current RC systems, hardware state readback is a crucial mechanism to debug the runtime system. However, the same mode for debugging cannot be required for the user application development. User application debugging should be interactive at the source-code level. This is not difficult since the choice to use the Cilk language allows us to target standard platforms and rely on the standard debugger technology. The weak link is the correctness of final compilation of a debugged program to synthesized hardware and software.

6 Related Work in Hardware-Software Co-design

Similar research on synthesizing a combination of hardware and software from an unified design specification is described in [14] and [26]. This area of research originates from the development of embedded systems that uses both microprocessor and custom ASICs to simultaneously minimize cost and satisfy real-time performance requirements.

In their approach, an application is described as a set of communicating sequential processes (CSP)[15]. Profiling information and user-specified timing constraints (specifying maximum or minimum latency between specific events) are used to help identify code sections for either hardware or software implementation. Small pieces of the applications are recursively grouped together according to their similarities and interactions. Next, the clusters are partitioned for hardware and software implementation, optimized according to a cost function based on meeting timing and hardware constraints and minimizing hardware-software interactions. After the partition, the changes to implement the interaction between the hardware and software components are inserted automatically, producing a system that is behaviorly equivalent to the original description.

These systems are similar to our proposed system since they also rely on a multithreaded description framework to support an unified hardware software specification. However, our fork-and-join multithreaded execution model with shared memory is more popular and can be efficiently targeted for more platforms. Furthermore, their decision on a hardware-software partition is static at compile time and assumes all CSP threads, long or short, are always alive. Whereas in our system, we take advantage of our execution model to dynamically allocate threads to either FPGAs or software execution and to reclaim a FPGA when a hardware thread terminates. Our per-thread based hardware allocation is also simpler than their global analysis. Finally, their system does not benefit from a hardware architecture designed to meet the software system requirements.

7 Summary

This paper identified the lack of a suitable programming methodology as the challenge to the progress of RC. This paper promoted the uses of a high-level programming environment to provide an unified abstraction over the underlying hardware and software. Rather than explicitly designing a separate piece of software and reconfigurable hardware, the programmer should work within the simple, intuitive programming model of a high-level programming language.

Three current RC programming approaches are studied for their desirable and missing features. Based on the findings summarized in Section 5, a hypothetical RC system is proposed to provide a simple abstract programming environment. The proposed system leverages the existing commercial hardware technology and parallel programming research. Multithreaded execution with shared memory is selected because of its intuitive abstraction and natural expression for concurrency. A runtime system that dynamically manages the reconfigurable hardware resources is suggested to support the multithreaded execution. A RC architecture is designed to meet the requirements of the programming model and the runtime system.

Research toward raising the RC programming abstraction can significantly improve the usability of existing RC systems. Efficient support for a unified high-level RC programming will remain a difficult problem given the current state of behavioral synthesis and hardware-software partition analysis. However, proper architectural support can help resolve some of the programming issues.

A Reconfigurable Computing Hardware

RC research dates back to the microcoded machines when programmable micro-control allowed some customization of the datapath control and therefore ISA. Recent RC architectures have relied on the Field Programmable Gate Array (FPGA) technology[9] to support hardware reconfiguration. This section starts by examining the FPGA technology, and then assess its potential as high-performance RCs. The section finishes with a survey of a few prominent RC architectures.

A.1 Field Programmable Gate Arrays

FPGA is an integrated circuit structure composed of a large array of simple combinational and sequential logic primitives. The exact interconnection and operation of the logic primitives are controllable by end-user accessible memory elements embedded in the primitives[9]. Using FPGA to realize a complex digital logic involves:

- Decomposing the logic into the basic functions supportable by the FPGA primitives (Technology Mapping).
- Assigning each decomposed function to a distinct physical primitive in FPGA, and determining the required interconnections (Place and Route).
- Determining the state of the memory elements that achieves the selected functions for each primitive (Generation).
- Setting the FPGA configuration memory elements to the determined states (Configuration).

With the largest FPGA reaching over 100K gates[10], Field Programmable Gate Arrays (FPGAs) are poised to replace the Application Specific Integrated Circuit (ASIC) gate arrays within a few years.

A.2 Performance of Reconfigurable Computing using FPGAs

Not to be misled by the previous claim, gate for gate, FPGA logic still occupies 100 times more area, runs 10 times slower and costs six times more than the best VLSI alternative. This difference translates to state-of-the-art von Neumann microprocessors rated at over 500 MFlops versus FPGAs capable of a few MFlops. One must question the potential of a FPGA-based RC as an improvement over traditional computing.

Fact is, however, for specific applications, current RC platforms already achieve performance unsurpassed by any implementation, including ASICs and supercomputers. In these cases, the RCs are taking advantage of the following:

[9]This discussion is limited SRAM-based reprogrammable FPGAs.

[10]A mid-1980 Intel 80386 is equivalent to many tens of thousands of gates, and a mid-1990 Intel Pentium-Pro is a few million gates.

- Exposing a application's micro-grain parallelism and other optimizations visible only to a low-level hardware implementation and not a general-purpose ISA.

- RCs can make more efficient use of available hardware resources without wastage such as 64-bit data busses and FPUs when not necessary.

- RCs with an array of FPGAs can exploit an application's coarse-grain parallelism through parallel executions.

- FPGA's reconfigurability produces superior hardware design through iterative refinements of algorithm and design trade-offs.

Thus, the winning applications are exemplified by high-throughput, 8-bit, 16-bit DSP, sequence matching, and multimedia processing. RCs typically cannot handle full precision floating point arithmetics. Nevertheless, the FPGA technology has improved rapidly. Like SRAMs, FPGAs' simple and regular structure allows it to tracks the pure VLSI technology curve and not be burdened by architectural issues facing microprocessors.

A.3 Survey of RC Platforms

Although still at an early stage of development, several RC designs have been proposed and built. Several commercial platforms are also available. At a high-level, the RC platforms are all similar. Typical features are:

- Both reconfigurable (FPGA array) and standard (workstation) hardware are used. The FPGA array is attached to a standard host computer via a standard bus or cable interface. The host computer controls the configurations of the RC hardware and executes the user software portion of the computation.

- The reconfigurable hardware is consisted of a number of FPGAs with tightly coupled discrete SRAMs parts.

- Sophisticated clocking circuits allow programmable frequency, wave form, and distribution.

- Addition I/O connectors are provided for extending the system size and for interfacing with special hardware (These generic connectors are referred to as Programmable I/O Connectors (PICs) in Table 1, Table 2 and Table 3 of this section.)

Varieties exists in the detail design and operation. The paragraphs below will highlight the reconfigurable hardware architecture of a few prominent RC systems. The systems are categorized by their intended model of operation.

RC	Core FPGAs	Reported Capacity	Topology	Memory	I/O Interfaces	Max # Boards	References
PAM DECPeRLe-1	16 XC3090	200K gates	2D mesh	4 banks x 1MB	TurboChannel PCI		[28]
Teramac	108 Plasma	64K gates	see text	4 banks x 8MB	SCSI, PIC	16	[1]
Virtual Wires	16 XC4005	20K gates	2D mesh	16 banks x 32KB	SBus RS232, PIC	10	[25]

Table 1: Summary of RC Basic Blocks with Sea-of-Gates

A.3.1 Sea of Gates

The basic reconfigurable block is a board with a moderate number (>10) of FPGAs connected in a regular pattern. The most popular topology is a 2D mesh. Some systems employ Field Programmable Interconnect Devices (FPIDs) in the topology to add routing flexibility. The basic board is often extensible via connectors or cables to form larger systems.

This class of reconfigurable hardware presents a hardware-centric, sea-of-gates abstraction. Ideally, a random logic design is specified without knowledge of the underlying FPGA. The task of mapping a generic logic design to physical resources (FPGAs, interconnects, memory, etc.) is automated through CAD tools and compilers. Table 1 summarizes the most relevant systems.

The most interesting problem in supporting the sea-of-gates abstraction is in dealing with the disproportionate reduction in interconnect-to-logic ratio at the FPGA boundaries. A design partitioned to fit the logic capacity of the FPGAs will run out of I/O pins. In other words, the design must be portioned further to meet the narrower interconnect constraint, causing each FPGA to be under-utilized. For a synchronous design, multiple off-chip signals can multiplex a given pin to increase the effective interconnect-to-logic ratio. Virtual Wires [25] provide a systematic way of automatically transforming a standard design to this framework. HP's Teramac [1] approaches the problem by designing a balanced system from ground up, using exotic technologies like custom FPGAs and 27-FPGA MCMs[11].

A.3.2 Structured Array

RCs in this category may be structurally identical to RCs in the previous category. The deciding difference is the implicit grouping of inter-FPGA interconnects into ports in the design specification environment. Instead of a sea-of-gates, each FPGA is viewed as an individual entity with an explicit connection to the other FPGAs through these ports. Often times, the same FPGA configuration is replicated through out the array for a SIMD-style operation.

Table 2 summarizes the representative systems. Wildfire[19] is a commercial system derived from Splash 2[13] through the National Security Agency's Technology Transfer Pro-

[11]Multi-Chip Module

RC	Core FPGAs	Reported Capacity	Topology	Memory	I/O Interfaces	Max # Boards	References
Splash	16 XC4010		Linear + Crossbar	16 banks 512KB	SBus		[13]
Virtual Computer	40 XC4010	400K gates	2D mesh + 24 FPID	16 banks 16KB	VME64, SBus PIC		[5]
Wildfire	16 XC4010		Linear + Crossbar	16 banks 512KB	VME64 SBus, PCI	16	[19]

Table 2: Summary of RC Basic Blocks with a Structured Array

RC	Core FPGAs	Reported Capacity	Topoloy	Memory	I/O Interfaces	References
Giga900 RIC	32 Xilinx		bus	128MB DRAM 4MB SRAM	PCI, PIC	
Pamette	4 XC4000E	110K gates	2x2	256MB DRAM 2x128KB SRAM	PCI, PMC	
Rasa	3 XC4010		2 FPIDs	10x32KB 16-bit mult	PC AT	[21]
EVC1	1 XC4013			2MB SRAM	SBus,PIC	[5]

Table 3: Summary of Small RC Coprocessing Boards

gram. Virtual Computer[5] is another commercial platform and one of the largest RC platforms in existence. The topology is a 2D mesh, but any-chip-to-any-chip connection is possible through the use of FPID.

A.3.3 Small Coprocessor Boards

A more recent direction of development is in small coprocessor boards containing small number of FPGAs, FPIDs, and memory. The entire package, housed on a form-factored peripheral card, fits inside a standard workstation. These RC systems are designed to work in close interaction with the software on host processor. The host interface has good performance and usually has bus-master capability to access the host memory. However, none of the current systems has been designed for the higher-performing memory bus; instead they reside on the secondary peripheral bus. Table 3 summarizes the systems. Price and user-friendliness of this category has attracted commercial participation such as Giga900, Pamette and EVC1.

A.3.4 Microprocessor with Reconfigurable Functional Units

In an orthogonal direction, microprocessors with all or some reconfigurable functional units are being investigated. During compilation for a reconfigurable microprocessor, the compiler can detect special cases where the addition of an unsupported instruction can improve performance. ISA can then be customized by creating new functional units. A reconfigurable instruction dispatcher, and a cache/memory controller are also proposed.[8] Two particularly interesting directions of developments are:

- Given only limited reconfigurable resources on a chip, reconfigurable functionalities can be swapped on-demand, in the fashion of demand paging.[30, 29]
- Compilers can analyze an application and generate a fully customized ISA for optimal utilization of the hardware.[17]

References

[1] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider. Teramac–configurable custom computing. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 32–38, Napa, CA, April 1995.

[2] T. B. Bauer. The design of an efficient hardware subroutine protocol for FPGAs. Master's thesis, Massachusetts Institute of Technology, May 1995.

[3] R. Camposano. Design process in the Yorktown Silicon Compiler. In *Proceedings of 25th ACM/IEEE Design Automation Conference*, pages 489–494, June 1988.

[4] R. Camposano and W. Wolf. *High Level VLSI Synthesis*, chapter 1, pages 1–26. Kluwer Academic Publishers, 1991.

[5] S. Casselman. Transformable computers. In *Proceedings of 8th International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[6] S. Casselman, M. Thornburg, and J Schewel. Hardware object programming on the EVC1: A reconfigurable computer. In *Proceedings of FPGAs for Rapid Board Development and Reconfigurable Computing (Photonics East 95)*, 1995.

[7] A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, Napa, CA, April 1994.

[8] A. DeHon. Notes on coupling processors with reconfigurable logic. Technical Report Transit Note 118, Massachusetts Institute of Technology, March 1995.

[9] B. K. Fawcett. Field programmable gate arrays and reconfigurable computing. In *Proceedings of the International Society of Optical Engineering (SPIE). Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing.*, volume 2607, pages 155–166, Philadelphia, PA, October 1995.

[10] E. H. Frank and J. D. Lyle. *SBus Specification B.0.* Sun Microsystems, Inc., 1990.

[11] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 136–144, Napa, CA, April 1995.

[12] M. Gokhale and R. Minnich. FPGA computing in a data parallel C. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–101, Napa, CA, April 1993.

[13] M. B. Gokhale and R. Minnich. VHDL programming on Splash 2. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 301–318, Napa, CA, April 1993.

[14] R. Gupta and G. de Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design and Test of Computers*, pages 29–41, September 1993.

[15] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[16] Intel Corporation. *i960 RP Microprocessor External Architecture Specification*, July 1995.

[17] C. Iseli and E. Sanchez. Beyond superscalar using FPGAs. In *Proceedings of IEEE International Conference on Computer Design*, pages 486–490, Cambridge, MA, October 1993.

[18] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[19] J. McHenry and R. Donaldson. The WILDFIRE custom configurable computer. In *Proceedings of the International Society of Optical Engineering (SPIE). Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing.*, volume 2607, pages 189–199, 0, PA, October 1995.

[20] PCI Special Interest Group. *PCI Local Bus Specification*, 2.1 edition, June 1995.

[21] H. Schmit, L. Arnstein, D. Thomas, and E. Lagnese. Behavioral synthesis for FPGA-based computing. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 125–132, Napa, CA, April 1994.

[22] Stanford University. *HardwareC – A Language for Hardware Design*, December 1990.

[23] Supercomputing Technology Group, MIT Laboratory for Computer Science. *Cilk-4.1 (Beta 1) Reference Manual*, September 1996.

[24] Synopsis, Inc. *HDL Compiler for Verilog Reference Manual.*

[25] R. Tessier, J. Babb, M. Dahl, and S. Hanono A. Agarwal. The virtual wires emulation system: A gate-efficient ASIC prototyping environment. In *Proceedings of the 1994 ACM Workshop on FPGAs*, Napa, CA, February 1994.

[26] D. E. Thomas, J. K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design and Test of Computers*, pages 6–15, September 1993.

[27] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[28] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI*, 4(1):56–69, March 1996.

[29] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 99–107, Napa, CA, April 1995.

[30] M. J. Wirthlin and B. L. Hutchings. Sequencing run-time reconfigured hardware with software. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 122–128, Monterey, CA, February 1996.

[31] Xilinx, Inc. *XC4000 Series Field Programmable Gate Arrays*, September 1996.

[32] Xilinx, Inc. *XC6200 Series Field Programmable Gate Arrays*, October 1996.

Contents