

Programming High Performance Reconfigurable Computers

Melissa C. Smith⁺ Gregory D. Peterson^{*}

The University of Tennessee, Department of Electrical and Computer Engineering

ABSTRACT

High Performance Computers (HPC) provide dramatically improved capabilities for a number of defense and commercial applications, but often are too expensive to acquire and to program. The smaller market and customized nature of HPC architectures combine to increase the cost of most such platforms. To address the problems with high hardware costs, one may create more inexpensive Beowulf clusters of dedicated commodity processors. Despite the benefit of reduced hardware costs, programming the HPC platforms to achieve high performance often proves extremely time-consuming and expensive in practice. In recent years, programming productivity gains come from the development of common APIs and libraries of functions to support distributed applications. Examples include PVM, MPI, BLAS, and VSIP. The implementation of each API or library is optimized for a given platform, but application developers can write code that is portable across specific HPC architectures.

The application of reconfigurable computing (RC) into HPC platforms promises significantly enhanced performance and flexibility at a modest cost. Unfortunately, configuring (programming) the reconfigurable computing nodes remains a challenging task and relatively little work to date has focused on potential high performance reconfigurable computing (HPRC) platforms consisting of reconfigurable nodes paired with processing nodes. This paper addresses the challenge of effectively exploiting HPRC resources by first considering the performance evaluation and optimization problem before turning to improving the programming infrastructure used for porting applications to HPRC platforms.

Keywords: high performance computing, reconfigurable computing, programming, performance evaluation

1. INTRODUCTION

High Performance Computers (HPC) provide dramatically improved capabilities for a number of defense applications, but often are too expensive to acquire and to program. The smaller market and customized nature of HPC architectures combine to increase the cost of most such platforms. To address the problems with high hardware costs, one may create more inexpensive “Beowulf” clusters of dedicated commodity processors. Despite the benefit of reduced hardware costs, programming the HPC platforms to achieve high performance often proves extremely time-consuming and expensive in practice. In recent years, programming productivity gains come from the development of common APIs and libraries of functions to support distributed applications. Examples include PVM [11], MPI [25], BLAS [1], and VSIP [3]. The implementation of each API or library is optimized for a given platform, but application developers can write code that is portable across specific HPC architectures.

The application of reconfigurable computing (RC) into HPC platforms promises significantly enhanced performance and flexibility at a modest cost. Unfortunately, configuring (programming) the reconfigurable computing nodes remains a challenging task and relatively little work to date has focused on potential HPRC platforms consisting of reconfigurable nodes paired with processing nodes. This paper addresses the challenge of effectively exploiting HPRC resources by addressing the performance evaluation and optimization problem as well as improving the programming infrastructure used for porting applications to HPRC platforms. We describe our approach to this problem domain by first discussing HPRC architectures, including the implications of combining multiple computational nodes with reconfigurable computing elements. Next, we discuss performance modeling techniques used to assess the most appropriate means of exploiting the available resources. Finally, we consider programming issues and existing design environments for high performance computing and reconfigurable computing before discussing future work.

⁺ smithmc@microsys.engr.utk.edu; <http://www.ece.utk.edu>; The University of Tennessee, Electrical and Computer Engineering, 414 Ferris Hall, Knoxville, TN, USA 37996-2100

^{*} gdp@utk.edu; phone (865)-974-6352; fax (865)-974-5483; <http://www.ece.utk.edu>; The University of Tennessee, Electrical and Computer Engineering, 414 Ferris Hall, Knoxville, TN, USA 37996-2100

2. HPRC ARCHITECTURES

As shown in Figure 1, a High Performance Reconfigurable Computing (HPRC) platform consists of a number of computing nodes connected by an interconnection network (the architecture can be a switch, hypercube, systolic array, etc.), with some or all of the computing nodes having reconfigurable computing (RC) element(s) associated with them. Additionally, an optional configurable network can be constructed to connect the RC elements for synchronization, data exchange, etc. This optional configurable network could vastly improve performance for applications such as those that require data exchange for barrier synchronization events.

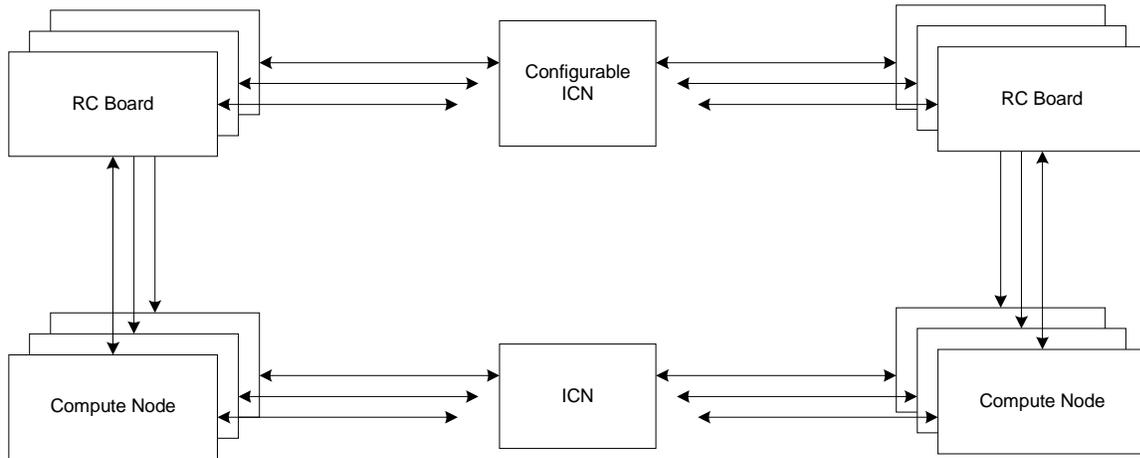


Figure 1 High Performance Reconfigurable Computer (HPRC) Architecture

Research in the architecture, configuration, and use of RC systems is ongoing. Efforts to date have primarily focused on single computing nodes with one or more RC elements, much of which has been supported by the DARPA Adaptive Computing Systems (ACS) program [2]. Even these basic building blocks of the HPRC platform remain a challenging task to efficiently configure and use. Some of the major challenges involve FPGA reconfiguration latency, hardware/software codesign, and sub optimal design tools. Often, the design time necessary to map to the RC system, the time consumed during reconfiguration, or both outweigh any performance advantages achieved by executing on the RC system.

2.1. Potential Parallelism

The HPRC platform is designed to exploit multiple types and levels of potential parallelism often found in DSP, simulation, numeric algorithms and other computationally intensive applications. Data or functional parallelism can often be exploited at different levels of abstraction, from concurrent software tasks executing on different processors to multiple functional units contained within a processor. Parallel and distributed computing research has long shown the advantages of exploiting parallelism via higher-level concurrent software for a wide range of applications. Recent research in the area of reconfigurable computing demonstrates performance advantages for many of the same applications. The reconfigurable hardware implementations in many cases have shown significant speedup over software-only solutions [17, 20-22, 24].

As shown in Figure 2, at a high level of abstraction there is coarse grain parallelism between the software tasks executing on the compute nodes. Each node could include multiple processors in a shared memory configuration to support multiple threads or processes; the interconnected compute nodes can also support distributed memory parallel processing. Beneath the high-level software tasks, there are parallel hardware and software tasks executing on the associated compute nodes and RC element(s). At the next level in the hierarchy, there are at least two options: multiple hardware tasks executing on a plurality of RC elements or bit-wise parallel operations within an RC element. Hence, we can support functional and data parallelism at a variety of levels of granularity in order to provide the maximum performance for a given application. We will discuss parallel processing techniques for the “software parallelism” achieved with multiple processors before considering the “hardware parallelism” provided by reconfigurable combinational logic embedded in RC elements.

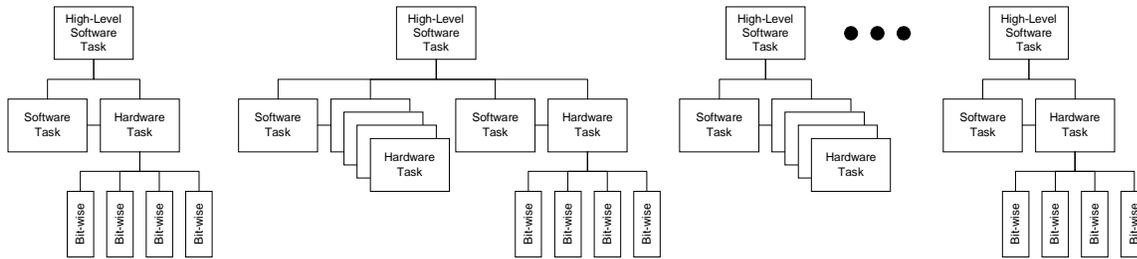


Figure 2 Hierarchy of Parallelism Exploited by the HPRC Platform

2.2. Software Parallelism

The primary thrust of the parallel processing field has been to dramatically improve performance by extracting concurrent tasks that naturally exist in applications such as simulations, signal processing computations, numeric algorithms, etc. For a given program, functional parallelism is exploited by creating software tasks to concurrently perform separate, but related, tasks. Similarly, data parallelism is exploited by replicating software tasks that operate on subsets of the problem. Parallel and distributed processing applications exploit both of these techniques, although achieving high performance depends on carefully crafting an algorithm and its implementation to best use the available processors, memory hierarchy, and interconnection network. Despite the practical difficulties, approaches to achieving high performance parallel software constitutes a relatively mature field, with a broad range of applications accelerated by HPC platforms.

2.3. Hardware Parallelism and Virtualization

RC systems can be thought of as a “demand-paged” hardware resource similar to software cache. Within the RC system, there are many different types of computations, each having a separate mapping to the reconfiguration logic. This idea of *virtual hardware* is similar to the virtual memory in today’s computers [12]. Taking this idea of virtual hardware a step further, different phases of an algorithm could have mappings to just a portion of the FPGA. Operating like a *hardware cache*, multiple mappings are co-resident in the FPGA and can interact individually with the microprocessor. The set of mappings that are co-resident can vary over time depending on the demands of the computation algorithm. A host of issues pertaining to the most appropriate architecture, memory hierarchy, computational model, and runtime infrastructure must be considered to fully exploit this approach. The RC system can then be modeled as a variation on the well-known *Harvard Architecture* as shown in Figure 3.

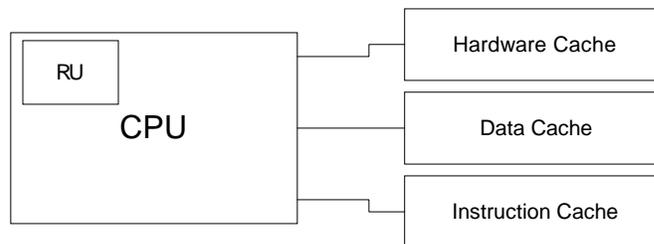


Figure 3 Harvard Architecture with Hardware Cache

During the execution of an application, different function mappings will be needed in the RC element. When the required functions exceed the available space in the RC element some or all of the function mappings will need to be replaced. Hardware cache replacement issues such as these share similarities to software caches but have some additional complexities. RC systems are more complicated because the integrity of the I/O and routing of all configurations simultaneously in a device must be maintained. When a configuration block is mapped into a device, there are three fundamental types of mappings: direct mapped, fully associative, and set associative. If a configuration is *direct mapped*, the block can only be configured into one position of the device. In *fully or set associative*, the configuration block can be positioned anywhere in the device or in a limited set of positions respectively. Additionally, in FPGAs, there is a spatial dependency due to the routing and I/O required, further complicating the mapping process. Different positioning of a function may require more or less routing resources. Even if we limit ourselves to direct mapping, we have to ensure that configurations do not overlap or create

destructive mappings. In either of the associative types of mappings, some run-time mapping will likely be necessary. Some of the work to date is discussed in [9, 10, 12, 13, 19].

By using RC elements as virtual or multi-mode hardware, fewer resources are required for an application since the RC units can be dynamically reconfigured and reused to implement multiple functions over the application's lifetime. Additionally, the only limitation on the number of possible configurations or mappings in virtual hardware implementation is the storage space for the configurations.

3. PERFORMANCE MODELS

Performance models can prove to be tremendously useful in assessing the effectiveness in exploiting the available computational resources. In order to support programming HPRC resources, we first consider performance modeling approaches and their accuracy. We can then employ these modeling approaches to better understand the tradeoffs in mapping applications to HPRC resources as well as the most effective ways of doing so. Some examples include determining the best computational granularity, decomposition of tasks and data, number and types of processors, and load balancing techniques. We consider queueing models, analytic models, and simulations for evaluating the performance of HPRC resources and applications.

3.1. Queueing Models

An exact queueing network approach to modeling applications with internal concurrency quickly results in an explosion in the size of the state space [14]. In their paper on Analytic Queueing Network Models [26], Thomasian and Bay present a recursive algorithm to compute state probabilities for directed acyclic graphs or DAGs. The algorithm uses a hierarchical model based on a Markov chain at the higher level to compute state probabilities and an analytic solution at the lower level to compute transition rates among the states of the Markov chain.

A queueing network (QN) can also be used to model the computing system when analyzing the performance of an application. Devices in the system such as CPU's, disks, communication links, etc. belong to D categories. The processing requirements of a given task are represented by their service demands, which can vary depending on the particular device to which it is assigned. In one model [26], the task system is specified as an 8-tuple, $\{T, [\prec\bullet], P, Z, Y, R, A, S\}$ whose definitions are given in Table 1. An example task graph and corresponding Markov chain is given in Figure 4 and Figure 5 respectively.

A broad range of applications can be represented with such an acyclic task graph. Aperiodic algorithms trivially can be represented with a DAG, while iterative algorithms can be represented with a DAG for each iteration executed by the application.

Table 1 Task System Model Definitions

Symbol	Definition
$T = (T_1, \dots, T_i)$	The set of tasks to be executed
$[\prec\bullet]$	A partial order defined on T specifying precedence constraints
$P = [p_{ij}]$	$I \times J$ matrix where p_{ij} is the probability that task T_j is executed after task T_i
$Z = [Z_{id}]$	$I \times D$ matrix where Z_{id} is the processing required by T_i at device type d
$Y = [Y_{ij}]$	$I \times J$ matrix specifying the amount of data communication between task T_j and task T_i
$R = [R_i]$	The passive resources required by task T_i
$A = [A_{ij}]$	$I \times J$ matrix denoting the allocation constraints in a multiprocessing system where J is the number of processing nodes; $A_{ij} = 1$ if task i can be run on node j and $A_{ij} = 0$ otherwise
S	The system scheduler

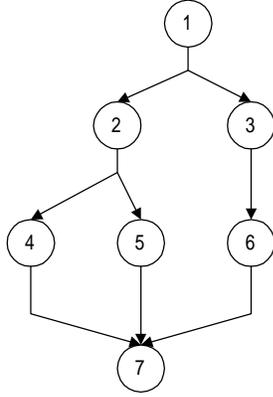


Figure 4 Task Graph for Parallel Application

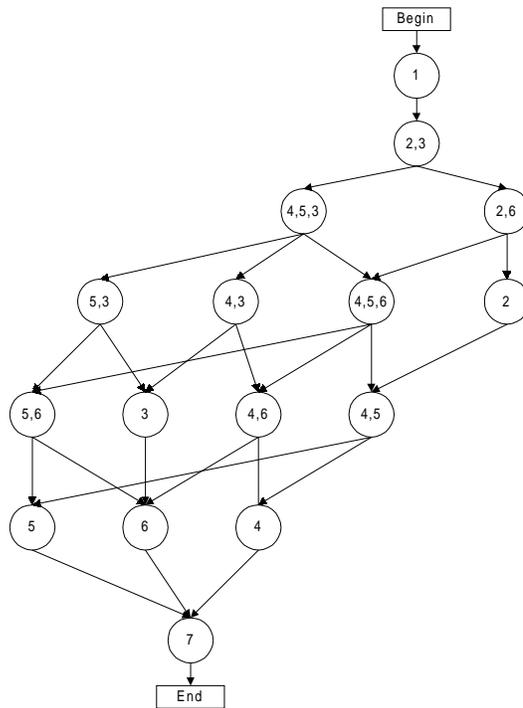


Figure 5 Markov Chain for Task Graph

The primary performance measures of interest are the mean completion time for the overall task system C , the individual task initiation and completion times, I_i and C_i , and the task execution time $E_i = C_i - I_i$. Other performance measures such as device utilization, queue lengths, etc. can be derived from the state probabilities of the Markov chain. The standard method to compute the *mean cycle time* with respect to a reference state R , is [26]:

$$C = \sum_S v_{RS} M(S)$$

where, $M(S)$ is the mean residence time in each state and v_{RS} is the mean number of visits to all other states S . This method requires the solution of a set of linear equations corresponding to the embedded Markov chain. An alternative solution method proposed by Thomasian and Bay takes advantage of the fact that since the task graph is acyclic, the corresponding Markov chain is also acyclic. Therefore, the Markov chain can be generated in a breadth-first or level-by-level manner. The computation of state probabilities at each level is possible by solving the local balance equations with respect to the previous level.

From [26] the completion time of a task is computed by weighting the delays incurred in completing states. Given that R denotes all states preceding R , the *mean path delay* from the initial state to the completion of state R is

$$D(R) = M(R)p(R) + \sum_{S \in R^-} b_R(S)D(S)$$

where $b_R(S)$ is the branching probability and $p(R)$ is the probability of reaching state R from the initial state

$$p(R) = \sum_{S \in R^-} p(S)b_R(S)$$

Unnormalized state probabilities can also be computed level-by-level [19]:

$$P(R) = \sum_{S \in R^-} T_R(S)P(S)/T(R)$$

where R^- is the set of immediate predecessors of R , and $T_R(S)$ is the transition rate from S and R . Given that A_{ij} denotes all states in which T_i is active:

$$E_i = C \sum_{S \in A_{ij}} P(S) = \sum_{S \in A_{ij}} p(S)M(S)$$

With these results, we can derive the execution time for each of the tasks comprising the application task graph. We can determine which tasks are bottlenecks to the computation, consider the addition of extra processing elements, and assess the impact of idle times.

3.2. Analytic Models

Analytic models have been employed to describe the performance of parallel applications executing on shared, heterogeneous networks of workstations [23]. In the case of synchronous iterative (or multiphase) algorithms, a simple, accurate model we can use has the following form for P processors, I iterations, and execution times for the serial, parallel, and parallel processing overhead given:

$$R_p = I * \left(t_{serial} + \frac{\eta t_{par_work}}{P} + t_{par_overhead} \right)$$

The η term is used to represent load imbalance, background load, or the effects of different processor speeds. This analytic performance model yields accurate runtime predictions for a variety of applications executing on various heterogeneous networks of workstations. A generalization of this approach to support any directed, acyclic task graph as above can be easily completed for modeling the iterations of a multiphase or an aperiodic algorithm. Similarly, communications costs can be included as additional task graph vertices, with modeling techniques like those employed in [23].

Note that the queueing and analytic modeling results discussed above, although derived for applications executing on parallel processors, could be used to represent applications executing with additional reconfigurable hardware as well.

3.3. Simulation Models

Although queueing and analytic models provide a powerful mechanism for representing application behavior, the size of the problem or the behaviors of interest may not be easily represented or solved with such models. In such cases, simulation is often used. Although a much broader class of problems can be considered with simulation, the results can be thought of as samples from a random process, so a number of simulation runs are needed to develop confidence in the results. The mathematical form resulting from solving queueing or analytic models enables a designer to assess the sensitivity to a parameter or project the result of a different parameter value. With simulation, this is very difficult to do. Consequently, we do not employ simulation as a performance evaluation tool for programming HPRC resources. The use of simulation is used to functionally verify configurations as well as to validated the modeling techniques in the absence of empirical results.

4. PROGRAMMING ISSUES

Programming of parallel and distributed systems remains challenging but many tools and libraries are available to assist the developer such as PVM [11], MPI [25], BLAS [1], and VSIPL [3]. The HPRC platform presents additional complexities to the programming task in a distributed environment. In addition to partitioning and balancing tasks across computing nodes, at each node hardware/software partitioning decisions are necessary for the RC element. If the RC board is populated with more than one FPGA unit, then decisions about the hardware partitioning across the multiple devices must be made. Similarly, functionality can be temporally partitioned with

multiple, cooperating configurations on the RC elements. Once the partitioning decisions are made, overhead and communication costs must be analyzed. Mapping a given application onto an HPRC platform requires parallel processing, hardware design, and system design capabilities. We discuss parallel and distributed software design environments, hardware configuration design environments, and system level hardware/software co-design environments to support HPRC applications development.

4.1. *Parallel and Distributed Processing Design Environments*

Much research has been invested in studying the performance improvements gained by the parallel execution of software tasks. When applications are partitioned into multiple tasks to potentially be executed on multiple processors or machines, vehicles are needed for determining the optimal partitioning, scheduling and mapping the task sequence, mapping and distributing the data set, and communicating between concurrent tasks. Previous work on developing packages, libraries, and programming environments helped with application portability, reuse, and performance. PVM [11], MPI [25], BLAS [1], and VSIPL [3] are good examples.

The University of Tennessee received a grant from the National Science Foundation to develop the Scalable Intracampus Research Grid (SinRG) [5] to develop the software, networking, and programming infrastructure necessary to support clusters of machines to execute parallel applications with minimal user involvement or intervention, in contrast to most parallel processing environments that force the user to have detailed knowledge of the processing environment. The NetSolve [6] middleware project targets the development of a simple software interface to the parallel processor. Similar work on HARNESS [8] focuses on distributed virtual machines and dynamic reconfiguration issues. By leveraging these efforts to develop common software development environments for parallel applications, we can reduce the difficulties associated with effectively employing parallel processing applications, thus helping in the use of HPRC resources.

4.2. *Hardware Configurations Design Environments*

Reconfigurable computers bring together aspects of both hardware and software systems. Not surprisingly, debate rages about the best design languages, methodologies, and tools for reconfigurable computing systems. Many of the same issues and arguments concerning systems design and hardware/software codesign are applicable.

Most development efforts to map applications onto reconfigurable computers uses VHDL or Verilog for capturing the design, typically at the register transfer level. In doing so, hardware designers can use the same design capture, simulation, and synthesis languages and tools already used for ASIC development. In practice, the productivity from directly using HDLs lags behind industry needs. Designers write much of the HDL code at RTL, and too often do not employ language constructs such as VHDL generics, configurations, and generate statements to create portable, flexible designs. In addition, the synthesis tools provide roughly equivalent capability for FPGAs as with ASICs, enabling the reuse of much of ASIC design flows and tools.

The same domain specific attributes that make hardware description languages effective for designing electronic systems prove to be a significant limitation to the widespread adoption of VHDL or Verilog for capturing designs intended for reconfigurable computers. Software and systems engineers are not familiar with these hardware description languages and resist using them.

At the system design level, a number of proposed extensions to C or C++ have been forwarded by various companies to address behavioral design. Because C/C++ is widely used by systems engineers to develop system prototypes or executable specifications, it is hoped providing a facility to develop hardware designs in some C/C++ dialect will improve productivity and bring systems and hardware engineers closer together. Adoption of a C/C++ dialect potentially will potentially enable a much larger pool of designers to describe hardware because C/C++ users dwarf the HDL user population. The amount of infrastructure required with these C/C++ extensions may approach or even exceed that of using HDLs.

In an attempt to leverage the surging popularity of the Java programming language, as well as its support for code portability and for reuse via object-oriented facilities, researchers at BYU developed JHDL [16]. The JHDL approach exploits the explosion in software development tools for Java and the much larger population of Java programmers to ease in the general adoption of reconfigurable computing. JHDL lowers many of the barriers to entry for potential developers, and significantly simplifies the mapping of functionality between hardware and software. Nonetheless, performance limitations for Java hinder its adoption for high-performance applications.

Researchers at The University of Tennessee developed a Khoros-based design environment that maps “glyphs” representing functional blocks to synthesizable VHDL that results in configuration data targeting different FPGA architectures or multiple FPGA-based reconfigurable computers [4]. Using this system for image processing

applications, a 100X design productivity improvement was demonstrated. Similar research at Northwestern University addresses design in MATLAB for reconfigurable computers [7].

4.3. System Design Environments

For HPRC systems, the task of hardware/software partitioning is complicated because the hardware and software are interdependent, making it difficult to make a decision about one without affecting the other. Hardware-software partitioning algorithms attempt to meet the design constraints (performance, cost, etc.) by deciding which operations will be implemented in software (CPU) and which in special-purpose hardware, in our case, reconfigurable hardware. There are two general styles for co-design partitioning: *hardware-oriented* and *software-oriented* [18, 28]. Hardware-oriented algorithms start with everything in hardware and move some of the operations to software until the performance goals are met. Software-oriented algorithms start with all operations in software and move selected ones to hardware.

The “cost of the system” (design cost, procurement cost, efficiency, performance, etc.), is affected by all phases of the co-design process. The partitioning of processes between hardware and software affects the implementation and performance cost of the entire system. Less than optimal partitioning, either at the high-level software application tasks or the hardware/software division at each node, may cause excessive interprocess communication. Inefficiencies can delay computation at one or more nodes while other processors sit idle. High performance interconnection networks and customizable interconnects among RC elements can lessen the impact but optimal partitioning resulting in a balanced system will have the greatest impact on overall efficiency.

The use of object-oriented programming languages such as C++ to describe a system’s functionality in terms of communicating objects naturally supports coarse and fine-grained parallelism. Wolf [29] conducted research on the co-synthesis of embedded systems to partition, schedule, and map the application software and synthesize the appropriate hardware. The algorithm Wolf presents does not partition tasks into smaller sections of code, but it does split the variable set of an object across several processing elements. The algorithm is designed to over-allocate hardware to meet rate requirements, then iteratively reduce the system cost function by moving tasks and data to new processing elements [29]. These techniques in conjunction with the tools and libraries already developed for distributed systems can be leveraged in the development of a programming paradigm for the HPRC platform.

Another research effort at the Institute of Information Science in Taiwan has produced a method for programming general-purpose parallel systems. CMAPS, a system-level co-synthesis methodology for general-purpose parallel systems, targets a general parallel system [15]. By interleaving the modeling and synthesis phases, the CMAPS tool is able to explore the interaction between hardware and software. The CMAPS tool is used to define specifications with a *Problem Graph* using elementary problems from a *Problem Database*. The CMAPS tool then maps the graph into an initial solution, which then can be transformed into hardware and software models. These models are analyzed and inferior mappings are eliminated to decrease the complexity of synthesis and produce the best solution. After the scheduling algorithm is chosen, a co-simulation of hardware and software is performed to confirm results.

5. CONCLUSIONS

High Performance Reconfigurable Computing promises to cost-effectively leverage the benefits of both high performance computing and reconfigurable computing. The potential impact of high performance reconfigurable computing cannot be overstated; embedded and computer systems will never be the same. In the future, reconfigurable computing will see the widespread adoption of the co-processing model in general computer platforms. The tremendous growth in gate capacity will make reconfigurable processing units attractive additions to support platform-based design and product customization.

Future generations of processors will include reconfigurable logic units as functional units in processors as well. Transmeta’s Crusoe processor support for the emulation of other instruction set architectures introduced the notion of morphable computing to many; this trend will continue by enabling dynamic instruction set architecture computers [27].

The performance evaluation techniques presented here will help designers to effectively exploit the potential benefits of HPRC platforms. Ongoing research will validate these approaches for reconfigurable computing applications, thus enabling performance predictability for applications on HPRC platforms. The design environments for the software, hardware configurations, and overall system remain as a critical enabling technology to help make HPRC applications economically feasible.

ACKNOWLEDGEMENTS

This work was partially supported by the Air Force Research Lab (AFRL/IFTC) via contract F30602-99-D-0221 with CACI International, Inc. The authors thank Steve Drager for his inputs and support of this research.

REFERENCES

- [1] BLAS: Basic Library of Algebraic Subroutines. <http://www.netlib.org/blas/index.html> . 2001.
- [2] DARPA Adaptive Computing Systems. <http://www.darpa.mil/ito/research/acs/projects.html> . 2001.
- [3] Vector Signal Image Processing Library (VSIPL). <http://www.vsipl.org> . 2001.
- [4] See <http://microsys6.engr.utk.edu/~bouldin/darpa/>
- [5] See <http://www.cs.utk.edu/sinrg/>
- [6] Arnold, Dorian C. and Dongarra, Jack. "The NetSolve Environment: Progressing Towards the Seamless Grid," 2000 *International Conference on Parallel Processing (ICPP-2000)*, Toronto Canada, August 21-24, 2000
- [7] Banerjee, Prith. et al. "A MATLAB Compiler for Distributed Heterogeneous Reconfigurable Computing Systems," *Int. Symp. on FPGA Custom Computing Machines (FCCM-2000)* Napa Valley, CA, Apr. 2000.
- [8] Beck, Micah et al. HARNESS: A Next Generation Distributed Virtual Machine, *International Journal on Future Generation Computer Systems*, Elsevier Publ., Volume 15, Number 5/6, 1999.
- [9] Compton, K., Cooley, J., Knol, S., and Hauck, S. Configuration Relocation and Defragmentation for FPGAs. IEEE Symposium on Field-Programmable Custom Computing Machines . 2000.
- [10] Compton, K. and Hauck, S. Configurable Computing: A Survey of Systems and Software. Northwestern University, Dept. of ECE Technical Report. 1999. Northwestern University.
- [11] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sundareem, V., *PVM: A User's Guide and Tutorial for Networked Parallel Computing* MIT Press, 1994.
- [12] Hauck, S., "The Roles of FPGAs in Reprogrammable Systems," *Proceedings of the IEEE*, vol. 86, no. 4, pp. 615-638, Apr.1998.
- [13] Hauck, S. The Future of Reconfigurable Systems. Keynote Address, 5th Canadian Conference on Field Programmable Devices . 1998. Montreal.
- [14] Heidelberger, P. and Trivedi, K. S., "Analytic Queueing Models for Programs with Internal Concurrency," *IEEE Transactions on Computers*, vol. C-32, no. 1, pp. 73-82, Jan.1983.
- [15] Hsiung, P.-A., "CMAPS: A Cosynthesis Methodology for Application-Oriented Parallel Systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 1, pp. 51-81, Jan.2000.
- [16] JHDL. Java Hardware Description Language. <http://www.jhdl.org>
- [17] Levine, Ben, "A Systematic Implementation of Image Processing Algorithms on Configurable Computing Hardware." Master of Science Electrical Engineering, The University of Tennessee, 1999.
- [18] Li, Y., Callahan, T., Darnell, E., Harr, R., Kurkure, U., and Stockwood, J. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. Design Automation Conference DAC 2000 , 507-512. 2000. Los Angeles, California.
- [19] Li, Z., Compton, K., and Hauck, S. Configuration Caching Techniques for FPGA. IEEE Symposium on Field-Programmable Custom Computing Machines . 2000.
- [20] Natarajan, Senthil, "Development and Verification of Library Cells for Reconfigurable Logic." Master of Science Electrical Engineering, The University of Tennessee, 1999.
- [21] Natarajan, S., Levine, B., Tan, C., Newport, D., and Bouldin, D. Automatic Mapping of Khoros-Based Applications to Adaptive Computing Systems. MAPLD-99 . 1999. Laurel, MD.
- [22] Ong, S.-W., Kerkiz, N., Srijanto, B., Tan, C., Langston, M., Newport, D., and Bouldin, D. Design Flow for Automatic mapping of Graphical Programming Applications to Adaptive Computing Systems. unknown . 2000.
- [23] Gregory D. Peterson and Roger D. Chamberlain, "Parallel Application Performance in a Shared Resource Environment." *IEEE Distributed Systems Engineering Journal*, 3(1):9-19, March 1996.
- [24] Shettlers, Carl Wayne, "Scheduling Task Chains on an Array of Reconfigurable FPGAs." Master of Science University of Tennessee, 1999.
- [25] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J., *MPI: The Complete Reference*, 2nd ed. MIT Press, 1998.
- [26] Thomasian, A. and Bay, P. F., "Analytic Queueing Network Models for Parallel Processing of Task Systems," *IEEE Transactions on Computers*, vol. C-35, no. 12, pp. 1045-1054, Dec.1986.
- [27] M.J. Wirthlin and B.L. Hutchings, "A Dynamic Instruction Set Computer." In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, April, 1995.
- [28] Wolf, W., "Hardware-Software Codesign of Embedded Systems," *Proceeding of IEEE*, vol. 82, no. 7, pp. 967-989, July1994.
- [29] Wolf, W., "Object-Oriented Cosynthesis of Distributed Embedded Systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 3, pp. 301-314, July1996.