AFRL-RI-RS-TR-2009-181
Final Technical Report
July 2009

ADVANCED COMPUTING ARCHITECTURES FOR COGNITIVE PROCESSING

University of Tennessee

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-181 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
LOK YAN	EDWARD J. JONES, Deputy Chief
Work Unit Manager	Advanced Computing Division
	Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)* JULY 2009	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* June 2006 – December 2008

4. TITLE AND SUBTITLE ADVANCED COMPUTING ARCHITECTURES FOR COGNITIVE PROCESSING	**5a. CONTRACT NUMBER** N/A
	5b. GRANT NUMBER FA8750-06-1-0185
	5c. PROGRAM ELEMENT NUMBER 62702F
6. AUTHOR(S) Gregory D. Peterson	**5d. PROJECT NUMBER** 459T
	5e. TASK NUMBER AC
	5f. WORK UNIT NUMBER CP
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Tennessee, 1331 Circle Park Drive, Knoxville, TN 37996-0003	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RITB, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** N/A
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2009-181

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA#: 88ABW-2009-3306 Date Cleared: 20-July-2009

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the research results of the Advanced Computing Architectures for Cognitive Processing project. Cognitive processing algorithms promise to transform Air Force mission capabilities if practical, deployable systems can be created. Advanced computing architectures based on approaches such as polygranular parallel processing (e.g., high performance reconfigurable computing), data intensive systems (e.g., processors in memory or intelligent RAM), configurable/morphable processors, DNA computing, and quantum computing could make embedded cognitive processing feasible. Given the diverse new computational technologies that are now emerging, this report presents an assessment of the effectiveness of various types of computational architectures for cognitive processing for Air Force systems acquisition.

15. SUBJECT TERMS
Cognitive Processing, Cognitive Algorithms, Polygranular, Reconfigurable Computing, Sandboxing, Heterogenous Systems

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Lok K. Yan
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	123	**19b. TELEPHONE NUMBER** *(Include area code)* N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

We would like to thank the Air Force Research Laboratory for supporting this research. In particular, thanks to Lok-Kwong Yan and Steve Drager for their support, input, ideas, and feedback.

A number of graduate students are working on projects discussed in this report and helped with the results reported. They include Saumil Merchant, Brian Sharp, Rick Weber, Akila Gothandaraman, Junqing Sun, Chris Rathgeb, Akan Udoeyop, and JunKyu Lee. In particular, this project helped support Brian and Chris in their studies.

Additional support for this work came from The University of Tennessee's Science Alliance and the National Science Foundation. We thank them as well for their support of our research.

1.0 INTRODUCTION

Although increasingly powerful information systems are revolutionizing defense capabilities, the nation faces difficult challenges with the amalgamation of systems of systems within dynamic networking environments using multiple protocols. This problem is further complicated as the constituent systems simultaneously increase in complexity. The emerging defense operational C4ISR (Command, Control, Communications, Computers, Intelligence, Surveillance and reconnaissance) environment requires support for capabilities such as real-time sensor-to-shooter targeting support, rapid bomb damage assessment, UCAV (Unmanned Combat Aerial Vehicle) operation, and air tasking order generation and distribution. Support for joint forces including Army, Navy, Air Force, and allied units presents a significant additional challenge to the effective deployment, operation, support, and protection of the C4ISR infrastructure required to manage modern battlefields. The Joint Battlespace Infosphere (JBI) is a combat information management system under development by the Air Force that combines C4ISR inputs from a variety of source, including existing systems such as JTIDS (Joint Tactical Information Distibution System) or AWACS (Airborne Warning and Control System) and emerging systems such as UAVs (Unmanned Aerial Vehicle)/UCAVs and the Joint Tactical Radio System. From this collection of systems and their inputs, the JBI builds an aggregated picture to provide situational awareness for users from joint task force commanders down to individual soldiers or pilots in the field. The vast collection of sensors provides tremendous amounts of raw data that must be quickly processed to extract timely information. Given the critical role of the JBI, evaluation of the most effective computer architecture approaches to support the associated cognitive processing applications is needed to ensure the JBI and similar systems can effectively provide mission-critical capabilities such as situational awareness.

Cognitive processing algorithms promise to transform Air Force mission capabilities if practical, deployable systems can be created. Advanced computing architectures based on approaches such as polygranular parallel processing (e.g., high performance reconfigurable computing), data intensive systems (e.g., processors in memory or intelligent RAM (Random Access Memory), configurable/morphable processors, DNA (Deoxyribonucleic Acid) computing, and quantum computing could make embedded cognitive processing feasible. Given the diverse new computational technologies that are now emerging, an assessment of their effectiveness for various types of cognitive processing would be helpful for Air Force systems acquisition.

Specific cognitive processing approaches addressed herein include global information grid approaches such as the JBI and embedded real-time, adaptive neural network structures. A host of architectural options exist that can be employed for cognitive architectures, including vector supercomputers, massively parallel processors, symmetric multiprocessors with shared memory, systolic arrays, dataflow architectures, configurable instruction set architectures, processors in memory, reconfigurable computing, general purpose programming with graphical processing units, multicore architectures, grid computing, quantum computing, and DNA computing. When one considers the execution of cognitive processing applications on these architectures, questions of performance, scalability, reliability, and deployability arise. The computing architecture space will be evaluated for each of these cognitive systems with respect to the computational model and suitability for implementing cognitive processing algorithms, the requirements for architectures to meet the computational, memory, I/O (Input/Output), storage, and communications demands of cognitive algorithms, and additional system constraints such as size, weight, cost, and power dissipation. In addition, considerations impacting practical systems development are surveyed with a particular emphasis on topics such as runtime systems support (e.g., operating systems, libraries, middleware, and compilers), performance evaluation, application development tools, and support infrastructure. Prior work with analytical performance modeling of parallel systems is exploited and extended as appropriate for these architectures and applications. In so doing, we seek to develop an analytical framework with which to evaluate the most effective approaches under various operational regimes and based on technology trends. For example, as a result of Moore's Law and Pollack's Rule, vendors of commodity microprocessors are moving towards multicore implementations. At the same time, trends for clock speed, interconnect technologies, and memory cycle times indicate that memory latency issues render ineffective serial architectures and prior commodity-based supercomputer architectures.

Symmetric multiprocessors include a number of processors that share a common global memory. The data caches for the processors must be managed so that coherence and consistency is maintained, which results in caching protocols that limit system scalability. In practice, emerging computer architectures are likely to employ small numbers of shared memory processors to comprise individual nodes, with a global distributed memory model based on message passing between nodes. Such an approach leverages the multicore technology trends while enabling scalable systems. To the extent that cognitive processing applications include significant locality (both temporal and spatial) for modest numbers of concurrent threads of execution, symmetric multiprocessing nodes are likely to be widely deployed in future cognitive architectures.

True system scalability and flexibility will require distributed memory and message passing. In particular, the JBI will consist of a host of heterogeneous computational nodes that will be interconnected by various communications links with disparate bandwidth, latency, and reliability characteristics. Technologies associated with massively parallel processors and grid technologies can potentially be employed for cognitive processing applications. The publish/subscribe model for distributed sensors making data available in a controlled manner to various users will be a distributed system of cooperating computational nodes. Technology trends for these types of platforms will be particularly important to understand in order to predict future needs and capabilities.

Traditional vector supercomputing platforms are not likely to be as useful for cognitive processing applications within the JBI framework using current algorithms. Although these systems employ vectors as an effective means to mask memory access latency, the floating point focus of these platforms does not match well with most cognitive processing.

A reconfigurable computing (RC) system offers a revolutionary combination of the performance of custom hardware and the flexibility of software by employing programmable logic technology to create customized application accelerators. The key feature of an RC is the reconfigurable processing element (PE) which, in the current generation, is a FPGA (Field Programmable Gate Array) chip. Flexibility, improved performance, and cost effectiveness are opening up new avenues for FPGAs in the area of reconfigurable computing. Reconfigurable computing is often achieved via the coupling of FPGA units as configurable co-processors or attached units to general-purpose processors. Many of today's computationally intensive applications can benefit from the speed offered by application specific hardware co-processors, but for applications with multiple specialized needs it is not feasible to have a different co-processor for every specialized function in order to achieve the maximum performance. Such diverse applications stand to benefit the most from the reconfigurability of RC architectures. One RC unit potentially can replace several ASIC (Application Specific Integrated Circuit) co-processors and by reconfiguring the FPGA(s) during run-time, one can achieve superior performance over that of a software only solution. Alternatively, High Performance Computers (HPC) provide dramatically improved capabilities for a number of coarse granularity parallel applications, but often are quite expensive to acquire and program. To address the high hardware costs, one may create more inexpensive "Beowolf" clusters of dedicated commodity processors. With the emergence of grid computing, vast numbers of processors become available for parallel and distributed applications at virtually no cost.

One can accelerate cognitive processing applications by exploiting the potential parallelism of an algorithm subject to the constraints of the cost of communications and the granularity of the PEs. In practice, cognitive processing applications often contain tasks with a heterogeneous mix of potential parallelism that, to fully exploit the potential parallelism, must be addressed using PEs with different computational granularity. For example, for tasks containing coarse-grained parallelism, MIMD (Multiple Instructions Multiple Data) architectures can achieve significant speedups. Some fine-grained parallelism can be exploited using the instruction level parallelism (ILP) found within modern superscalar, superpipelined processors. In both these cases, the statically defined processor can exploit a specific granularity of parallelism. In contrast, reconfigurable computing elements also exploit fine-grained, bit-level parallelism, although multiple RC engines can be collected together to support coarser-grained parallelism as well. With high performance reconfigurable computing (HPRC), the combination of RC and microprocessor computational elements provides the infrastructure to effectively accelerate *polygranular parallel processing*. This project explored the polygranular nature of cognitive processing applications and their best mapping to various types of processing elements.

The author is also collaborating with computational scientists to perform research into next-generation computing approaches and architectures for scientific computing. The effort described here includes a similar approach, but with a focus on embedded systems and cognitive applications of interest to the military. Some comparison between these application domains and computer architectural issues that arise will also be discussed.

In the next chapter we give a brief overview of current and emerging architectures of interest for cognitive processing applications. We then explore a set of application areas related to cognitive processing and their appropriateness for various architectures. Next we explore CAD (Computer Aided Design) and language issues for the computing architectures. Then we develop a performance modeling framework to help understand the performance of applications and architectures and then explore ways to use it for optimizing the architecture or mapping thereon. Finally, we draw conclusions and suggest future directions for research.

2.0 COMPUTING ARCHITECTURES

Computing architectures have been a topic of significant research for decades, and the need for architectures that are effective for cognitive processing applications has helped impact these architectures. We explore various types of architectures in this chapter to appreciate the diversity of emerging platforms that can be targeted for cognitive processing applications. DARPA has funded a number of efforts related to cognitive processing architectures, with the Strategic Computing, Adaptive Computing Systems, Rapid Prototyping of Application Specific Signal Processing Systems, Data Intensive Systems, Cognitive Architectures, Polymorphic Computing Architectures, and High Productivity Computing Systems (HPCS) programs all addressing this area.

Cognitive processing and computational science applications feature voracious appetites for processing power. To meet this demand, next-generation supercomputers have now achieved petascale capabilities and discussions have already begun on approaches to field exascale supercomputers. Although the demand for processing power continues to explode, the performance of individual microprocessors has largely stalled due to issues such as delays and power dissipation. Because of this current state of affairs, the computing industry is now at an inflection point as a set of possible emerging computational approaches vie for prominence in future platforms. In particular, multicore processors, general purpose processing with graphical processing units, reconfigurable computing with programmable logic devices, and accelerators built with custom logic circuits have achieved promising results and are areas of active research programs and commercial development. Other technologies such as intelligent RAM, intelligent disk, quantum computing, and DNA/biomolecular computing show promise as well, but are not as mature or popular, so we will not address them herein. The primary focus of this report is on exploring the emerging computational platforms to evaluate their effectiveness and appropriateness for creating next-generation cognitive processing applications. We first briefly discuss each of these technologies and then explore how they may impact next-generation computational platforms.

Traditional serial processors have evolved over the past 75 years from huge systems based on vacuum tubes, to systems built from discrete transistors, to small, medium, and large scale integration to create processors composed of dozens of integrated circuits, to microprocessors implemented on a single integrated circuit using very large scale integration (VLSI). Starting with the Intel 4004 microprocessor, the typical computer now is commonly implemented with a central processing unit with attached main memory, I/O devices, and storage (e.g., cards, tapes, floppy and hard disks, optical disks, FLASH disks). The processor is either a complex instruction set computer with an instruction set architecture that supports complicated instructions with varied addressing modes or a reduced instruction set computer with an instruction set architecture featuring simpler addressing modes and targeted for faster, typically pipelined implementation. The processors have evolved to include pipelining, hazard detection and mitigation (e.g., forwarding, branch delay slots, branch prediction units), superscalar execution to allow issuance of multiple instructions per cycle, out of order execution and/or completion, speculative processing, and symmetric multithreading or hyperthreading. The memory systems have evolved to include caching of up to several levels with different associativity, split or unified construction, various write back or write through protocols, and cache consistency and coherence for shared memory processing as well as support for memory segments, pages, virtual memory, shared memory, DMA (Direct Memory Addressing), and support for different memory affinities. Multimedia extensions such as MMX (Multi-Media eXtension, Matrix Math eXtension), SSE (Streamind SMID Extensions), and Altivec provide vector units for accelerating operations on arrays of data, particularly for graphics and audio applications. For all the dizzying variety of processor enhancements and optimizations, processors now face practical performance limits due to power dissipation issues leading to clock speed constraints and a dearth of new microarchitectural enhancements to maintain performance growth. Consequently, processors are now shifting to include multiple cores per die. See Hennessy and Patterson's excellent text for an overview of the field [1].

Virtually every vendor of microprocessors now ships multicore processors. This approach takes advantage of the additional transistors posited by Moore's Law [2], which is especially useful at a time when architectural advances to improve the performance of each core have largely stalled. Hence, quad-core processors are common and larger numbers of cores are now becoming available. IBM has the Power and Cell/BE architectures for homogeneous and heterogeneous multicore processing. Similarly, Intel has the Xeon, Core, Nehelem, Itanium, and Larrabee multicore processors, AMD has the Opteron multicore processors (e.g., Barcelona and Shanghai), and Sun has the SPARC/Rock/Niagara multicore processors. The number of cores is generally expected to double every eighteen months for the foreseeable future. With current multicore processors, the porting and tuning of applications is now quite similar to the adoption of shared memory symmetric multiprocessors. The memory hierarchy performance and ability to bring data onto the device looms as the primary issue for today's modest number of cores, but will likely become a critical issue as the number of cores scales significantly. Because of the memory hierarchy and bandwidth limitations of the die, many question the ability to scale cores beyond 8 while sustaining performance. At the same time, applications must be recoded to exploit the additional cores. For example, see [3] for an overview of programming Intel's multicore processors.

A number of larger high performance computing systems are available for supercomputing applications. Dating back to the DARPA's Strategic Computing and DOE's ASCI programs, a variety of parallel supercomputers have become popular. Since the early 1990s, nearly all supercomputers have shifted from custom processors (e.g., Cray vector units, NCUBE and nMOS Transputer custom processors) to commodity microprocessors. Modern supercomputers now reach petascale size and performance with tens of thousands of sockets populated with multicore microprocessors. For example, the UT (University of Tennessee) and ORNL (Oak Ridge National Laboratory) Cray XT5s, Kraken and Jaguar, are based on AMD multicore microprocessors. The IBM BlueGene/L and /P are based on multicore PowerPC microprocessors. Other large supercomputers using Sun and other microprocessors are also available. The current fastest computer in the world, RoadRunner, is comprised of x86 processors augmented with IBM Cell/BE multicore processors. Although Cray and other manufactures include support for accelerators such as vector units, GPUs (Graphics Processing Unit), and FPGAs, none of these accelerators is widely adopted on full-scale computer systems as of yet. However, these technologies are demonstrating enough performance improvement over commodity microprocessors, that the supercomputer vendors are all actively exploring the insertion of these technologies into their platforms.

Graphics processors achieve impressive speedups for applications that can exploit their pipelined processing units while fitting into their on-board memory. These devices prove quite effective for streaming applications with limited conditionals, but current GPUs are largely limited to single-precision floating point. With double-precision floating point support having recently become available, GPGPUs (General Purpose Graphics Processing Unit) seem likely to become quite popular for accelerating computational science applications. The primary vendors are AMD/ATI and NVIDIA, although Intel is addressing this architectural space with their Larrabee project. Although a wide range of papers have reported speedups of 200-500x, one might reasonably expect speedups on the order of 50-100x for suitable single precision applications when compared to tuned serial codes. Similarly, codes with double precision floating point computation achieve speedups of 10-20x.

Reconfigurable computing (RC) platforms typically include devices such as field programmable gate arrays (FPGAs) that enable the creating of optimized circuits on demand to accelerate applications [4]. RC platforms enable one to exploit fine-grained, bit-level parallelism through pipelining and replicated functional units. The data precision, movement, and storage are all controlled by the application developer. A similar approach is to develop customized circuits for specific processing tasks such as image processing for automatic target recognition, signal processing with customized datapath elements, encryption circuits optimized for specific keys, string matching circuits for publish/subscribe computations or bioinformatics, computing forces for molecular dynamics simulations, or to perform floating point operations for linear algebra.

Reconfigurable computing systems first became available in the early 1990s as FPGAs shifted to programmability with SRAM (Static Random Access Memory) bits (as opposed to antifuse-based) and they became large enough to support significant designs. Early RC systems included several FPGAs on a board connected via PCI (Peripheral Component Interconnect) or VME to a host processor that controlled configuration and reconfiguration of the FPGAs, data movement, and initiation and termination of FPGA processing. By the end of the 1990s and into the early 2000s, FPGAs were becoming large enough and fast enough that significant processing capabilities were demonstrated, particularly in the areas of signal and image processing and encryption. By the mid 2000s, FPGA devices each supporting millions of gate equivalents were available and RC systems with more powerful development and runtime systems were becoming available. The interconnection between FPGA and host processor was improving, with PCI express becoming popular as an interface. In the late 2000s, the Hypertransport (HT) interconnect for AMD processors was opened to support accelerators such as FPGAs and vendors (DRC [5] and Xtreme Data [6]) made processor socket compatible FPGA systems that provided HT high-speed communications to memory and other devices. Intel similarly opened its Accelerator Applications Layer and QuickPath Interconnect to allow FPGA boards to be inserted into Intel processor sockets for similar benefits. These new systems provide much higher bandwidth for sustained high performance processing. At the same time, the devices now are large enough and fast enough to provide floating point support in addition to the fixed point support provided in earlier generations. As processor technologies have stagnated due to power constraints and a lack of microarchitectural enhancements, FPGAs continue to grow with Moore’s Law, increase clock rates, and provide built-in functions for high speed communications, memory, and arithmetic units.

Reconfigurable computing not only enables flexibility in mapping problems onto the hardware; one can consider using machine intelligence approaches to optimize the hardware implementation in real time. The insight of repeated programming of FPGAs under the control of an evolutionary algorithm was first introduced by deGaris [7] back in 1992. The idea is to encode the bit streams that configure the FPGAs as genetically mutable chromosomes and evolve a population of such chromosomes over hundreds of generations using genetic operators such as selection, crossover, and mutation to find the fittest circuit design for particular circuit functionality. The fitness function is the performance quality of the function towards which the chromosomes are being evolved. This gave birth to the field of evolvable hardware (EHW) systems. deGaris classified these systems into extrinsic EHW systems and intrinsic EHW systems. Extrinsic evolution is the offline evolution where the evolutionary algorithm is wrapped around a software model of the hardware system, whereas the intrinsic evolution employs an online evolution paradigm where an evolutionary algorithm is directly changing the hardware. deGaris later used EHW principles he developed to evolve 3D cellular automata (CA) based neural network modules directly in Xilinx's XC6264 FPGAs in special hardware called a CAMBrain Machine (CBM) [8]. Although deGaris introduced and classified EHW, Thomson illustrated its promise by developing the first intrinsic evolvable hardware system design [9]. He used a Xilinx XC6216 chip to distinguish between two square wave inputs of 1 and 10 kHz. The circuit was evolved intrinsically so that the output would be 0 volt for the 1 KHz input, and 5 volts for the 10 KHz input. Early researchers concentrated on evolving bit streams, in effect gate level evolution of a digital circuit. Researchers such as Higuchi used the concept of evolvable hardware to do functional level hardware evolution instead of evolving logic gates [10]. He evolved connections between functional modules such as adders, multipliers, sine, cosine etc. One of the rationales for functional level evolution was that a gate level bit stream encoded as a chromosome is very long, making the evolution process very time consuming. But it limits the circuit functionality that can be targeted for evolution to circuits that can be constructed with the functional blocks already available at the time of evolution.

Reconfigurable systems comprised of a collection of hard or soft processor cores, reconfigurable logic (e.g. FPGAs), memory, and interconnect are now widely available. The management of the resources for high performance reconfigurable computing (HPRC) systems remains an important gap in capability: current systems have very primitive infrastructures to support device scheduling, sharing (spatially and temporally), library support, debugging and development support, and to provide fault tolerance. Moreover, no real notion of security models is included with these HPRC systems. In particular, we need a runtime environment for HPRC systems to address the missing capabilities with respect to supporting multilevel security, sharing the system among multiple processes that may use the reconfigurable hardware, and providing reliable constructs for assuring confidentiality, integrity and availability.

As part of the DARPA Polymorphic Computing Architectures program, new architectures such as TRIPS (Tera-op Reliable Intelligently advanced Processing System) from UT Austin and Monarch from Raytheon were developed. Other work included smart memories research at Stanford and RAW at MIT. These projects addressed ways to reconfigure processing elements and caches/memories to provide faster, customized processing. These architectures appear promising, but aren’t commercially available or economically viable yet.

There are a host of other architectural approaches as well. For example, transactional memories provide a beneficial semantic for large systems, particularly when fault tolerance issues are considered. Intelligent RAM and Intelligent Disk (and variations on both) have shown promise for some application areas as well. Exotic technologies such as quantum computing and DNA/molecular computing show promise, but are years away from being ready for practical deployment. Application specific processors and accelerators, such as MD-GRAPE for molecular dynamics or Clearspeed [148] devices for accelerating floating point computations also show promise, but with much smaller installed bases.

For each of these types of platforms, programming represents a major challenge. The best languages, compilers, runtime systems, and even computational model for each of these emerging platforms remain as open problems to be solved.

For all of these types of devices, a range of system-level issues remain to be addressed. For example, to solve a given problem, what is the best type of processing platform to use? Should it contain multicore processors, GPUs, and FPGAs? If so, of what type and in what number? How should the memory be organized? What I/O capabilities should be supported?

In order to address these questions, we have been evaluating a set of cognitive processing applications relevant to USAF as well as computational science applications from chemistry, biology, and physics. We seek to define a useful set of metrics or attributes of interest in assessing the appropriateness of the various emerging computational platforms for the computational science applications. To understand the tradeoffs in this large, complex architectural space, we propose to create a performance modeling framework to describe the applications and platforms of interest. This framework can then be used to understand the behavior of systems, including the location and cause of performance bottlenecks or load imbalances. We can then employ optimization techniques to explore the architectural space, scheduling policies, or programming strategies to most effectively exploit the polygranular parallelism present in large-scale computational science applications. Before we address the programming infrastructure, performance modeling, or optimization, we next turn to an overview of applications relevant to this study.

3.0 APPLICATIONS

In this chapter, we explore a set of representative cognitive processing applications and their mapping onto various computer architectures. In so doing, we gain insight into the most appropriate architectural approaches for specific types of cognitive processing applications as well as how to best map these applications onto the hardware resources. We first consider neural networks and artificial brains as exemplars of common machine learning techniques. Next, we evaluate distributed publication/subscription databases such as that of the JBI. We then explore text and image processing applications that are necessary for situational awareness and sensor to shooter capabilities. Finally, we look at a set of scientific computing codes to draw comparisons to the other application types and highlight the different architectural approaches needed to best address cognitive processing application needs.

3.1 MACHINE LEARNING WITH ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks have gained a lot of popularity in the computational intelligence and machine learning community over the last couple of decades. They are networks of fully or partially interconnected information processing elements called artificial neurons. Artificial neurons are simulations of their biological counterparts, typically producing an output from summation of multiple weighted inputs and a bias. The output is passed through a nonlinear monotonically increasing activation function also called a transfer function. Various network topologies proposed for the artificial neural networks can be broadly classified into recurrent and non-recurrent networks. Recurrent networks have feedback connections from outputs back to input nodes or to one of the hidden layers. Non-recurrent networks are feed-forward networks such as the popular multilayer perceptron model. These networks are exposed to a training dataset from which they extract information and learn over time some particular characteristic of the input data. The learning algorithms are classified into supervised and unsupervised training algorithms. Under supervised training the input data used to train the network have corresponding target output vectors that are typically used to calculate the mean squared error between the network output and target output. This error is used to guide the search in the weight space to optimize the network. It is a gradient descent search algorithm, popularly known as the back-propagation algorithm, which tries to minimize the total mean squared error between network and target output. These networks can effectively model complex nonlinear relationships between inputs and outputs. They are widely used in pattern classification, sequence recognition, function approximation, and time series prediction, as well as in data processing systems for filtering, clustering, and compression applications. There have been many successful artificial neural network implementations in real world scenarios ranging from military applications, medical diagnosis, autonomously flying aircrafts, and credit card fraud detection systems.

3.1.1 REVIEW OF ANN FPGA IMPLEMENTATIONS

FPGAs offer a custom hardware solution with the flexibility of runtime reconfigurations to change the circuit functionality after fabrication. This feature makes them attractive for low volume custom ANN (Artificial Neural Network) implementations and many have been reported in the literature. Zhu and Sutton present a good survey of FPGA implementations of artificial neural networks [11]. One of the earliest reported FPGA implementations was the Ganglion connectionist classifier [11]. The implementation used FPGAs to achieve higher connection processing speeds relying on runtime reconfigurations for each application of the classifier. The neural network training was performed in microprocessor-based offline software simulations. [12,13,14] are some other implementations relying on reconfigurations for different neural network applications. Implementations of the back-propagation algorithm on FPGAs for accelerating training of ANNs have also been reported in the literature, but have relied on runtime reconfigurations due to smaller FPGA capacities for different sequential stages of the algorithm, making them unsuitable for online training [15-16]. To be area efficient and fit larger networks on a single FPGA, some approaches have used bit stream and other encoding techniques for representing real values. This approach can replace larger multipliers by simple logic gates such as bitwise XOR and demonstrate area efficiency and higher processing capacity [14,17]. Others have used a vector-based data parallel approach to represent real values and compute the sum of products [12,13]. Noory and Groza [18] demonstrate a distributed arithmetic (DA) approach for their implementation.

This section addresses intrinsic artificial neural network training and evolution on FPGAs. It presents an intrinsic evolvable hardware design of a class of artificial neural networks called block-based neural networks trained using genetic algorithms. The presented design addresses some of the implementation issues with intrinsic hardware designs of artificial neural networks.

3.1.2 GENETIC EVOLUTION OF ARTIFICIAL NEURAL NETWORKS

The back-propagation algorithm, being a gradient descent approach, has two drawbacks as outlined by Sutton [19]. The search often gets trapped in local minima if the gradient step is small and could have an oscillatory behavior for large gradient steps. The method is inefficient in searching for global minima, especially with multimodal and non-differentiable search surfaces. Also, there is a problem of catastrophic interference with these methods. There is a high level of interference between learning with different patterns, because those units that have so far been found most useful are also the ones likely to get changed to handle new patterns. The problem of global minima can be solved using global search procedures such as genetic algorithms. Many researchers have proposed using genetic algorithms to evolve neural networks to find optimized candidates in the large deceptive multimodal search space [20,21,22,23,24,25,26,27,28]. Genetic evolution works on a population of neural networks, each encoded as a chromosome. These chromosomes are genetically evolved to survive the fittest individual. The fitness function is defined using the aforementioned total mean square error. The selective pressure is against the least fit individuals, thus surviving and selecting the fittest for genetic crossover and mutation to produce newer generations. The population of chromosomes is evolved over multiple generations until an individual is found with fitness equal to or greater than the target fitness.

GA (genetic Algorithm) being a global approach avoids the pit falls of local minima faced in gradient descent algorithms. It does not need to calculate derivatives of the error function and hence works very well with non-differentiable error surfaces. Also there are no restrictions on network topologies as long as a good fitness function can be defined for the network. Thus it can handle a wide variety of artificial neural networks. But, the evolutionary algorithms are computationally intensive and can be slower compared to gradient descent based training such as back-propagation algorithm. If the initial guess in gradient descent algorithms gives an error that is closer in proximity to the global minima on the error surface, these algorithms can converge faster than a global sampling technique such as genetic evolution. But, if the neural network is more complex with multiple hidden neural layers, the error surface will be complex with many discontinuities. In such cases gradient descent search algorithms may get stuck in local minima or not converge at all, whereas, the global search techniques such as GA are more likely to find an answer [29]. Genetic evolution being an adaptive process is good at global sampling, but performs poorly for local fine tuning.

3.1.3 HARDWARE IMPLEMENTATIONS OF ANNS

Due to wide ranging applications of these networks and inherent parallelism in the network structure, there has been a lot of interest in building dedicated hardware for these networks to increase the computational and training speeds. A lot of references of fully analog, fully digital as well as hybrid neurochips can be found in literature. [30] has a very good survey of neural hardware implementations until 1995. [11] has a good survey of FPGA implementations of artificial neural networks. Dedicated hardware can give much higher speedups in the recall speeds over software-only implementations, exploiting the inherent parallelism in these networks. Also, longer training times for these networks can benefit a lot from faster hardware implementations. This research focuses on the FPGA implementation of artificial neural networks and their on-chip training.

FPGA IMPLEMENTATIONS OF ANNS

The iterative training algorithms for these networks make custom hardware implementations of on-chip training of these networks challenging. Multiple training iterations with varying network parameters and structure need a custom hardware design that can support these dynamic changes in network structure and parameters. A custom ASIC (Application Specific Integrated Circuit) implementation of ANNs not taking these factors into consideration would be unsuitable for an online training platform. Consider the hugely popular multilayer perceptron (MLP) model of ANN. MLP is a feed-forward neural network comprised of layers of artificial neurons typically trained using the back-propagation algorithm. The first layer is called the input layer, last layer is called the output layer and the ones in between are the hidden layers. Figure 1a shows an example MLP network under training at training iteration '*n*'. Assume that in the next iteration '*n+1*' there is a slight change to the structure of the network; an additional neuron has been added in the first hidden layer of the MLP. This is shown with dotted lines in Figure 1b. Now if this network is implemented in digital hardware for online training the hardware designer has to somehow handle dynamic routing issues arising due to iterative structure and parameter changes. Also, the number of inputs to the neurons in the second hidden layer has increased from 4 to 5 as shown in Figure 1. Hence the sum of products module will have to increase the number of pipeline stages in the multiply-accumulate unit of these neurons dynamically to handle the additional inputs. For a rigid digital design this requires a time consuming hardware re-synthesis and re-routing, which kills any motivation for an online training scenario. These dynamic changes can be very easily handled in software making it attractive for training these networks. Providing this flexibility in digital hardware comes at a significant cost in area and speed. Due to such limitations most implementations reported in the literature use software for neural network training to find an optimal network and use custom hardware to achieve higher connections per second (CPS) recall speeds.

To gain the flexibility of software and the processing speeds of digital hardware many researchers have proposed FPGA implementations of artificial neural networks, relying heavily on multiple FPGA reconfigurations [12,15,31,32,33,34,35,36]. FPGAs are 'soft' hardware chips consisting of arrays of programmable logic components interconnected using programmable interconnects. The logic components are typically implemented using lookup tables and they duplicate the functionality of basic logic gates such as AND, OR, NOT, XOR. Current generation FPGAs include a lot of other complex functions on-chip such as memory elements, multipliers and accumulators, high speed serial communication transceivers, on-chip embedded processors or DSPs (Digital Signal Processor), etc. The advantage of hardware reconfigurability in FPGAs makes them very attractive low cost solution for prototyping digital circuits before fabrication. This considerably reduces the time-to-market for products using dedicated ASICs. Due to slow processing speeds and low capacities per unit size, the early FPGAs were mainly used as glue logic between various ASIC components on PCBs, but the current generation FPGA devices have come a long way. Higher processing speeds, typically a few hundred MHz and capacities in millions of equivalent ASIC gate counts make these devices a very attractive low cost, low volume option to ASIC implementations which typically cannot fit the budget for lower volume productions. Also, availability of good software design/verification suites from leading CAD companies makes them more attractive to hardware designers.

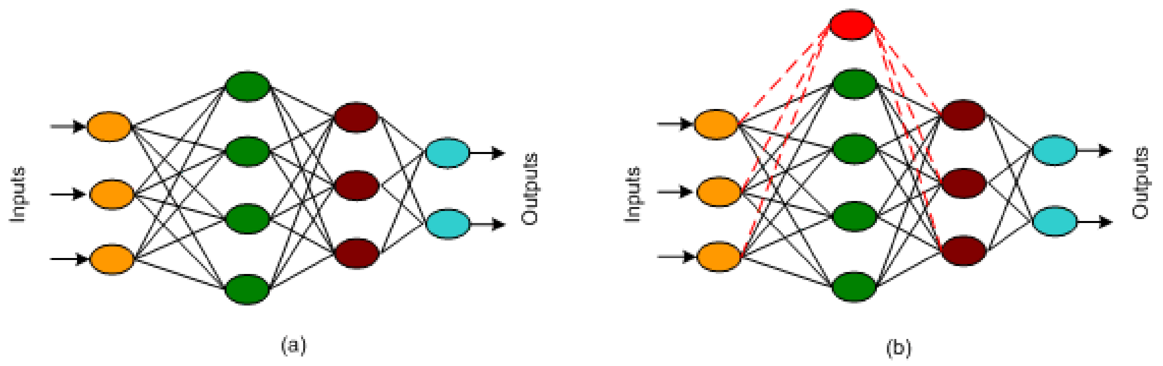


Figure 1: Multilayer Perception Example (a) Training Iteration 'n' (b) training iteration 'n+1'

FPGA IMPLEMENTATION ISSUES

Many researchers in the field of artificial neural networks have reported varying success with FPGA implementations of ANNs. Typical issues faced in FPGA ANN implementations are that of data representations and precision, reconfiguration time overheads, and activation function implementation. These play an important role, guiding various hardware design decisions.

DATA REPRESENTATION AND PRECISION

A double precision floating-point representation of data (weights, biases, inputs, outputs) in a neural network may still be impractical to implement on FPGAs despite the current advances in FPGA technology [37]. Floating point arithmetic units are slower and more complicated as well as need more silicon area than their integer counterparts. There exists a body of research to show that it is possible to train ANNs with integer weights and biases. The interest in integer values stems directly from the fact that integer multipliers and accumulators are much more efficient in terms of area and speed than corresponding floating point units. [38] proposed special training algorithms for multilayer perceptrons that uses weight values that are powers of two. This eliminates any need for multipliers in the ANN implementation, as they are replaced with simple shifters. Many approaches encode real values in randomly generated bit streams and implement the multipliers in bit-serial fashion, essentially serializing the flow and using simple logic gates instead of complex expensive multipliers for real estate efficiency. Murray and Smith's VLSI implementation of ANNs [39], used pulse-stream encoding for real values which was later adopted by Lysaght et al. [14] for ANN implementations on Atmel FPGAs. Van Daalen et al [17] used digital bit serial stochastic techniques to represent real valued signals so that the product of two stochastic bit streams can be computed using a bitwise exclusive OR. Economou et al [40] used pipelined bit serial arithmetic for their ANN implementation on FPGAs used for medical expert systems. Salapura [41] used delta encoded binary sequences to represent real values and used bit stream arithmetic to calculate a large number of required parallel synaptic calculations. [42] has a good overview of pulse stream arithmetic technique based hardware implementations of artificial neural networks. [43] is a more recent implementation using pulse stream encodings. The flip side of using pulse stream arithmetic approach is the precision limitation which can severely affect ANNs capability to learn and solve a problem. Also, for multiplications to be correct the bit streams should be uncorrelated. To produce these would require independent random sources which again require larger resources. Guccione and Gonzalez [13] have used vector based data parallel approach to represent real values and compute the sum of products. The distributed arithmetic (DA) approach of Mintzer [44] for implementing FIR filters on FPGAs was used by Szabo et al, [45] for a digital implementation of pre-trained neural networks. They used Canonic Signed Digit Encoding (CSD) to improve the hardware efficiency of the multipliers. Noory and Groza [18] also used the DA neural network approach and mapped it to FPGAs.

A more direct approach of using fixed point numbers for representing real values has also been of great interest as the fixed point arithmetic essentially uses integer arithmetic units. One of the delicate issues here is to select the weight precision. A larger precision will have fewer quantization errors but requires larger multiply-accumulate units, increasing the required area on silicon, whereas with choosing smaller bit widths, in effect lowering precision, arithmetic unit implementations will be simpler, smaller, faster, and more power efficient. But reduced precision causes larger quantization errors which could severely limit the ANNs capabilities to learn and solve a problem. There is a trade off in selecting one over the other, and a way to resolve this conflict is to select a 'minimum precision' that would be required for target applications. Holt and Baker [46] investigated the minimum precision problem on a few ANN benchmark classification problems using simulations and found that 8 bit fixed point precision was sufficient for networks to learn and correctly classify the input datasets.

RECONFIGURATION TIME OVERHEADS

One of the motivations of using FPGAs for evolvable hardware or ANN implementations is its flexibility of full/partial hardware logic reconfiguration. This gives us an advantage of software flexibility and higher speedups achieved by exploiting inherent parallelism in artificial neural networks. As classified by Zhu and Sutton [11] the purposes for FPGA ANN implementations are mainly – (a) prototyping and simulation, (b) density enhancement and (c) topology adaptation.

Prototyping and Simulation: Taking advantage of the multiple reconfiguration capability of FPGAs, they are used as prototyping chips to validate hardware design strategies before actually sending the validated design for fabrication. In the case of artificial neural networks, an evolved network design after training for a particular application is implemented on the FPGAs. The FPGA can be reconfigured with a newer design for a different application. The Ganglion project [12] used this strategy for different applications of their connectionist classifiers.

Density Enhancement: Full or partial FPGA reconfiguration can be used to implement hardware circuitry that is time sliced into multiple temporal stages, saving the intermediate results and feeding them back in the next stage. This approach temporally folds the hardware design and uses partial or full FPGA reconfiguration to reuse the limited FPGA resources. This increases the amount of effective functionality per unit reconfigurable circuit area of FPGAs. Eldredge et al [15] used run-time reconfiguration to implement the back-propagation training algorithm using three sequentially executable stages, temporally dividing the back-propagation algorithm into feed-forward, error propagation, and synaptic weight update stages. The feed-forward stage feeds in the input to the network and propagates the internal neuronal outputs to output nodes. The back-propagation stage calculates the mean squared output errors and propagates it backward in the network in order to find synaptic weight errors for neurons in the hidden layers. The update stage adjusts the synaptic weights and biases for the neurons using the activation and error values found in the previous stages. Hadley et al [31] improved the approach of Eldredge by using partial reconfiguration of FPGAs instead of full chip runtime reconfiguration. Gadea et al [32] show a pipelined implementation of the back-propagation algorithm in which the forward and backward passes of the algorithm can be processed in parallel on different training patterns, thus increasing the throughput. Another scenario here is as adopted by James-Roxby et al. [33]. They use dynamic reconfiguration of FPGAs to update the read only lookup tables with coefficients which can only be determined at run time for constant multipliers of a multilayer perceptron neural network.

Topology Adaptation: Run-time reconfiguration capability of FPGAs can be used to change the network structure and topology on-the-fly. This idea was initially proposed by deGaris [7] for evolving digital circuits under the control of an evolutionary algorithm. The same idea is applied to evolving neural networks. Perez-Uribe and Sanchez used this technique for implementing an adaptable-size neural network [34-36].

A typical current generation FPGA takes about 20-30 msec for a full run-time reconfiguration, assuming the bit stream to be loaded in the FPGA is preloaded in designated configuration memory [47]. For systems employing an online learning capability or using FPGA reconfigurations for density enhancement, the overhead of total reconfiguration time can be significantly high for multiple FPGA reconfigurations. In this case, performance hinges on the computational time versus the reconfiguration time. Guccione and Gonazalez [48] investigated this and came up with the equation: $q=r/(s-1)$; where s denotes the computational time, r denotes the reconfiguration time and q is the number of times the configured logic should be used before another reconfiguration to achieve good performance.

3.1.4 Activation Function Implementation

Activation functions or transfer functions are nonlinear monotonically increasing sigmoid functions. A direct hardware implementation of these functions can be very expensive. A more typical approach is to use piece-wise linear approximations of the nonlinear sigmoid functions as shown in Figure 2. One problem of direct implementations of the activation function is that one has to redesign the hardware logic for every application using a different activation function such as logsig, tansig, etc. Another approach is to use a preloaded lookup table for the activation function. The lookup table can be easily reloaded with the different values for different activation functions. This obviously is a lot easier than designing the hardware logic for a direct implementation and also could be faster in execution. The size of the LUT (Look Up Table) is directly influenced by the data precision used in the design. So if the LUT is embedded on chip along with rest of the hardware logic, the size of LUT that can fit could influence design decisions regarding the data precision being used. Wolf et al [49] have used piece-wise linear approximation techniques to implement nonlinear activation functions in their implementation. Krips et al [50] have implemented an MLP model on FPGAs for a real-time hand tracking system. The real valued data is represented as fixed point values and a tansig activation function has been implemented as a lookup table.

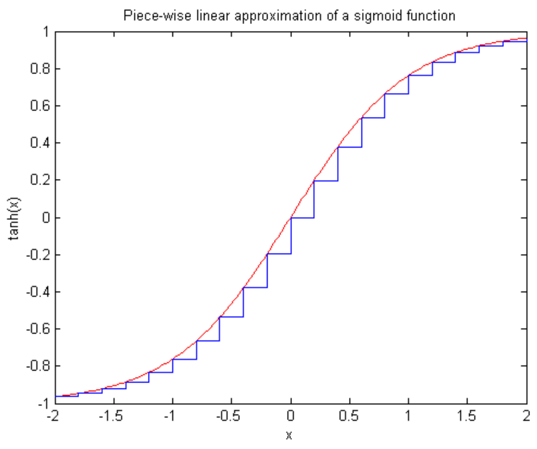


Figure 2: Piece-wise linear approximation of a nonlinear sigmoid function

3.1.5 Block-based Neural Networks

A block-based Neural Network (BbNN) is a flexible network design that is convenient for hardware implementation. [26,51,52,53,54,55,56]. A BbNN is a network of neuron blocks interconnected in the form of a grid as shown in Figure 3. These blocks are the basic information processing elements of the network. Depending on the number of inputs and outputs on a block, a basic neuron block can have the following possible variations in the internal configuration modes:

(a) 1-input, 3-output (1/3),

(b) 2-input, 2-output (2/2), and

(c) 3-input, 1-output (3/1).

Figure 4 shows the various internal configurations of a basic neuron block.

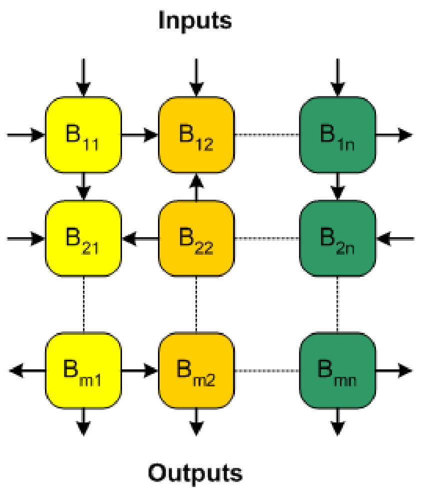


Figure 3: Example BbNN network Structure

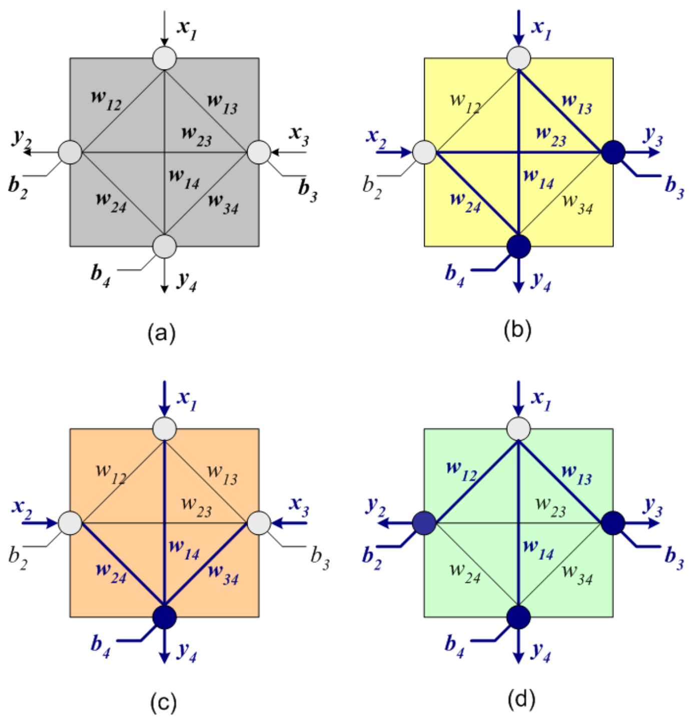


Figure 4: Four different internal configuration modes of a basic block
(a) General block (b) 2/2 configuration (c) 3/1 configuration (d) 1/3 configuration

Each individual neuron block computes outputs that are a function of summation of weighted inputs and a bias as shown in equation 1.

$$y_k = g\left(b_k + \sum_{j=1}^{J} w_{jk} x_j\right) \;,\quad k = 1,2,\cdots,K \qquad (1)$$

Where,

y_k	→ *k*th output signal of the neuron block.
x_j	→ *j*th input signal of the neuron block.
w_{jk}	→ synaptic weight connection between *j*th input node and *k*th output node.
b_k	→ bias at *k*th output node.
J, K	→ number of input and output nodes respectively of a neuron block.
g(•)	→ linear / nonlinear Activation function.

A neuron block can have up to six synaptic weight connections, three inputs and three outputs depending on the internal configuration of the block. A 2/2 neuron block has 6 synaptic weights, and 2 inputs as well as two outputs. Similarly, a 1/3 block has 3 synaptic weight connections, 1 input, and 3 outputs. The activation function *g(•)* can be linear (e.g., 'purelin') or a nonlinear function (e.g., 'logistic sigmoid'). Internal configurations of blocks used in the network are determined by the signal flow in the network from input to output which in turn is determined by the network structure.

3.1.6 MULTI-PARAMETRIC GENETIC EVOLUTION OF BBNN

To train the BbNN network, both structure and the synaptic connection weights of the neuron blocks need to be evolved simultaneously. The error surface in a multi-parametric optimization is typically non-differentiable with lot of discontinuities and many local minima. A gradient descent technique such as back-propagation will be very inefficient and may not converge at all, getting repeatedly trapped in local minima. A global search technique is required for such problems and it fits the case for using genetic algorithms.

The network structure and the synaptic weights in BbNN are encoded in a chromosome and simultaneously evolved using multi-parametric genetic evolution. The network structure is encoded as a gene using a sequence binary numbers. Any connection between the blocks is represented with either 0 or 1. Bit 0 denotes down (↓) and left (←), and bit 1 indicates up (↑) and right (→) signal flows. The number of bits required to represent the signal flow of an $m \times n$ block-based neural network are $(2m-1)n$. This is illustrated in Figure 5.

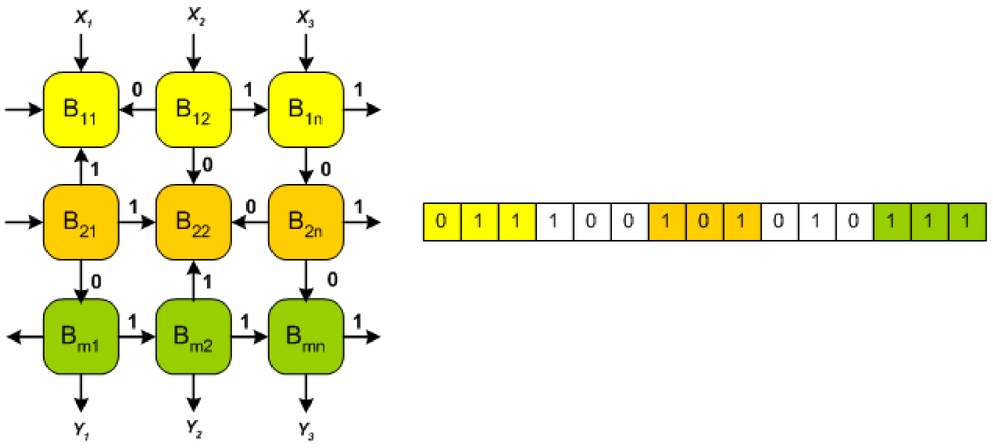


Figure 5: BbNN network structure encoding

Although BbNN block structures can support feedback connections with the bottom node being an input as shown in the example above, the hardware implementation has been restricted to only feed-forward networks for simplicity in hardware design and because many interesting applications can be successfully implemented as feed-forward networks. This also, reduces the size of the network structure gene, since the signal flow will always be downward and hence need not be encoded in the structure gene. Thus the number of bits required to represent the signal flow of an $m \times n$ block-based neural network is mn. This is illustrated in Figure 6.

Synaptic connection weights of each neuron block in a network are encoded in an array. The arrays of all the blocks are concatenated sequentially to form a weight gene. The weight gene along with the structure gene forms the BbNN chromosome. Figure 7 shows the BbNN weight gene encoding for a single block in a network.

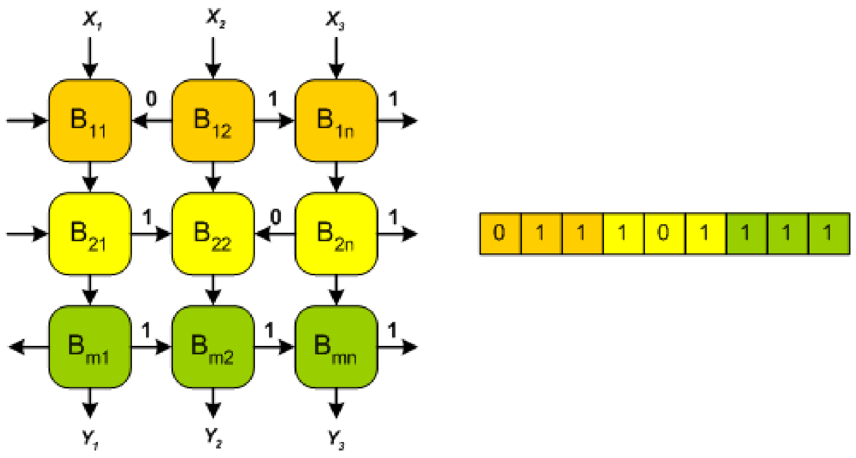
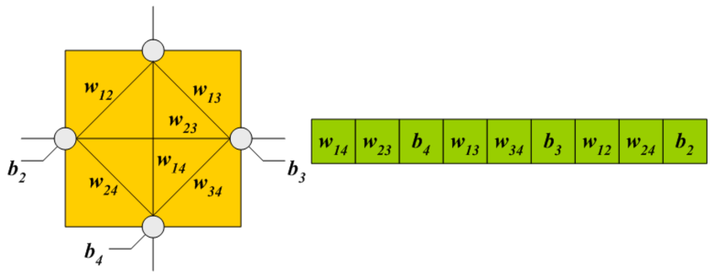


Figure 6: Network Structure encoding for a feed-forward BbNN network



w_{12}
w_{13}
w_{23}
w_{14}
w_{24}
w_{34}
b_2
b_3
b_4
w_{14} w_{23} b_4 w_{13} w_{34} b_3 w_{12} w_{24} b_2

Figure 7: BbNN block weight encoding

A population of these networks encoded in chromosomes is genetically evolved using selection, crossover, and mutation operators, with selection pressure against the least fit individuals, thus selecting fit individuals for survival or crossover to form newer generations. The fitness function is derived from total mean squared error between target and actual outputs of the network. Equation 2 shows the fitness function used.

$$Fitness = \frac{1}{1+e} \tag{2}$$

$$e = \frac{1}{Nn_o} \sum_{j=1}^{N} \sum_{k-1}^{n_o} e_{jk}^2 \tag{3}$$

$$e_{jk} = d_{jk} - y_{jk} \tag{4}$$

Where,

N	→ number of training data,
n_o	→ number of actual output nodes,
e_{jk}	→ error between desired and actual outputs of the k^{th} output block referred to j^{th} pattern,
d_{jk} and y_{jk}	→ desired and actual outputs of the k^{th} output block referred to j^{th} pattern.

The flowchart in Figure 8 shows the evolution process.

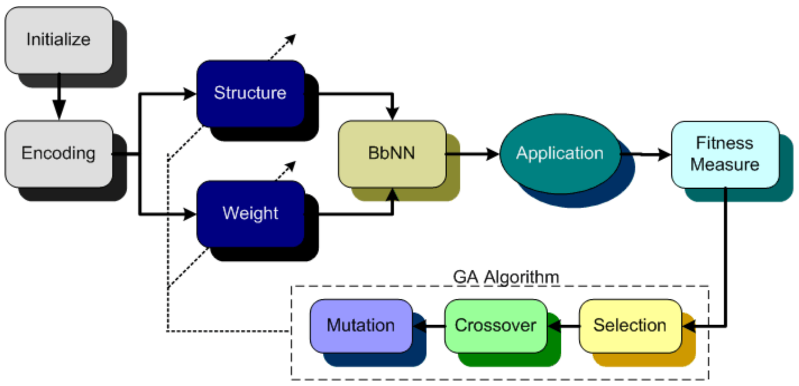


Figure 8: Flowchart showing GA Evolution

3.1.7 HARDWARE IMPLEMENTATION OF BLOCK-BASED NEURAL NETWORKS

The goal here is to build an intrinsically evolvable platform on FPGAs for evolution of block-based neural networks that can be trained online and dynamically adapted to changes in environmental stimuli. The grid based topology of BbNN limits the number of connections between neuron blocks to four connections per block. Thus, adding a new row or column of neuron blocks to the grid just adds fixed external routing connections that can be pre-determined during design and it does not affect the internal multipliers and accumulators of the existing neuron blocks. One of the disadvantages of the limited number of connections between neuron blocks is that more blocks are needed to solve a particular problem as compared to a multilayer perceptron model, but this is offset by the advantages of ease in developing a dynamically adaptable system.

SMART BLOCK-BASED NEURON BLOCK DESIGN

One of the challenges here is to design a neuron block that can dynamically emulate all the various internal configuration modes. The simplest approach to do this would be to design a library of neuron blocks for all the internal configuration modes and combine them in a super block using a multiplexer to select each depending on the structure gene. But the problem with this approach would be the silicon area required per such super block will be four times that required by a single block, making this brute force approach very inefficient. A smarter way is to develop a neuron block as shown in Figure 4a that has all the required connections for all the internal configuration modes, by selectively activating only the required connections for the configuration mode being emulated and deactivating others. This approach yields a neuron block design that requires about 35% of the silicon area as would be required by the brute force approach described above and still is able to emulate any internal configuration modes dynamically depending on the structure gene. Also, included in the design is a neuron block bypass 'greyed' internal configuration mode. In this mode the inputs are just passed on to the outputs with any modifications, essentially bypassing the neuron block. This was an important design choice to successfully implement an evolvable system as will be evident later. We call this design the smart block-based neuron design.

The design was implemented using a Xilinx Virtex-II Pro (XC2VP30) FPGA [47] housed on a Digilent Inc. XUP development board and also an Amirix AP100 board with the same FPGA [57]. Some of the interesting features of this FPGA are

- Two embedded PPC405 processors
- 30,816 logic cells
- 136 built-in 18 x 18 multipliers
- 2448 KBits (306 KBytes) on-chip block RAM
- 8 Gigabit Rocket I/O transceivers

The real values in the network are represented in an 8.8 fixed point format based on the simulation results of Holt and Baker [46]. Provisions are made in various design decisions to support a 16.16 implementation. All the synaptic weights as well as inputs and outputs are stored in software readable/writable memory-mapped registers. The activation function has been implemented using a software writable lookup table. One of the reasons for this design choice was the ease of implementation of activation function and the advantage of changing the activation function being used dynamically during evolution by rewriting the lookup table using software drivers. The size of the lookup table (LUT) required is directly associated with data widths used. An 8.8 fixed point representation requires a LUT that is 16 bits wide and 2^{16} deep. This requires a total 128 KBytes per LUT. The LUTs are implemented in FPGA using the available on-chip block RAMs. If we use a separate LUT for individual neuron block, we can only fit two blocks per FPGA chip before we run out of block RAMs. But, if the LUT is shared between all the neuron blocks we have to serialize access to the LUT using a FIFO, slowing down the computational speed. Hence a design decision was made to share a LUT between neuron blocks in a column. The advantage of this design decision is that no two blocks in a column can 'fire' (process) at the same time and thus will not need to access the LUT simultaneously. This will be evident when we discuss the dataflow implementation details. But again this still limits the number of columns that can be implemented to two, before we run out of block RAMs. To further optimize the size of the LUT we implemented a LUT that was 16 bits wide but only 2^{12} deep. This reduces the size of the LUT to 8 Kbytes per LUT. This was done taking into consideration that most of the activation functions used are saturating functions with output tapering off to a constant value. Thus there is no need to store these values repeatedly, in effect chopping of the activation function in the LUT beyond the saturated values on the negative as well as positive side. Thus the number of LUTs and hence columns that can be implemented on the FPGA would be a lot higher, not posing a bottleneck for implementation. Figure 9 shows a logic diagram for the smart neuron block implementation.

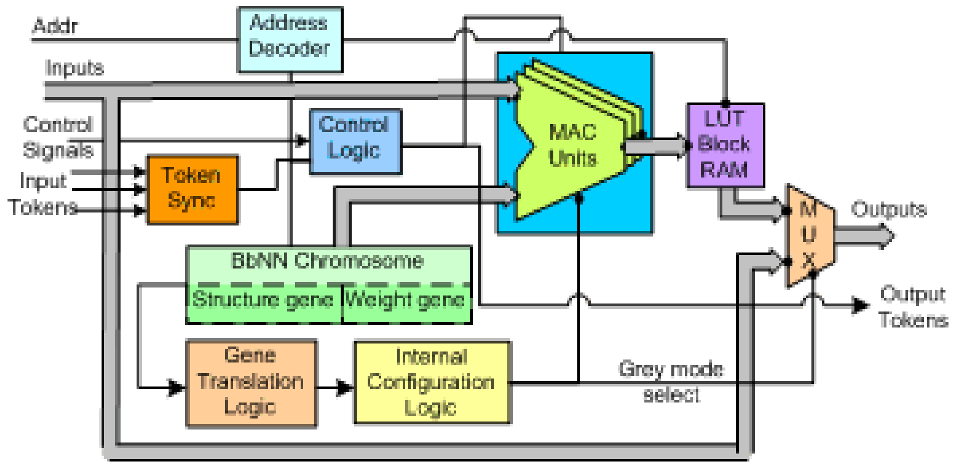


Figure 9: Logic diagram smart block-based neuron

DATAFLOW IMPLEMENTATION

One of the issues with implementing data flow architectures in hardware is to know when the outputs are stable. This is a much bigger problem with feedback in the network structure. Currently we are looking at only feed-forward BbNN networks. To solve the problem of latching the correct outputs, we have implemented a control structure inspired by a Petri-Net model architecture. A Petri net (also known as a place/transition net or P/T net) is one of several mathematical representations of discrete distributed systems. As a modeling language it graphically depicts the structure of a distributed system as a directed bipartite graph with annotations. As such, a Petri net has place nodes, transition nodes, and directed arcs connecting places with transitions [58,59].

At any time during a Petri net's execution each place can hold zero or more tokens. Unlike more traditional data processing systems that can process only a single stream of incoming tokens, Petri net transitions can consume tokens from multiple input places, act on them, and output tokens to multiple output places. Transitions act on input tokens by a process known as firing. A transition fires once each of the input places has one or more tokens. While firing, it consumes the tokens from its input places, performs some processing task, and places a specified number of tokens into each of its output places. It does this atomically, namely in one single nonpreemptive step.

The BbNN dataflow can be represented using an acyclic Petri net. Each of the blocks can be represented by an equivalent Petri net model as shown in Figure 10. The input and output registers can be represented by places. When each of the input registers (input places) have a valid input (a token), the BbNN fires and computes the outputs. Each of the output places will now get a token after the BbNN fires and the tokens at the input places are consumed. Thus the dataflow through a BbNN network can be represented using an equivalent Petri net network model (replacing each block with equivalent Petri net model as shown in Figure 10) for the entire BbNN network structure. Figure 11 shows the firing sequence for a particular BbNN network example. The side inputs have been hard-coded to be zero and have a valid token (shown as a '●') until consumed by firing. When the top inputs are applied the input places get tokens and they fire computing the outputs. As can be seen only the blocks with valid input tokens fire and validate the corresponding input tokens for the neighbors, which in turn fire next.

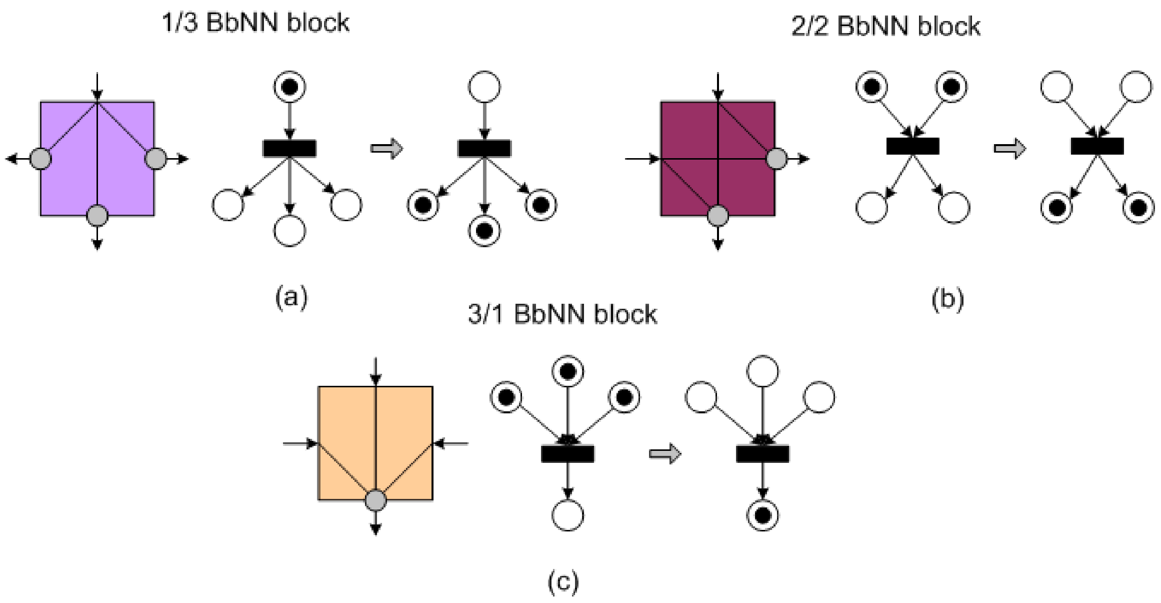


Figure 10: Equivalent Petri Net models for BbNN blocks (a) 1/3 (b) 2/2 (c) 3/1

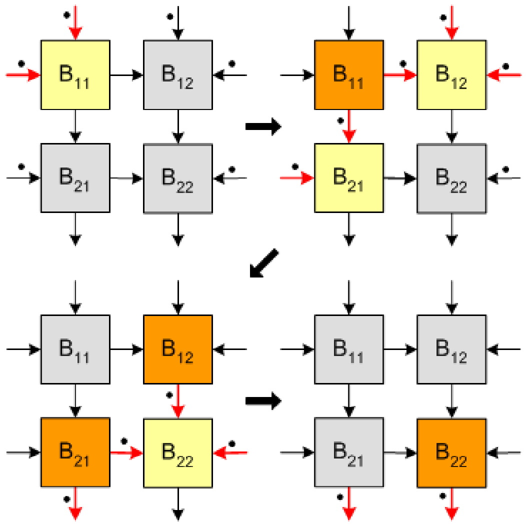


Figure 11: An example 2 x 2 BbNN firing sequence

3.1.8 INTRINSIC EVOLUTION STRATEGY

The BbNN design developed is part of a Programmable System on-Chip (PSoC). The PSoC platform is designed using the Xilinx Embedded Development Kit (EDK). It includes a PPC405 processor along with on-chip local memory communicating via Processor Local Bus (PLB), and other peripherals such as UART for serial communication connected as slaves on an on-Chip Peripheral Bus (OPB). The platform is shown in Figure 12. The GA code runs on the on-chip PowerPC processor. It writes the structure and weight gene in corresponding BbNN registers which are memory-mapped to the PowerPC processor. BbNN network structure in hardware dynamically realigns according to the structure gene loaded in the register. The inputs and outputs along with synaptic weights also are memory-mapped to the PowerPC and can be written to and read from using standard PowerPC assembly instructions. The computed outputs of the network can be read back for fitness evaluation. The entire genetic evolution process intrinsically runs on the FPGA without requiring a single FPGA reconfiguration cycle, thus avoiding reconfiguration time overhead for better speedups. The idea here is to program the FPGA once with the maximum network size that can be accommodated on the chip along with the platform and other required logic cores for the application at hand. The structure and weights can be evolved and adapted intrinsically in the field without requiring any further FPGA reconfiguration cycles. So, for example, we have a 3 x 6 network size programmed on the FPGA, then we can use any subset of that network size, such as 2 x 5, 3 x 2, or 3 x 6 without requiring any FPGA reconfiguration cycle.

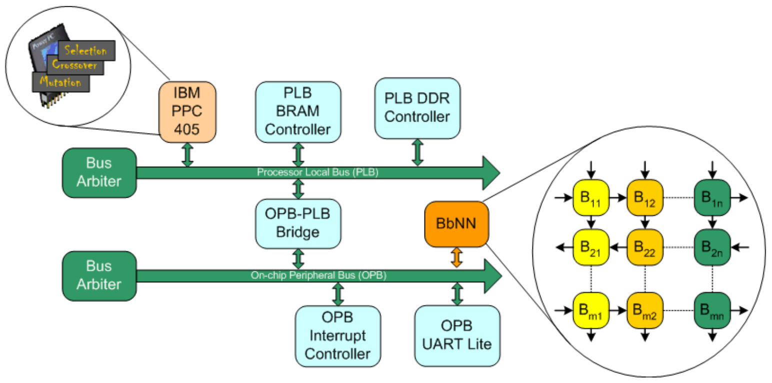


Figure 12: BbNN PSoC Platform

3.1.9 Performance and Utilization Summary

The BbNN network design can operate at 100 MHz frequency on the Xilinx Virtex-II Pro FPGA (XC2VP30). Each block takes at the most 10 cycles to complete processing of the inputs to produce an output. This again depends on the internal block configuration and the number of output nodes using the activation function LUT instead of a simple ‘purelin’ function with slope 1. Each block computation processes 6 synaptic connections. Thus, each block has a peak connection per second speed of 60 MCPS (Millions of Connections Per Second) per block for a 16 bit data width. With generally more than one block computing at a time, depending on the network structure the peak CPS would be *(n computing blocks)×(80 MCPS / block)* processing speed. Considering an $m \times n$ BbNN grid the theoretical peak processing speed would be *60n MCPS* and the minimum speed would be *60 MCPS.* In practice, processing speeds for each BbNN execution cycle would be different and will depend on the network structure, number of output nodes using the activation function LUT as opposed to a ‘purelin’ function and number of concurrent block computations.

The minimal platform excluding the BbNN network needs about 13% of the Xilinx Virtex-II Pro FPGA XC2VP30. Table 1 shows the post-synthesis device utilization summaries for various BbNN network sizes excluding the rest of the platform. According to the utilization summaries we can fit around 20 neuron blocks on a single FPGA chip along with the rest of the platform. Table 2 shows the post-synthesis device utilization summary for a larger FPGA device (XC2VP70) in the Xilinx Virtex-II Pro series, widely used in many commercially available FPGA boards. This device can fit around 48 neuron blocks.

Table 1: Post-Synthesis Device Utilization Summary on Xilinx Virtex-II Pro FPGA (XC2VP30)

Network Size	Number of Slice Registers		Number of block RAMs		Number of MULT18x18s	
	Used	Utilization	Used	Utilization	Used	Utilization
2 x 2	2724	19%	8	5%	12	8%
2 x 4	4929	35%	16	11%	24	17%
2 x 6	7896	57%	24	17%	36	26%
2 x 8	10589	77%	32	23%	48	35%
2 x 10	12408	90%	40	29%	60	44%
3 x 2	3661	26%	8	5%	18	13%
3 x 4	7327	53%	16	11%	36	26%
3 x 6	11025	80%	24	17%	54	39%
3 x 8	14763	107%	32	23%	72	52%
3 x 10	18456	134%	40	29%	90	66%
4 x 2	4783	34%	8	5%	24	17%
4 x 4	9646	70%	16	11%	48	35%
4 x 6	14587	106%	24	17%	72	52%
4 x 8	19508	142%	32	23%	96	70%
4 x 10	24461	178%	40	29%	120	88%

FIXED POINT SOFTWARE SIMULATOR FOR BBNN EVOLUTION

The BbNN genetic evolution code is written in the C programming language and runs on the embedded PPC405 processor in the FPGA. The real values as discussed above have been implemented as 8.8 fixed point numbers and hence have a storage type defined as 'short' in the C code. Since the PPC405 doesn't have any hardware floating point unit, the code almost uses no floating point operations. The code implements the genetic operators such as selection, crossover, and mutation as functions, and defines the BbNN chromosomes that contain the structure and weight genes, and BbNN populations as data structures. Various genetic evolution control parameters such as population size, maximum number of generations, maximum fitness value, etc. are implemented as global input variables. The software is cross-compiled using the

Table 2: Device Utilization Summary on Xilinx Virtex-II Pro FPGA (XC2VP70)

Network Size	Number of Slice Registers		Number of block RAMs		Number of MULT18x18s	
	Used	Utilization	Used	Utilization	Used	Utilization
2 x 2	2497	7%	8	2%	12	3%
2 x 4	4929	14%	16	4%	24	7%
2 x 6	7390	22%	24	7%	36	10%
2 x 8	9915	29%	32	9%	48	14%
2 x 10	12403	37%	40	12%	60	18%
3 x 2	3661	11%	8	2%	18	5%
3 x 4	7327	22%	16	4%	36	10%
3 x 6	11025	33%	24	7%	54	16%
3 x 8	14788	44%	32	39%	72	9%
3 x 10	18461	55%	40	12%	90	27%
4 x 2	4783	14%	8	2%	24	7%
4 x 4	9646	29%	16	4%	48	14%
4 x 6	14561	44%	24	7%	72	21%
4 x 8	19534	59%	32	9%	96	29%
4 x 10	24470	73%	40	12%	120	36%
4 x 12	29221	88%	48	14%	144	43%

Xilinx Software Development Kit (SDK) which is part of the Xilinx embedded development kit (EDK). The code is loaded on the SDRAM (Synchronous Dynamic Random Access Memory) during the PPC (Power-PC) bootstrap process either from an external compact flash card memory or using Xilinx debugger software and executed from SDRAM. Fitness evaluation is done in BbNN FPGA hardware circuitry as designed above. Also, written was software code for fixed point simulation of BbNN genetic evolution and fitness evaluations. This was also implemented in C. One of the purposes here was to verify and validate the outputs of the hardware design. Also, it provides a bit true software simulator for the BbNN evolution platform in hardware that can be used for research on newer evolution strategies and applications for BbNNs. The simulator can work with various fixed point representations. Floating point software for BbNN genetic evolution and fitness evaluations has also been developed for research purposes.

3.1.10 BbNN Evolvable platform applications

The developed evolvable hardware platform has many real-world applications. BbNNs have been applied to mobile robot navigation [26], multivariate Gaussian distributed pattern classification [51], chaotic time series prediction [52], and ECG (ElectoCardiGram) signal classification [53]. Here we have shown two example applications of our platform.

N-BIT PARITY CLASSIFIER

The designed platform was used for solving the n-bit parity computation problem. This is widely used in error correction and detection. A parity bit is a binary digit that indicates whether the number of bits with value of one in a given set of N bits is even or odd. The given set of bits is applied as inputs to the BbNN, and the output gives the value of the parity bit. A 3-bit parity has three inputs to the BbNN network thus needing at least three columns. Similarly 4-bit parity needs at least 4 columns. Figure 13 and Figure 14 show the evolved networks, and fitness curves for 3-bit and 4-bit parity problems, respectively. Various parameters used for the genetic evolution are as follows

- Population size = 30
- Target Fitness = 1.0
- Structure and weight crossover probabilities = 0.7
- Structure and weight mutation probabilities = 0.1
- Activation Function = Logistic sigmoid function
- Selection Strategy = Tournament selection
- Elitist genetic evolution model used to speed up the convergence.

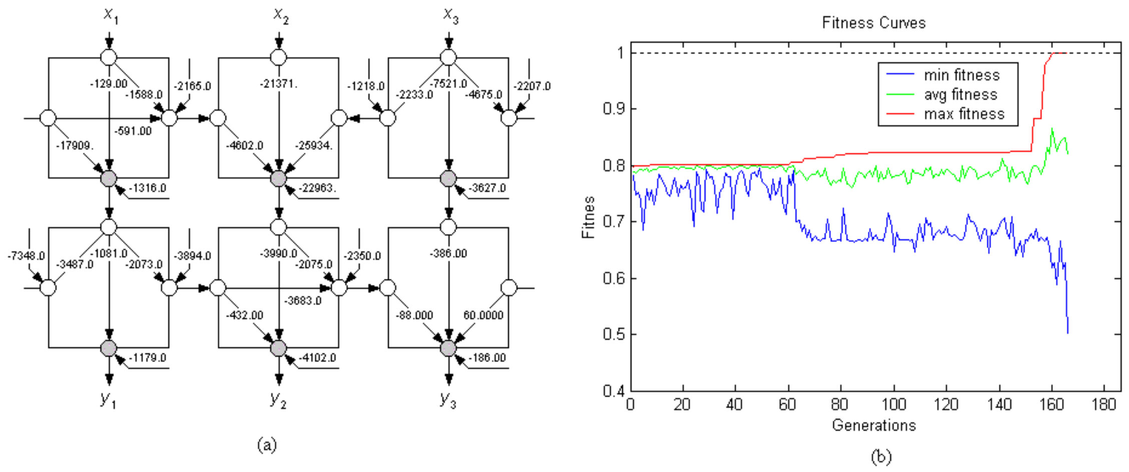


Figure 13: 3-bit parity example (a) Evolved BbNN (b) Fitness Curves

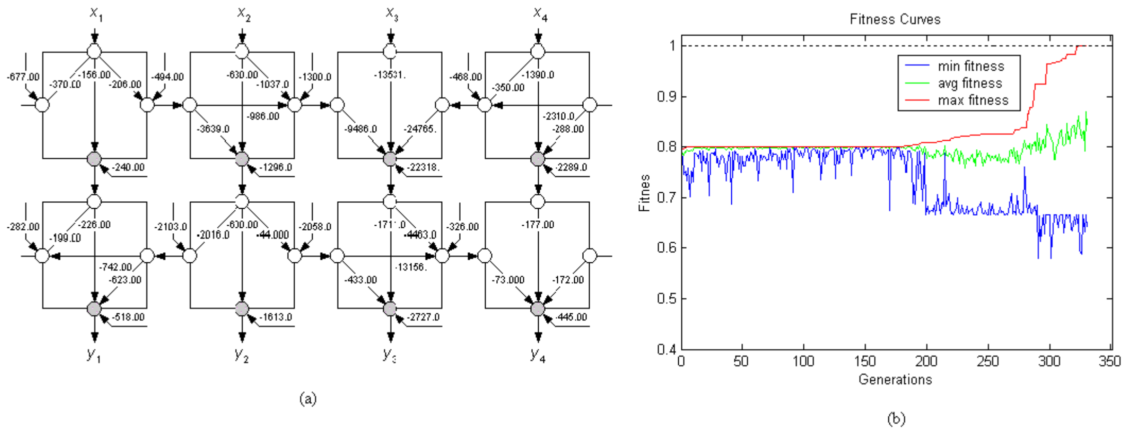


Figure 14: 4-bit parity example (a) Evolved BbNN (b) Fitness Curves

IRIS PLANT CLASSIFICATION

The dataset used in this classification example was originally compiled by R.A Fisher [60]. This dataset has been very widely used as a benchmark application for various classifier systems. It has 150 samples with three classes, with 50 samples per class instance. The attributes are sepal length, sepal width, petal length, and petal width for the three classes of Iris plants namely Iris Setosa, Iris Versicolour, and Iris Virginica. The Iris Setosa class is linearly separable from the other two, the latter are not linearly separable from each other. The designed platform for block based neural networks was used as a classifier for the dataset. The entire dataset of 150 samples was used as the training dataset. The inputs for the network were sepal area, and the petal area calculated by multiplying the sepal width with the sepal length, and the petal width with the petal length respectively. Figure 15 shows the evolved network, the classification error, and the fitness curves of the genetic evolution. We can see around 2% misclassification in the result which has been consistent with some of the other methods used for classifying this dataset [61]. The following genetic evolution parameters were used

- Population size = 80
- Maximum generations = 10,000
- Target Fitness = 0.9843
- Structure and weight crossover probabilities = 0.7
- Structure and weight mutation probabilities = 0.2
- Activation Function =Tangent sigmoid function
- Selection Strategy = Tournament selection
- Elitist genetic evolution model used to speed up the convergence.

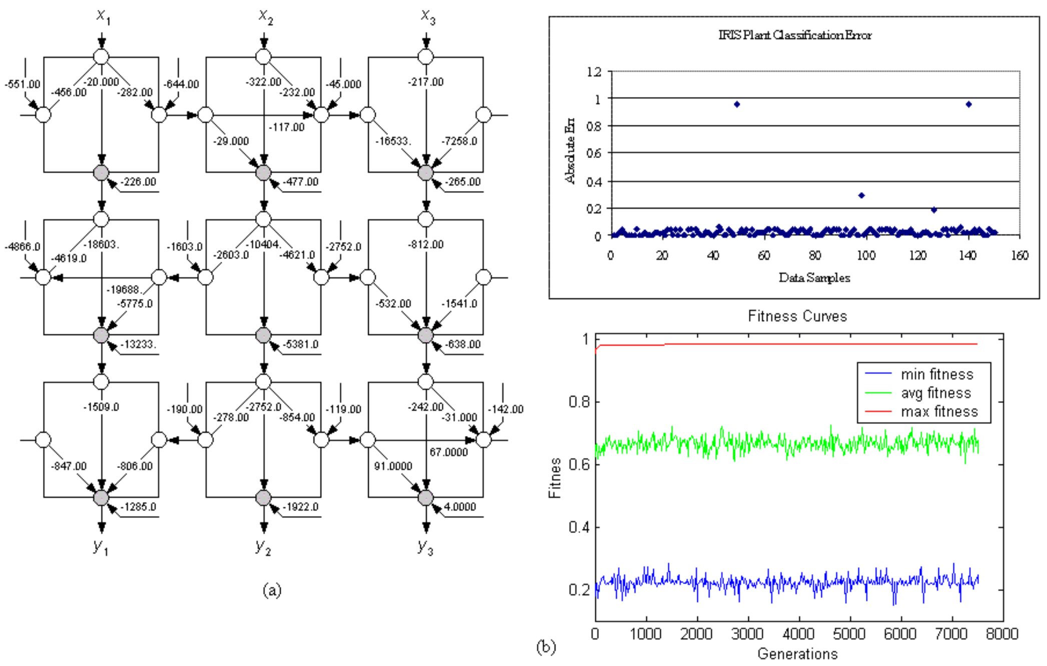


Figure 15: IRIS plant classification results

3.1.11 Neural Network Summary

The BbNN hardware platform developed here can be used for research as well as eventual practical implementation in an evolvable embedded system. Since the entire evolutionary algorithm, as well as the hardware network, all run on a single chip, they can be compactly deployed in a larger embedded system. The BbNN architecture gives us the flexibility and ease of hardware implementation, and promising capability for application in many real-world scenarios such as navigation [26], pattern classification [51], signal prediction [52], and biomedical signal classification [53]. The current implementation uses the Xilinx Virtex-II Pro (XC2VP30) FPGA which is at least a 3 year old technology from Xilinx. The current generation Virtex-4 and Virtex-5 FPGA series from Xilinx have at least 3 to 4 times the capacity of the Virtex-II Pro FPGAs used here. Currently we could fit a network with around 20 blocks in one Virtex-II Pro FPGA chip (XC2VP30) used here. As a comparison with a slightly larger FPGA chip (XC2VP70) in Virtex-II Pro family, we can fit about 48 neurons. With the currently available capacities we should be able to realize BbNN networks with about 60 to 80 neuron blocks. Also, the current implementation running at 100 MHz clock rate on the Virtex-II Pro chip could be expected to run at a much faster clock rate on newer Xilinx FPGAs, increasing our current 60 MCPS per neuron block processing speed to a much higher value. The PowerPC 405 processors available in Virtex-II Pro chips run at maximum 300 MHz delivering 400+ MIPS processing power. Current generation devices have on-chip PowerPC processors with 700+ MIPS (Millions of Instructions per Second) processing capability, significantly speeding up the genetic evolution process. To construct larger networks, multiple FPGA chips could be daisy-chained using the gigabit serial communication transceivers available on these FPGAs or reuse the BbNN platform on the same chip temporally. Other possible scenarios for future work include programming partial or the entire genetic evolution algorithm on the FPGA reconfiguration fabric. Hardware implementations of genetic algorithms have already been reported in literature [62,63]. This could significantly speedup genetic evolution. Looking at the past and current trends, in increasing capacities and speeds of the FPGA devices, and embedded processors in accordance with Moore’s law, and assuming it holds true over the next few years, widespread deployment of these platforms is becoming more practical.

3.2 MACHINE LEARNING WITH ARTIFICIAL BRAINS/MASSIVELY PARALLEL COGNITIVE SYSTEMS

Although artificial neural networks were developed to mimic the neurons in our brains, the processing performed by ANNs is quite different from that performed by biological neurons. In the past twenty years, much better understanding of the function of biological neurons has been obtained. Moreover, a number of research efforts have addressed building circuits that more closely approximate the behavior of biological neurons. The complexity of ANN models and overheads for simulation are significant, so that ANNs are not appropriate for modeling systems with large numbers of neurons, particularly when training is taken into account. Other approaches are the topic of research to better mimic biological behavior.

One area of research with interesting results in recent years pertains to the architecture of the cortical columns of neurons. Many believe that a seven layer model of the cortex with localized hypercolumns of neurons is accurate, with debate about the specific feedback mechanism within and between the hypercolumns. See Hawkin’s book for a good introduction on this area [64].

At a more detailed level, the function of biological neuronal synapses is known to be based on spiking signals, in contrast to the approach typically used with ANNs. See [65,66,67] for more information. This type of model is harder to implement with digital computers, requiring complicated continuous value simulation support in order to correctly compute the behaviors of interest. These models of neurons employing analog signaling and transformations may be implemented best with analog circuits implemented as ASICs/full custom integrated circuits (or perhaps with field programmable analog arrays, although these are not as powerful or economically viable as FPGAs).

The implementation of analog-based spiking neurons on traditional digital processors (or GPUs, FPGAs, etc.) is not a good fit, and would require very high computational power to achieve significant performance with large neuronal circuits. More abstract models of neuronal behavior have been fit onto supercomputers though. For example, IBM’s BlueGene/L supercomputer can be used to simulate such systems, for example with the SPLIT simulator evaluating a network of hypercolumns using abstract modeling of interactions (in this case with sequences of events on axons), reasonable scaling performance is reported with 8192 processors [68].

With a large number of researchers focusing their efforts on understanding the function of the brain, there is hope for significant improvements in our understanding of the brain and how to create artificial brains or intelligent systems along similar lines. The current state of the field still remains somewhat immature, with ongoing fundamental research addressing the structure and behavior of neurons, hypercolumns, and higher structure, not to mention learning, representation, ontologies, and other topics. Hence, it is still difficult to forecast the best computational technologies for cognitive processing applications based on brain circuit emulation, but it seems most likely that large, detailed analog circuits will require large arrays of custom analog circuits, but that large digital systems could be used for higher level research, training, and learning.

3.3 DYNAMIC PUBLICATION/SUBSCRIPTION DATABASES

A major component of the Joint Battlespace Infosphere is the capability to collect information and disseminate it in an efficient, scalable, flexible manner. To do so, database queries must be supported as well the ability to quickly determine permissions for producers and consumers to publish data and subscribe to data. The access control, authentication, cryptographic, and related services have been studied extensively, and reconfigurable computing or application specific integrated circuits have shown great promise for accelerating this processing. For instance, AFRL in-house research using string matching approaches for pub/sub processing showed speedups of hundreds of times over software processing. In order to assess the effectiveness of these technologies for this type of cognitive processing, we consider related processing from bioinformatics.

As part of a separate effort, we performed related research addressing bioinformatics database processing. Previous work addressed the BLAST string matching code for comparing genomic data by accelerating its execution using parallel processing and reconfigurable computing [69]. Similar acceleration can be applied to other bioinformatics codes.

The HMMER package is widely used for the detection of protein sequence homology, functional annotation, and protein family classification. It uses profile Hidden Markov Model (HMM) methods for sensitive database searches. Multiple sequence alignments are used as search queries to build statistical models for database searches [70]. HMMER operations rely on accurate construction of the profile HMMs. These HMMs are applied to protein sequence databases for homology determinations in order to extend the protein families with functional annotations of query sequences. HMMER functions are based upon a profile HMM architecture which is constructed using a plan-7 model [71]. The plan-7 architecture is constructed using the Viterbi algorithm [71,72,73]. The hmmpfam tool compares the protein sequences to the protein domain databases of HMM models and identifies the protein domains in the query protein sequence [70].

The nucleotide and protein sequences in various databases are growing at an exponential rate [74] and doubling their size every six months [75], but according to Moore's law [2] the number of transistors on a chip doubles every 18 months. Hence, processor performance improvements lag behind the growth of sequence databases. This performance gap led to porting the HMMER algorithms onto clusters of computers, shared memory architectures, networks of workstations, GPUs, and FPGA-based hardware accelerators. Each of these HMMER acceleration approaches focuses on improving the speeds of either a single search or a few hundred protein searches. There are around 6.5 million protein sequences in the non-redundant (nr) protein database [76] alone and an estimate of around 13 million proteins currently known. This number is growing rapidly. Thus to identify domains for these millions of proteins it will take months or even years of time based on computations done on either clusters of computers or on a single custom architecture. At the same time, PFAM [77] domain models are increasing every year. With every new release of PFAM domain models one has to run the millions of proteins to update the databases with the latest domain information.

Here, we describe an efficient parallel hmmpfam tool used on a supercomputer that allows the analysis of millions of proteins in less than a day using thousands of processing nodes. We modified the hmmpfam.c code using MPI (Message Passing Interface) [78] to distribute individual serial hmmpfam jobs to each of the computer cores. Each hmmpfam job accesses its own input sequence with no communications between the computer nodes. This scenario is often referred to as “embarrassingly parallel”. The approach obviously simplifies programming issues while avoiding overhead associated with sending and receiving messages. This latter problem is likely a significant factor in the low performance observed with MPI-HMMER, especially when the number of computer cores is high (more than 500). The performance of our parallel version of HMMER and MPI-HMMER are both affected by the intense I/O for reading and writing to and from disk. Input data has to be read for each protein sequence, with the analysis results written to separate output files. When using 1000 or more compute cores, this can create very intense I/O traffic. We partially mitigate this problem by redistributing the protein sequences for each hmmpfam job. The main idea consists of distributing different lengths of amino acid sequences so that each job finishes (and writes results) and starts a new sequence (reads) at different times, thus randomizing as much as possible the I/O events. This minimizes simultaneous reads/writes and avoids major time delays due to contention. Once the computation is done we populate a flat file database with proteins and their respective domain information using simple scripts.

Both coarse-grained and fine-grained parallelisms, along with hardware acceleration, are exploited to accelerate HMMER algorithms. The HMMER software distribution comes bundled with a PVM (Parallel Virtual Machine) [79] implementation. HMMER is a computationally intensive algorithm, so the higher the processor speed the faster the execution [80]. Hyper-threading and load balancing play significant roles in increasing speedups of HMMER [80]. JackHMMER exploits the coarse-grained parallelism to accelerate profile-HMM searches [81]. JackHMMER is a version of HMMER designed to run on an Intel IXP 2850 network processor consisting of heterogeneous multi-core processors. By using a high degree of thread-level parallelism, JackHMMER outperforms the hyper-threaded HMMER version on a Pentium 4. ClawHMMER, a streaming implementation written in the Brook language to run on graphics processors, outperforms CPU implementations [82]. Opteron processors are also used to accelerate HMMER searches with minimally invasive recoding, and the authors claim to achieve better performances than Intel architectures [50]. FPGAs are used to design an accelerator for HMM search that exploits both coarse-grained and fine-grained parallelism [83,84]. FPGA-based hardware accelerator of HMM search achieved 100-fold speedup over the software HMM search implementation [83]. A combination of hardware (FPGA) and software acceleration is used by MPI-HMMER-Boost [85] to accelerate the hmmsearch and hmmpfam tools. MPI-HMMER performs well on small clusters from a few nodes to tens of nodes [86]. This led to the development of parallel I/O HMMER that performs well for a few hundred nodes [86]. There is also a web-based interface to submit batch jobs without a local HMMER installation known as SledgeHMMER [87]. Although each of these approaches shows promise, none provides the scalable performance required for the scope of processing addressed herein (and that relates to JBI applications).

More than thirteen million protein sequences exist in various databases, with the latest nr database containing around 6.5 million protein sequences. The rate at which new proteins are discovered is growing exponentially. The distribution of protein sequence lengths is shown in Figure 16 [88]. The large variation in protein lengths enables us to take advantage of this variation to randomize the communication between the processing nodes and the file system to reduce contention due to correlated I/O. The computation time of hmmpfam depends on both the query sequence length and the domain database size [70,86]. The PFAM22 database is used for domain identification in this paper, so each node uses the same database. Thus, the computation time is proportional to the sequence length; so longer sequences take more time to finish than shorter sequences.

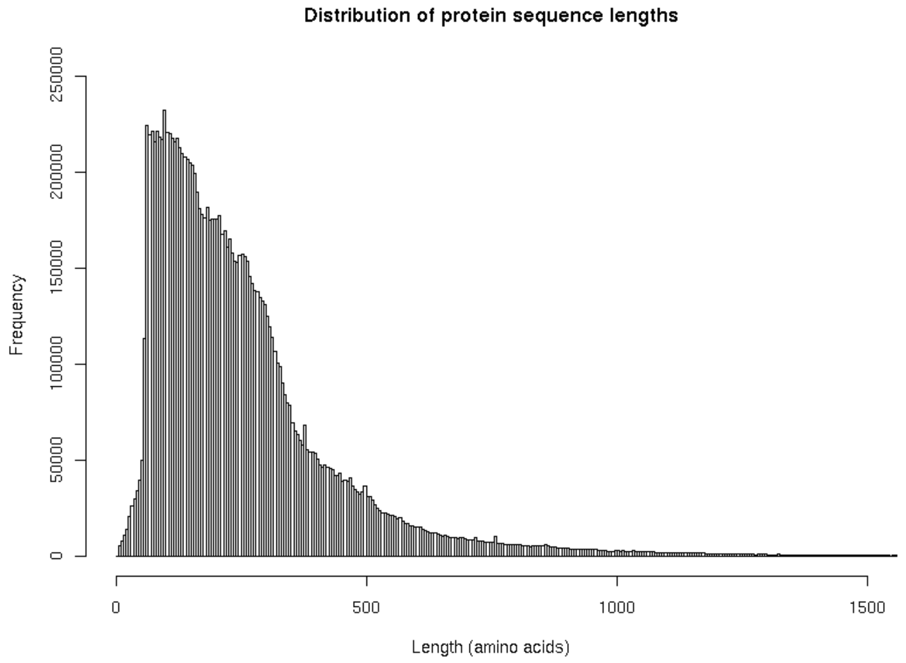


Figure 16: Distribution of Protein Sequence Lengths

HSPHT was run using a combination of nodes ranging from 16 to 8192 processing cores on Jaguar, the fifth supercomputer on the Top500 list as of June 2008 [89]. At the time of the experiments, Jaguar was a 250TF Cray XT4 system with 7832 nodes, each of which consisted of a quad-core 2.1 GHz AMD Opteron processor with 8GB of memory per node and 600 TB available in the scratch file system. There were more than 31K processing cores on Jaguar. According to the job scheduling policy, smaller jobs could run only few hours and the biggest job could not exceed 24 hours. Hence, one has to optimize the core request based on the job size. Two data sets are used for computation: dataset1 is a subset of the nr database with 300K proteins (~300 million AA(Amino Acid) count) used for performance comparison between MPI-HMMER and HSPHT. The entire nr database was used for getting domain information for all 6.5 million (~2.25 billion AA count) proteins as a second dataset.

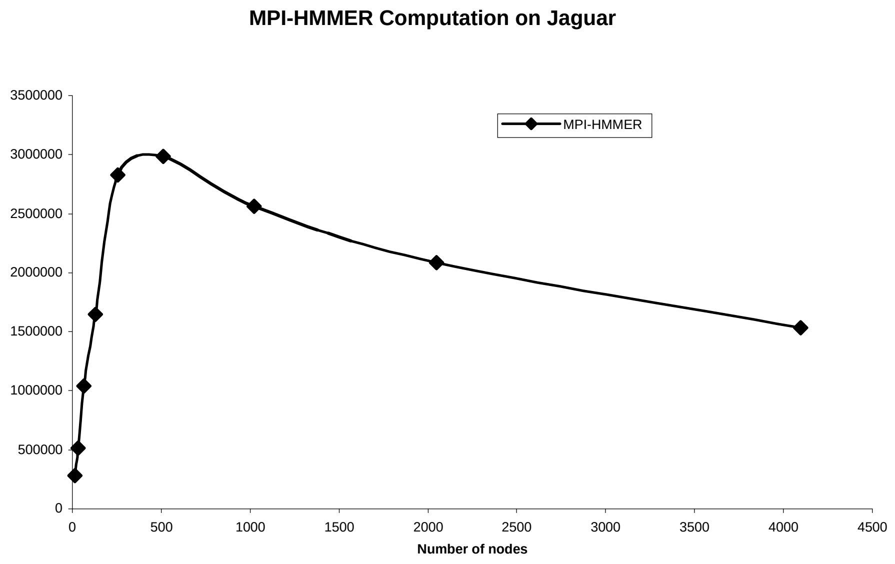


Figure 17: Computation Speed for MPI-HMMER

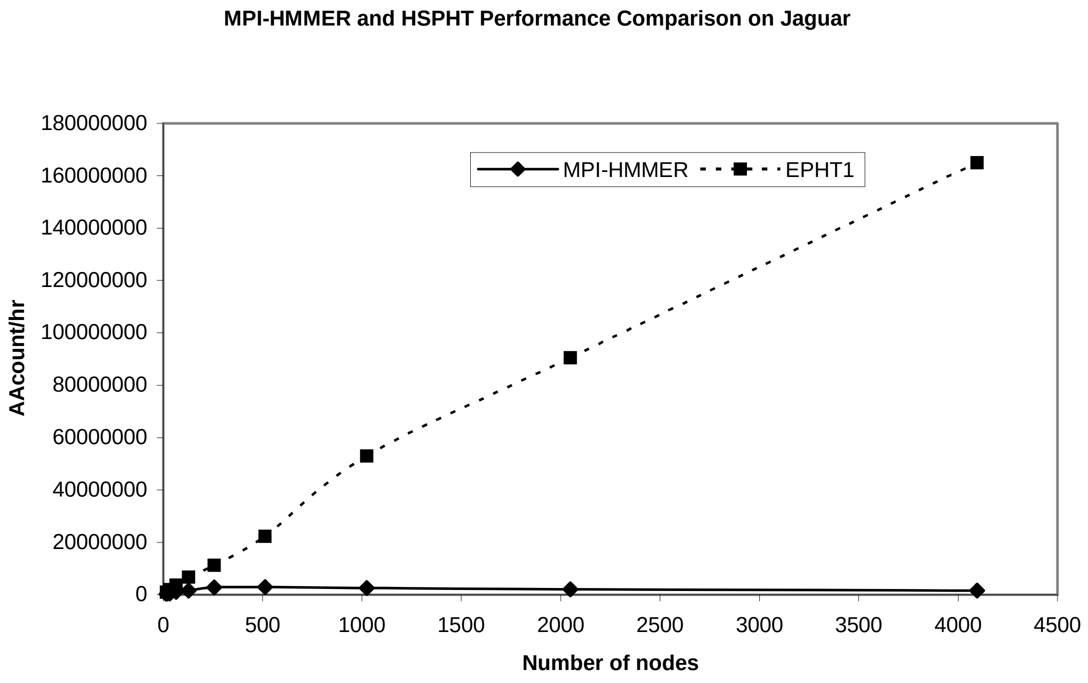


Figure 18: MPI-HMMER and HSPHT Performance on Cray XT5

We use dataset1 with 300K proteins for performance comparison between MPI-HMMER and HSPHT. MPI-HMMER was run using 16 to 4096 processing cores on Jaguar, allocating an hour per job. Figure 17 shows the total number of AA count processed by these jobs per hour using different combination of processing cores. From Figure 17 it is clear MPI-HMMER performs linearly until 256 processes and then flattens between 256 and 512 processes. The performance drops beyond 512 processes due to increased communication contention. The best allocation using MPI-HMMER on Jaguar is to submit jobs using 256 processes. With this low number of processing cores per job one can execute a job for only 2.5 hours. Thus with MPI-HMMER using 256 processes per job it will take ~320 (2.5 hours/job) simultaneous jobs, taking approximately just over a month to finish computation on Jaguar. Figure 18 shows a comparison between the MPI-HMMER and HSPHT computation using thousands of processes, demonstrating the embarrassingly parallel approach is much more scalable than MPI-HMMER. We reduced simultaneous communications by separating the output files for each process and randomizing the sequence lengths, thus achieving optimized bandwidth and reducing the latency. Speedups of 3-4x are achieved by HSPHT over MPI-HMMER until 256 processes and a speedup of ~107x using 4096 processes.

One advantage of HSPHT is the utilization of the multi-threaded functionality of HMMER. We compared the performance of HSPHT using one thread and two threads jobs with processes varying from 16 to 8192 cores. The single threaded implementation uses 1 thread per processing core, whereas the dual threaded implementation uses two processing cores. Thus a job with N processes uses N cores for the one-thread implementation and 2N cores for the two-threaded implementation. Due to the limited allocation of processing hours on Jaguar only these two runs were conducted. The two threaded implementation gives almost two fold speedups using 256 to 2048 processes, but the best performance is achieved between 512-1024 nodes. Thus the entire nr database was divided into four equal parts. Four jobs were used for finishing the computation for the entire nr database, with each job consisting of 1024 two-threaded processes using a total of 2048 processing cores. Each of the four jobs took less than 12 hours thus taking less than two days for identifying domains for all 6.5 million proteins of the nr database using HSPHT. The estimated time to complete identifying domains for all the proteins in nr database using 2048 dual threaded processes is less than a day.

We demonstrated that the embarrassingly parallel approach used in HSPHT performs much better than MPI-HMMER for large problems on supercomputers. With this approach we can identify domains for millions of proteins in a day, as compared to months of processing for other approaches. This demonstrates the power of supercomputing for attacking large-scale bioinformatics problems. HSPHT achieves speedups of over 100 times faster than MPI-HMMER for the nr dataset. Future work includes developing better load balancing strategies for very large problems. We are also working on further improving the HMMER code so that the communications are minimized between the processing nodes and file systems and extending the efficient embarrassingly parallel approach to support both hmmpfam and hmmsearch.

3.4 TEXT AND IMAGE PROCESSING

The processing of text, images, or other signals to extract information is ubiquitous for the DoD and one of the primary forms of input to any current or future cognitive systems. We consider each of these types of processing as typical, important types of computation for cognitive systems and discuss the suitability of emerging computational platforms for each of these types of computation.

Processing text may involve symbolic computations, including integer operations and many branches, or could include processing using Bayesian reasoning or Hidden Markov Models implemented using floating point computations. Other types of text processing include string matching (as with the examples with BLAST discussed above). All of these types of text processing are often performed with processors, but often with low efficiency with respect to speed or power dissipation. There has been some success with using GPUs to speed up some of these types of codes; see for example the work with ClawHMMER [82]. GPUs should perform best when the computations are dominated by single precision floating point operations. Reconfigurable computing platforms have had great success with this type of text processing. By building custom parallel datapaths, RC implementations can concurrently search various paths for determining likely meanings or predictions for text strings. This type of approach is very good for decimating data demands for processing further downstream, as with BLASTp acceleration [69].

Image processing is the raison d'etre for GPUs, so they tend to perform well for these computations. Processors include special multimedia instructions and functional units to accelerate these types of operations. FPGAs have also been shown to be quite good for these computations, particularly when there can be some optimization based on word sizes. Similarly, signal processing works best on processors that are optimized for this type of computation (DSPs). GPUs can also perform well for these types of computation, but are hindered currently by immature tools. FPGAs provide great flexibility for signal processing applications, with the ability to relatively easily build a pipelined datapath optimized for the specific application needs. For this constrained problem domain, the FPGA tools are most mature, with the ability to take higher level codes and generate bitstreams to program the circuitry. Accordingly, power efficiency for FPGAs is much better than for more traditional processors.

3.5 SCIENTIFIC COMPUTING CODES

Computational science codes typically involve high precision, very large data sets, and often include linear algebra formulations. Processing these applications on vector supercomputers then massively parallel supercomputers has been the topic of research for decades. With modern supercomputers now achieving more than a petaflop, truly massive computations can be supported. Modern microprocessor architectures have been stagnant with respect to clock speed or microarchitectural improvements, so the basic building block has now become the multicore processor. Each of the largest supercomputers now available is based on these technologies.

As an alternative, the use of accelerators has emerged in the past few years as an area of exciting research for computational science applications running on supercomputers. Technologies such as Clearspeed's floating point accelerators, general purpose processing with graphical processing units (GPGPUs), and reconfigurable computing with FPGAs have all become topics of interest [90,91,92].

The data precision of computations has great impact on the effectiveness of each of these accelerator technologies for improving performance of applications. For example, the traditional processor architecture is hampered by the overhead of instruction processing, restriction of predetermined data (word) sizes, and managing data storage and movement. Hence, compilers and code developers must exert great effort to produce instruction sequences that minimize data movement, keep functional units busy, and perform computations in the fewest possible number of instructions. To the extent that the operations and data sizes map well for accelerators such as the multimedia extensions (e.g., SSE or MMX) and the data movement with the registers, then code can execute faster. Similarly, if the application can be mapped to threads or other forms of concurrency, then the multicore processors can be successfully exploited. If the application can be partitioned into very large numbers of concurrent tasks that do not require significant communication or coordination, then large parallel supercomputers may be used to accelerate it.

The use of Clearspeed accelerators or GPUs promises speedups for floating point computations. In particular, GPUs can provide significant speedups for single precision floating point, but not as well for double precision (anecdotal evidence from a range of applications indicate speedups for floating point intensive codes from 50-100x for single precision, but with speedups for double precision from 5-10x). Many higher speedups have been reported recently, but most (if not all) of these numbers are based on poor serial implementations as the baseline. A GPU board such as the Tesla 1060 from NVIDIA can perform at about the same rate as a dual quad-core Intel or AMD workstation (assuming both platforms have been tuned and optimized appropriately). Current capabilities of GPUs show limitations with the ability to move data flexibly, perform low-level optimizations, or to share the GPU resources with multiple functions or processes, but these limitations are easing over time as the technology matures. In the next chapter, we discuss the languages and design environments for GPUs and other computing technologies.

Reconfigurable computing promises the ability to build pipelined datapaths on demand with the specific mix of functional units, pipelining, resource sharing, operand size, and clock speed to best perform the computations. In recent years, the size and speed of FPGAs has increased significantly enough that scientific computations now are practical targets for the technology (Underwood reports an increase in performance over a decade of 10,000x compared to 100x for microprocessors [93]). Earlier reconfigurable computing systems suffered from relatively low bandwidth to/from the FPGA resources as well as limited memory. Newer architectures that support AMD’s hypertransport [5,6] or Intel’s QPI (Quick Path Interconnect) [6,141] promise to provide much better bandwidth. The amazing flexibility promises tremendous performance improvement, but at significant development cost because design tools and environments are still immature. We discuss this issue in more detail in the next chapter.

The optimized datapaths for FPGAs have been the focus of RC research for nearly twenty years. More recently, we have been investigating the ability to develop optimized algorithms that take advantage of the application-specific word sizes, including variable precision floating point. A student in our research group recently won the Supercomputing Student Research Contest with his implementation of a variable precision solver that used faster, lower precision circuitry during the computationally expensive approximation followed by higher precision iterative refinement [108]. For more information on this work, as well as related work with precision for fixed point representation used instead of double precision, see [94] and [95].

To the extent that cognitive processing applications share the attributes of scientific computing codes, these lessons will be applicable.

4 MAPPING APPLICATIONS ONTO ARCHITECTURES/CAD PROGRAMMING ISSUES

The problem of how to implement or port an application on a computational platform has profound impact on the practical ability of systems developers to exploit various computing technologies. The development of sophisticated cognitive processing applications faces enough difficulties without the additional burdens of immature development environments. Nonetheless, the runtime systems and development environments for newer computing platforms always lag behind the hardware systems. To address this issue, we next briefly discuss the infrastructure available for application development with the computational platforms of interest.

A significant amount of related work addressing application development for reconfigurable computing and next-generation parallel processing has been performed under the auspices of the DARPA Adaptive Computing Systems (ACS) and High Productivity Computing Systems (HPCS) programs, respectively. Under ACS, the Synopsys Nimble Compiler effort targeted the synthesis of C onto reconfigurable logic, but is not oriented for high performance computing tasks. The USC/ISI DEFACTO effort exploited the SUIF project to address compilation of more abstract representations onto reconfigurable hardware, and included some nice capabilities to map to customized Arithmetic Logic Unit ALU/CPU structures for a given program. Once again, this is a very low-level view of the partitioning and mapping problem. The compilation technology from these program can be applied to compiling function blocks to reconfigurable computing elements once partitioning is completed and the blocks are mapped to RC elements.

Efforts from Northwestern University and UT exploited visual programming languages like Khoros or MATLAB to ease the reconfigurable hardware programming task. These approaches provide an excellent programming model for applications developers. These projects do not support the migration of functionality between CPUs and RC elements in either a static or dynamic manner.

There are several Java-related ACS programs for applications targeting FPGAs. Brigham Young University (BYU) researchers developed the JHDL Java-based hardware description language environment for programming FPGAs. The approach enjoys the benefits of code portability inherent with Java, while still achieving good hardware performance on the FPGA devices. With respect to hardware/software systems, the approach suffers from the reduced performance typically encountered with Java interpreters or compilers, which limits its applicability to high performance computing environments. A related effort from Lava Logic (a division of TSI Telesys) targeted the use of Java for programming reconfigurable devices by developing hardware implementations for Java bytecodes, but it suffered from the same high performance computing software drawbacks as the JHDL effort. The Xilinx JBITS effort targets the use of the fine-grain reconfigurability of the Virtex family of FPGA devices. This ability to perform fine-grain reconfigurability during operation promises a performance boost, but focuses on low-level design issues.

The System Level Applications of Adaptive Computing (SLAAC) program provided the primary demonstration vehicle for the DARPA ACS program. A SLAAC RC architecture was developed and used for a number of technology demonstrations. In addition SLAAC completed compilation work to map C to VHDL on Annapolis Microsystems RC boards. The Virginia Tech ACS work used the BYU JHDL design environment, but it did not have automatic partitioning onto multiple FPGAs. The manual intervention required for this task is significant;

supporting multiple RC boards further complicates the design task. The SLAAC API (Application Programmers Interface) was developed at Virginia Tech to make porting application host code between reconfigurable computing boards easier. Related efforts are ongoing with OpenFPGA to develop common APIs for RC platforms. The CHREC NSF center for HPRC research is addressing a number of these concerns with their research projects as well.

A number of C-based languages have been developed by vendors, particularly for use with their boards. Examples include HandelC, MitrionC, and DimeC. Similarly, SRC provides Carte and Starbridge sells VIVA, which are library-based programming environments for RC platforms that support higher level languages such as C or FORTRAN. None of these languages has become dominant, nor has any supplanted Verilog or VHDL for development of RC applications.

In an attempt to better support portability of vector, signal, and image processing applications, the embedded systems community created the VSIPL (Vector, Signal and Image Processing library) API for commonly used functions. Vendors for various HPC platforms support the VSIPL API by providing a library of functions optimized for their particular machines. Because a variety of vendors support VSIPL, applications developers can develop their applications using familiar programming languages/tools like C/C++, MATLAB, or Khoros, and can more easily port their applications to other platforms.

As part of the HPCS program, several new languages are under development and evaluation, including X10, Chapel, and Fortress, as well as related work with CoArray Fortran and UPC. There has been significant work in testing these languages and benchmarking their performance for various applications and platforms to evaluate their effectiveness and appropriateness (for one example, see the work at ORNL located at http://www.csm.ornl.gov/essc/benchmarks/index.html). We will not discuss these languages in detail in this report as they are not directly focused on cognitive processing applications and architectures.

4.1 PROGRAMMING ENVIRONMENTS

It would be ideal to have a tool that would let users design their application and map it onto the HPRC system without having to know the details of parallel processing or reconfigurable computing. Such a tool would essentially achieve the following:

User → CAD Tool → design specification → HPRC

We know that the modeling framework for the HPRC architecture is also suitable to the design of Systems on a Chip (SoC), because the design and architecture of HPRC and SoCs will be homomorphic. Both the architectures contain Reconfigurable Units (RC units) and also share some of the same design methodologies in Hardware/Software Codesign [96].

Previous work at UT focused on the use of Khoros, a graphical design environment primarily used for image processing applications. The CHAMPION program used Khoros to capture algorithms, and then provided a set of VHDL libraries to map the application to a reconfigurable computing board with one or more FPGAs. A partitioning algorithm automatically broke the application into pieces that were then mapped to the collection of available FPGAs. Using this approach, speedups of up to 2000x were achieved in mapping applications to FPGAs.

MATLAB is another tool for capturing algorithms and is widely used. We discuss the interfaces provided with MATLAB for using parallel processing and how we can develop the interface between a processor running a MATLAB program and the FPGAs we would like to use to accelerate portions of the MATLAB problem. First, we discuss some issues related to computational models and how we are investigating the models of computation one may wish to consider for HPRC applications.

4.2 MODELS OF COMPUTATION

Ptolemy is a software framework developed at the University of California, Berkeley and is used for modeling, simulation and design of concurrent, real-time embedded systems [97,98,99,100]. The advantage of this tool is that it allows heterogeneous mixing of different “models of computation”. A model of computation varies from another mainly in its notion of “time”. Ptolemy II is a JAVA-based component-assembly framework and its GUI (Graphical User Interface) allows the user to create designs that involve one or more interacting components (which could be of different models of computation).

“Models of computation” are architectural patterns, which focus on relationships between concurrent or sequential components. Ptolemy II includes a suite of *domains*, each of which realizes a model of computation. It also includes a component library, in which most components are *domain polymorphic*, in that they can operate in several of the domains. Most are also *data polymorphic*, in that they operate on several data types. The domains that have been implemented are listed below.

CT: continuous-time modeling,

DE: discrete-event modeling,

FSM: finite state machines,

PN: process networks,

SDF: synchronous dataflow

CSP: communicating sequential processes, (only in the full release)

DDE: distributed discrete events (experimental),

DT: discrete time, (experimental),

Giotto: periodic time-driven (experimental) and

GR: 3-D graphics (experimental).

SR: Synchronous Reactive (experimental).

TM: Timed Multitasking (experimental).

The first aim of the research into computational models is to demonstrate the feasibility of using HPRC for the applications/computational models chosen. For each computational model, the process of dividing tasks between the processors and the reconfigurable units is done by hand. In so doing, a better understanding of the types of applications and the implementation approaches that are most appropriate for HPRC/cognitive architectures is achieved. This insight is critical to the ultimate goal of finding a way to efficiently automate the process while achieving high performance. The applications we consider for cognitive architectures include different computational models, so these demonstration applications help in understanding which computational models are most appropriate for cognitive architectural platforms. Some comments are included in the application discussion below concerning their computational model and the effectiveness in mapping the application/computational model to HPRC/CA. With emerging architectures replacing the von Neumann computing model, there remains a need for a robust computational model for these architectures to enable reasoning about programs and correctness, enable compiler transformations/optimizations, and similar tasks. Such work is ongoing.

4.3 PARALLEL PROGRAMMING

The Grid is the name given for the vast collection of interconnected computers distributed throughout the world, combined with the software tools and infrastructure required to develop and execute applications on these computational resources. The related development of Cloud computing shares many of the characteristics of grid computing. A wide variety of tools provide support for job scheduling, including such tools as Condor, Utopia, and LSF. Other tools such as the Internet Backplane Protocol for Logistical Computing help in caching data for distributed grid applications. Distributed operating system work, including the Globus project, seek to exploit grid resources as well by allowing tasks to be spawned off to other machines and migrated as necessary.

The programming infrastructure to support communicating processes executing on the grid is a quite difficult problem. Early efforts to provide these capabilities include PVM [79] and MPI [78], which are libraries of communications and control functions to coordinate processes on the grid. PVM is quite popular and widely distributed, with support for a wide variety of architectures. It supports a dynamic process model, hence tasks can be created or destroyed at runtime. In contrast, MPI supports a CSP-like computational model in which tasks continue for the entire life of the distributed program. OpenMP [101] provides shared memory programming support for multiprocessing. Similarly, POSIX threads (pthreads) and Threaded Building Blocks are other popular programming libraries for concurrent programming. MPI is the most common communications library used for parallel processing. MPI is used in our work for parallel applications.

The NetSolve project was developed at the University of Tennessee's Innovative Computing Laboratory. NetSolve is a client-server system that enables users to solve complex scientific problems remotely. The system allows users to access both hardware and software computational resources distributed across a network. NetSolve searches for computational resources on a network, chooses the best one available and using retry for fault-tolerance, solves a problem and returns the answers to the user. A load-balancing policy is used by the NetSolve system to ensure good performance by enabling the system to use the computational resources available as efficiently as possible.

As part of the DARPA High Productivity Computing Systems program, a number of new languages for large-scale parallel processing are in development. These include Cray’s Chapel, IBM’s X10, and Sun’s Fortress. Chapel and X10 continue development under the auspices of the HPCS program, but finding for Chapel has ended, so compiler development and runtime support is not as advanced as the others. These languages provide interesting features, but it is too early to determine their success in adoption by programmers.

4.4 GENERAL PURPOSE PROGRAMMING FOR GRAPHICAL PROCESSING UNITS

The multi-core paradigm has become the primary method for providing performance improvements in processors. As Instruction Level Parallelism (ILP) has begun to yield diminishing returns on transistor investments [102], multiple copies of processors (cores) are being placed onto a single die to exploit Thread Level Parallelism (TLP). While desktop users see benefit in a multi-tasking environment, single-threaded applications must often be recoded to take advantage of the multiple processors appearing in desktops. This presents a new set of challenges in algorithm design; programs must now account for parallelism and the pitfalls that come with it. In a different context, Graphics Processing Units (GPUs) have become programmable. Traditionally, GPUs had fixed pipelines for assembling vertices, mapping textures, and rasterizing scenes to create 3D environments for games and CAD applications. To allow for more complex effects in rendering, such as bump mapping and complex lighting, vendors replaced the fixed pipeline with programmable shader units. In doing so, they allowed the high bandwidth memory and parallel shader units to be used for general purpose computing [103].

Specifically, we compare the streaming model behind AMD's Brook+ and Computer Abstraction Layer (CAL) development environments and compare this to C, a traditional sequential language. To do this, the grid potential computation used in Ab Initio modeling serves as a memory bound application for exploration. This paper provides qualitative assessments of programmability, such as ease of use, expressability, and how to optimize an application written in these languages. Additionally, quantitative performance results are given for naïve and optimized kernels. These results are compared to what is theoretically achievable on these architectures.

AMD's GPUs use a hierarchy of processors to provide massive parallelism. This arrangement is shown in Figure 19. Stream processors are divided into SIMD engines. Each SIMD engine runs a number of threads concurrently on its thread processors. These threads are grouped into a *wavefront*. Each SIMD engine can time slice execution of multiple wavefronts. Within a wavefront, a number of threads execute concurrently. Four threads are multiplexed per thread processor to hide memory latencies [104]. Finally, each thread processor contains 5 stream cores, which serve as the ALUs that perform actual computation.

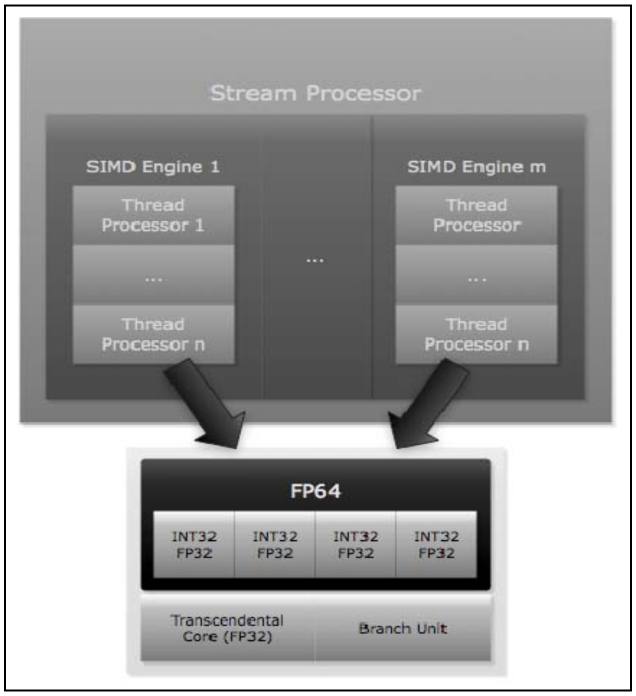


Figure 19: AMD FPU Architecture

Brook+ is a high level stream computing language developed by AMD. It is based on the Brook project at Stanford University [105]. Brook+ is a C-like programming language using kernels running on the GPU in conjunction with host-side code written in C. Kernels are defined using the `kernel` keyword. Top-level kernels operate on streams while other kernels can be used to modularize code. If a kernel is top-level, it can read from and write to streams, but cannot return data. Kernels that are not top level are inlined at compile time and cannot operate on streams. Instead, they serve as functions to return a result based on some inputs. Kernels may not call regular functions, though functions can call kernels. When running code in Brook+, the domain of execution is defined by default to be the size of a kernel's output stream(s). The developer can manually change this if needed. Kernel instances are created for each point in the domain of execution and may write only to the instance's location in an output stream (shown in Figure 20). The streaming model provides implicit parallelism and separates communication from computation [106]. Depending on how an input stream is declared, a kernel may read from any location or only the current domain instance location. Streams can be declared as input or output; input streams can only be read from while output streams can only be written to [104].

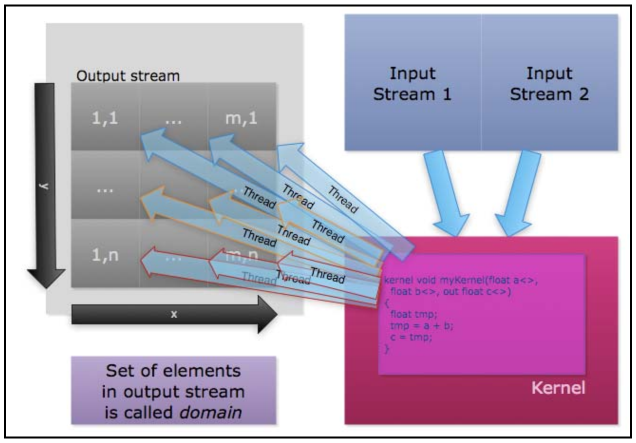

Output stream
1,1
...
m,1
1,n
m,n
y
x
Thread
Input Stream 1
Input Stream 2
kernel void myKernel(float a<>, float b<>, out float c<>)
{
float tmp;
tmp = a + b;
c = tmp;
}
Kernel
Set of elements in output stream is called *domain*

Figure 20: Streaming Model

The Compute Abstraction Layer (CAL) is a low-level streaming environment for performing computation. Like Brook+, it uses a streaming model for processing data. However, CAL has additional constructs that can be used in kernels. Shared registers are accessible by all threads in a wavefront (rather than being accessible by only a single thread). In addition to reading and writing to streams, the global buffer can be used in computation. The global buffer is 128-bit addressed and can be read or written to by any thread at any index [104].

CAL is divided into two components: the CAL runtime and Intermediate Language (IL). The runtime serves as the front end for managing streams, devices, kernels, contexts, hardware counters, and kernel compilation. The runtime is implemented as a library of C functions. For handling streams, a number of options exist. A stream can be allocated locally (on the GPU) or remotely (on the host). In addition to these, a feature exists allowing a stream to be allocated on the host in any address specified. This contrasts with remote allocation, which returns an address the runtime creates. Using this feature, data can be written directly into a buffer without needing to be copied. Local allocations are limited by the amount of memory on the GPU, remote allocations are limited to 64MB, and pinning memory is limited to 16MB (as of SDK v1.3 for Linux).

Streaming languages provide implicit parallelism. Since the kernel is invoked once for each element in the domain of execution, developers need only express the computation of a single element. Traditional sequential languages, like C, require users to explicitly define the parallelism granularity and patterns. This is often a difficult task, even with libraries like OpenMP [101]. Brook+ and CAL are simpler than C in this respect, since threading is handled by the hardware. Additionally, Brook+ and CAL hide architectural details, as opposed to Nvidia’s CUDA (Computer Unified Device Architecture) [107], which requires using shared memory and efficiently mapping threads to achieve high performance. In some cases, this may prevent the developer from fully exploiting hardware capabilities, but this does not affect the application presented in this paper. Brook+ is simpler than CAL for practical reasons; it hides API calls into elegant class abstractions while CAL requires numerous C API calls to perform operations. Both CAL and Brook+ presently lack double precision transcendental operations (as of SDK v1.4).

We created naïve and optimized implementations of the grid potential kernel in Brook+, CAL, and C. The naïve C algorithm is given in Figure 21. The optimizations applied to each kernel are given in Table 3. Kernel unrolling refers to having the kernel write to eight streams instead of one. This increases kernel efficiency [104]. SIMD instructions are exploited in this application by using Intel's Math Kernel Library (MKL) vector routines on the host machine and float4 data types on the GPU. Precomputing the radii eliminates redundant computation. Its performance benefit in Brook+ was very negligible and actually hurt performance in CAL. Cache blocking is done on the host machine to maximize L1 cache reuse in the inner loop. Finally, the host code was parallelized using an OpenMP parallel for construct with guided scheduling.

```
for(i = 0; i < npt; i++)
{
     float r2 =x[i]*x[i]+y[i]*y[i]+z[i]*z[i];
     for(j = 0; j < nbas; j++)
     {
          gridpot[j*npt+i] =
          exp(alpha[j] * r2);
     }
}
```

Figure 21: Naive Grid Potential Kernel

Table 3: Kernel Optimizations Applied

Optimization	C (x86)	Brook+	CAL
Kernel unrolling	No	Yes	Yes
SIMD	Yes (Intel Math Kernel Library)	Yes	Yes
Precompute radii	Yes	Yes	No
Cache alphas	Yes	No	No
Cache Blocking	Yes	No	No

Performance results shown in Table 4 highlight the performance advantage of AMD's GPU over the multi-core CPU, despite a simpler programming model. The C implementations were run on an 8 core X5355 machine. Using the time to compute the exponential function in MKL and an optimized STREAM copy benchmark, we computed the optimized performance to be 74% of what is theoretically achievable on this machine. GPU results were taken on a Firesream 9170 with SDK version 1.3. The optimized CAL implementation used 80% of the Firestream's 51.2GB/s of memory bandwidth (and hence achieved 80% of the theoretical performance). Brook+, while outperforming the CPU even in the naïve implementation, suffered from compiler overhead not present in the CAL version. The short nature of this kernel prevented Brook+ from amortizing this overhead. All results given are for single precision computation.

Table 4: Platform Performance of Naive and Optimized Kernels

Billions of points per second	Naïve	Optimized
C	0.02	0.86
CAL	3.05	10.19
Brook+	0.87	2.66

We have shown that the streaming model simplifies computation without sacrificing performance in the grid potential application. Using CAL, we were able to achieve 80% of the cards computational capabilities. The simplicity of Brook+ and CAL programming languages provide large speedups with little effort. The streaming model eliminates the need to explicitly parallelize applications. By far, the most complicated optimizations lied in the C implementation, which required complex cache blocking and leveraging libraries to exploit parallelism (SIMD and thread). While CAL is conceptually simple, its driver level API makes programming more difficult than Brook+.

Compute Unified Device Architecture (CUDA) is a programming environment provided by NVIDIA for general-purpose computation on their GeForce, Quadro, and Tesla GPUs [107]. The kernel offloaded to the GPU is specified as a computational grid of blocks of threads. The threads are grouped into warps, which use Single Instruction Multiple Data (SIMD) for instruction execution. We found for a computational chemistry application that Brook+ provided the better performance than CUDA, partially because there was better library support.

4.5 RECONFIGURABLE COMPUTING DEVELOPMENT

Reconfigurable computing comes from a broader application of programmable logic device technology. Programmable logic devices provide designers with the ability to modify circuit functionality after its fabrication. Examples of programmable logic devices include programmable arrays of logic (PALs), complex programmable logic devices (CPLDs), field programmable system level integrated circuits (FPSLICs), and field programmable gate arrays (FPGAs), with FPGAs the most commonly used devices for reconfigurable computing.

Currently, reconfigurable computing systems are typically comprised of processing nodes associated with co-processing boards with FPGAs. This co-processing model uses software executing on the processor for general computation and control, with the reconfigurable hardware accelerating computational intensive operations. Following some promising research results, the emerging system on a chip market will likely provide SoCs with configurable logic.

Computing systems, particularly those employed in embedded applications, are increasingly benefiting from the use of programmable logic. Many developers use programmable logic devices (PLDs) to rapidly develop customized circuitry for various applications, and then ship the systems with the configured programmable logic devices. FPGAs used to be widely used for system prototyping or implementing glue logic on boards. The reduced device costs, increased capacities, faster operational speeds, and shrinking product time to market requirements now result in the common deployment of FPGAs in embedded systems as a substitute for ASICs. Such systems typically have a static configuration throughout their lifetime.

In some cases, the configurations of the programmable logic devices can be updated after the system is fielded, resulting in a configurable computing system. This ability to configure a system can reduce maintenance costs and extend product lifetimes via system upgrades in the field accomplished by updated configurations. In contrast, reconfigurable computing systems frequently change the programming of the logic devices during operation. Recent developments in programmable logic device technology (density, speed) have made reconfigurable computing practical.

Reconfigurable computers provide the user with the ability to dynamically change the logical operation of its computational elements. Configurable computers also provide the ability to change the logical operation of computational elements, but in a more constrained manner.

Reconfigurable computing promises to provide a wide variety of benefits to system designers. A primary reason for the initial use of programmable logic devices is the relative ease in employing them in designs. Configuration requires seconds to perform, as opposed to the weeks required for fabricating an ASIC. If there are errors in the logic implementation, attractive system improvements, or changes to the specification, designers can make changes as necessary before the system is deployed. These benefits are equally applicable for configurable and reconfigurable systems.

The use of configurable or reconfigurable systems provides similar benefits in fixing errors, adding features, or providing other updates to systems throughout their life cycle. By modifying the functionality with a new configuration of the programmable logic, minimal update costs may be required. If one combines this model of providing upgrades or bug fixes with internet-connected configurable systems, radical improvements in the ability to maintain embedded systems become possible. Because reconfigurable computing provides this capability to modify system functionality to fix bugs or add features, cost reductions during design as well as after deployment result in significantly reduced life cycle costs. For products with long lifetimes, this cost reduction can be substantial.

The use of programmable logic results in accelerated development cycles yielding a reduced time to market. By avoiding the extended design and fabrication times required for ASICs, designers can reach the market more quickly. If design flaws or engineering change orders arise near the end of the design process, modifications to programmable logic can usually be implemented quickly with the existing parts. In the case of an ASIC-based design, a new version of the ASIC will typically be required, resulting in additional costs and delays. By avoiding these additional costs and delays, reconfigurable computing systems can reach the market more quickly. Because the timeliness in reaching the market often determines if a product is profitable or not one cannot overstate the importance of the schedule risk mitigation due to reconfigurable hardware.

The flexibility coming from the ability to reconfigure components of a system based on environmental or operational conditions enables systems to be fielded for a broad range of applications and for dynamic or unknown environments. A single reconfigurable computing system can be applied to image processing, signal processing, cryptographic, and string matching problems through reconfigurations. An ASIC-based hardware approach would require a large investment in hardware resources while a software solution may not achieve the require performance. In this context, the flexibility of reconfigurable computing provides unmatched capabilities.

By optimizing the hardware to address the specific task at hand, reconfigurable computing platforms can often perform operations using much less power than general purpose hardware or software based solutions. Even custom or specialized hardware may not provide the power savings obtainable from customizing the hardware used to the specific application. For example, in a number of signal and image processing applications, variable word sizes can be implemented with programmable logic to provide sufficient accuracy but no more. Reconfigurable computers can reduce the number of gates used in the design by optimizing word sizes as needed, whereas digital signal processors, microcontrollers, or general purpose processors must be designed with a static word size that cannot be changed to reduce power.

In our work, we found that solving systems of equations can be accelerated by employing LU decomposition and lower precision processing elements with RC devices to find an approximate answer, followed by iterative refinement in double precision on a host CPU. By reducing the precision used, the execution is accelerated, but at the potential cost of additional iterations to refine the solution or even failure to converge. Hence, a complicated optimization space results. This work was awarded first prize at the student research competition at *Supercomputing* 2007 and was awarded 3rd place in ACM's grand finals of the student research competition (compared to all other research competition winners) [108]. Similarly, FPGA implementation of fixed-point engines for quantum Monte Carlo (MC) simulations depends on careful selection of pipeline structure and data precision. Another student won the student research competition at *Supercomputing* 2008 for this work [109].

Reconfigurable computers can support a very large number of gates through the use of multiple configurations. In effect, a given set of programmable logic devices can replace a much larger collection of ASICs. This will result in a reduction in the physical size required for the electronics comprising embedded systems.

Programmable logic devices provide an ideal capability to detect, identify, and isolate faulty hardware elements in embedded systems. With the ability to support testing configurations and to adapt to detected faults, reconfigurable computers can support a variety of critical applications. By enabling graceful system degradation in the presence of faults, much cheaper and more reliable critical systems can be fielded.

Last, and certainly not least, designers can achieve higher performance by using reconfigurable computing to adapt to the environmental conditions during operation.

In one recent cryptography example, engineers achieved higher performance with an FPGA-based reconfigurable computing solution than available with an ASIC developed for the same task after a long and expensive development [110].

With reconfigurable computing systems, key attributes that determine its capabilities include the speed and granularity of reconfiguration. The ability to reconfigure programmable logic quickly makes it practical to change configurations more frequently during operation. The technology used in the programmable logic device dictates the granularity of reconfiguration that can most effectively be supported. The ability to selectively reconfigure parts of the programmable logic is known as fine-grain reconfiguration. Many families of programmable logic devices do not support the random access and modification necessary to support fine-grain reconfiguration. For these parts, most or all of the part must be reconfigured at the same time. Because the time to configure an entire part takes a large number of cycles, the performance penalty associated with such reconfigurations renders it impossible to support frequent function modifications while still attaining high performance. Hence, reconfigurable systems assembled from such parts have a coarse granularity of reconfiguration. In contrast, some newer generations of programmable logic devices support the fine-grain reconfigurability that enables hardware optimization based on the operations and data at hand.

Reconfigurable computers must include runtime systems to support the configuration data for the programmable logic device elements, much like a processor uses a sequence of instructions. For systems with slower, coarse-grain configuration, the programmable logic devices may be updated when each new application is run, or during changes between operational phases. As the frequency of reconfigurations increases and their granularity decreases, the runtime support required becomes more sophisticated. Memory banks or configuration caches may hold the necessary configuration data, as with instruction and data caches. Although in principal, dynamic optimization of configurations is possible via the runtime system, in practice current limitations in the design tools currently make it impractical. Work with the Xilinx Jbits [111] interface is improving the runtime and design environment to make such systems practical in the future. This research does not directly address runtime reconfiguration or configuration caching, but the performance modeling framework is intended to support these capabilities with little, if any, modification.

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 co-design 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 (Register Transfer Language), 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 (HDL) 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 (see http://www.jhdl.org). 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.

A number of challenges remain for developers using reconfigurable computers. The verification of RC systems faces all the challenges of ASIC verification combined with the additional complexity of multiple configurations. Fine-grain adaptation of configurations provides a higher bar for systems verification.

Design languages, tools, and methodologies for RC systems continue to be a topic of research and development. The same productivity gap between available gates and designed gates currently hindering ASIC designers affects RC designers as well. Tools that support more abstract design while automatically extracting a problem's parallelism and the most appropriate configurations are needed. Addressing this problem is the focus of this research.

For software engineers, the notion of an instruction set architecture greatly reduces the difficulty in developing applications. This abstraction of a processor provides sufficient insight into the hardware to enable its effect use without overwhelming the programmer. Currently, reconfigurable computers lack such a standard "programmable configuration architecture" to serve a role like that of an instruction set architecture. This lack of a programmable configuration architecture could pose significant portability problems for the industry in the future.

Understanding the impact of architectural changes is just beginning; the changes to models of computation are not well understood yet. This remains as an exciting area of basic research with wide-reaching implications. For example, just as the notion of virtual memory has resulted in significant improvements in software, the notion of the virtualization of hardware will provide similar benefits for hardware design.

4.6 SECURITY SUPPORT FOR COGNITIVE COMPUTING ARCHITECTURES

At the same time, reconfigurable computing provides the possibility to provide more secure computation. By providing functionality using reconfigurable circuitry, we can create customized systems with better security and reliability. We first consider work pertaining to reliability and fault tolerance and how it can apply to creating secure systems.

Significant prior work has been focused on FPGA testing, techniques for building fault tolerant systems using redundancy and voting, and related work for achieving reliability. Much of this prior work can be easily leveraged to improve the robustness of our systems (including security and reliability). For example, configurations can be included that provide testing for some or all of the slices of the device. By periodically performing tests, the parts of the FPGA device that are inoperable can be identified and potentially avoided. Similarly, redundant functionality can be easily mapped to the device along with voting circuitry in the platform to provide fault tolerant operation. Scheduling execution with different portions of the FPGA (slices) over time can provide temporal redundancy as well. Providing functional redundancy by a diversity of implementations of accelerated operations on sandboxes can also be easily employed with the approach herein. What is particularly promising for this approach with reconfigurable devices is the potential for tracking errors in devices and scheduling around defects. In this effort we did not implement dynamic generation of bitstreams, but considered requirements for this capability and explored how this could be achieved. Finally, previous work in performance modeling for parallel and HPRC systems was adapted to explore predictions related to reliability for large numbers of FPGAs, processors, and accelerated operations [112,113].

To create more robust systems, we explored the ability of reconfigurable computing systems to provide application-specific functionality on demand. Although the hardware devices already provide much of what we need to achieve this, the overall system must be modified to provide the services to achieve robustness. In particular, what is needed is a runtime system that works with the reconfigurable hardware to provide the user applications with the desired functionality and performance, but with robustness and flexibility.

A runtime system that supports a set of hardware configurations was explored first, with a set of configurations statically defined. Although this conservative approach does not initially allow support of aggressive (partial) runtime reconfiguration, current tool and language limitations have prevented the adoption of this approach, so it should have little practical impact on DoD systems. The runtime system includes the development of "sandboxes" for each collection of logic mapped to the reconfigurable system, where the sandboxes provide isolation to preclude interactions between the different processes. Note that soft processor cores with applications software and data can exist within a sandbox, not only digital logic. The runtime system must interact with each of the sandboxes to support secure I/O, interrupts, and similar interfaces. The runtime system must also interact with the operating system to schedule the execution of configurations on the reconfigurable hardware (with some consideration of meeting real-time schedules considered, but not implemented here). A primary issue for each of these tasks is providing the right set of abstractions for the configurations and their services to the runtime system/operating system and to the applications developer.

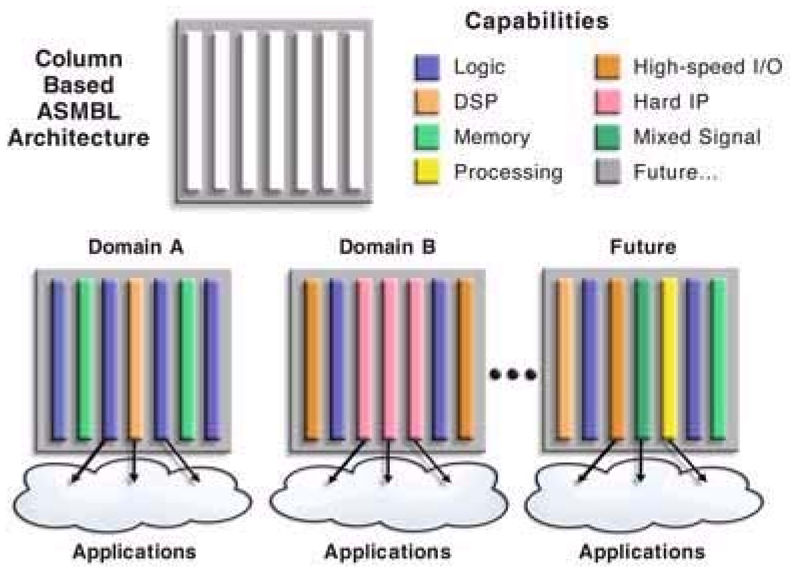


Figure 22: Column-Based Advanced FPGA Architecture (from www.Xilinx.com)

The requirements for runtime support of a reliable, secure, reconfigurable system are complex. For example, reconfigurable hardware can provide the ability to change functionality, but effective means of managing the potential configurations and minimizing resource fragmentation is a challenge. The RS3 (Runtime support for Reliable , Secure, Reconfigurable systems) runtime system and scheduler must account for the potential problems related to fragmentation in order to work. Similarly, effective scheduling algorithms must be explored, keeping in mind theoretic constraints [114,115]. The effort focused on Xilinx Virtex FPGAs, as shown in Figure 22. These

FPGA families use a “slice” based architecture that enable convenient analysis of resource usage, flexible support for dynamic use, inclusion of reliability, and consideration of separation for sandboxes. Operating system and middleware support for the dynamic scheduling of tasks to the FPGA devices entail additional requirements for the runtime system. Such requirements include both software and hardware issues. For example, the software requirements include standard requirements related to secure operating system management of resources as found with other types of devices and secure systems. Additional requirements include the ability for the operating system to allow multiple applications to share the FPGA resources in a controlled, correct manner. Such support levies requirements on the hardware, specifically including the platform and the sandboxes, so that signals, interrupts, memory, and I/O between the host processor, the platform, the sandboxes, and the accelerated operations.

To pursue this research, we first defined a base hardware platform for consideration (we considered using a multicore processor and FPGA system such as from DRC, but worked instead with the Cray XD1 and Xilinx University Platform, XUP). We abstracted of this platform to generalize the results. Given the hardware platform, we used a Linux kernel to allow easy modification. Although we did consider real-time operating system issues as well, this research primarily focused on desktop environments initially. See [123] for a good overview of operating system and middleware issues related to secure systems. The primary extensions required related to scheduling a mix of accelerated operations on the reconfigurable hardware, dynamically configuring the reconfigurable logic devices as needed, and managing the communications between processes and the accelerated operations in a secure and reliable manner. For example, a process associated with a specific accelerated operation can interact with that portion of the reconfigurable hardware, but not with other portions not associated with that process. The operating system services must ensure that no illicit communications can occur that disseminated sensitive information and that no process can corrupt or modify the execution of another process. In order to constrain the problem, this effort only targeted a set of statically predefined set of legal configurations (bitstreams) for the FPGAs. These configurations used “sandboxes” to provide encapsulation and isolation via a flexible interface.

4.6.1 SANDBOXES

An application requests services from the operating system in order to use the reconfigurable processing elements. These services include configuration management and scheduling, I/O support, exception handling, and encryption support. The operating system services then schedules the configuration for execution within a sandbox selected from a set of sandbox templates. The sandbox templates include different sizes and aspect ratios of combinational logic blocks or slices. The sandbox templates support different security levels; memory requirements for distributed registers, embedded memory, or portions of external memory address space; and I/O requirements for encryption, bus widths, and data rates. The operating system then manages the execution of various configurations for accelerated operations with a set of sandboxes that provide high performance while ensuring the isolation required for security. The platform then manages the communications between the host processor and each of the sandboxes containing an accelerated operation.

Often times FPGAs are used for rapid prototyping of application specific integrated circuits that are to be fabricated. In order to reprogram one, a circuit layout must be converted into a bitstream that details the FPGA functionality by dictating the function and interconnections of primitive elements. This is similar to a C++ program being turned into machine language for a CPU. The majority of FPGA implementations are static and require the system to be completely erased before modifications can be loaded. However, some FPGAs allow for dynamic circuit modification to enable systems that are re-programmable at runtime. These systems are known as partially reconfigurable circuits. They are a very powerful type of circuit that results in heightened security, higher performance, lower circuit complexity, and a flexibility that is similar to software [116]. Other notable benefits include increased function density, reduced power dissipation [117], and a smaller required area for the entire circuit, which results in cost savings. Unfortunately, all of these benefits come at the cost of implementation complexity. The first hurdle is that the implementation requires bitstreams to be pre-synthesized and loaded into memory. Then, a controller needs to be accessed to load the partial bitstreams. The nature of this pre-synthesis opens up all kinds of manual floor-planning requirements. Next, the methodology for implementing a PRC (Partially Reconfigurable Circuit) is quite complicated. It requires the insertion of bus macros between modules, specific synthesis guidelines to generate the partially reconfigured netlist, proper floor-planning and placement of the bus macros, and adherence to specific PRC design rules; all of which require meticulous implementation in order to ensure success [118]. Due to design tool limitations and the lack of design flows, PRCs have, for the most part, remained in the realm of research. Existing tools require designers to perform many extra design steps to manipulate primitive routing and function elements. Many of these steps such as designing special connection points that must be manually inserted between static and dynamic regions [119], significantly increase the complexity of the implementation. Because of this, many designers have not had the proper guidance and experience to make PRCs practical [120].

In order to simplify this complexity and gain additional design security benefits, we developed a method that implements an array of generic modules known as sandboxes. Rather than modifying the bitstream for circuit reconfiguration, the connections between different sandboxes are modified, which essentially build the circuit at run time. This concept of reconfiguration at a more abstract level allows for security levels far higher than were previously possible. Areas where this technique and heightened security may be applicable include data cryptography, pattern recognition, power controlling, time division multiple access applications, and data manipulation that requires extreme security. Here we detail the proposed approach from the smallest element, the sandbox, up through the design flow. An explanation of which methods can be implemented to dynamically create fast, reliable, and secure hardware functions will also be discussed.

In this system the sandbox is the smallest element of a larger design. The sandbox is a generic isolated module that can perform any one function, of a set of simple predefined functions. A simple sandbox that performs some basic functions can be seen in Figure 23 below:

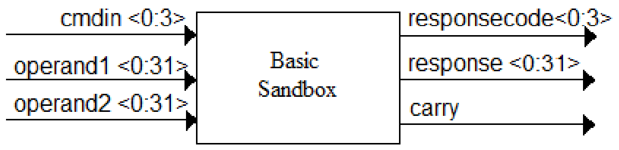


Figure 23: Simple Sandbox for Basic Arithmetic Operations

A sandbox such as the one above might be capable of executing different operations such as add, subtract, shift left, shift right, and, nand, or, xor, nor, or more complex operations.

Combining lots of these basic sandboxes together, with different operations on each, allows for the creation of a near limitless number of circuits without modifying the bitstream. The idea is to essentially design an abstraction layer to be placed over the FPGA, specifically for reconfiguration, that allows for circuit modifications without modifying the actual hardware. Such a system would allow for mobile hardware processes [121]. A connection description could be passed to the FPGA along with initial parameters. After the function is completed it can destroy existing connections. The end result is a method of true virtual hardware.

The emerging paradigm of providing a large number of processor cores on a die, combined with aggressive adoption of virtualization support, presents challenges for fielding systems with effective security. Similarly, soft processor cores as well as accelerated functions included in reconfigurable logic will need to support multiple processes that may have different levels of security or require immunity from tampering. Different models of security and associated policies have been the focus of research for decades in order to provide a hierarchy of security levels and to compartmentalize information in a safe manner [122,123].

The sandbox model relies on a hierarchical design to allow easy controllability. Connection bookkeeping is handled by a sandbox controller that has many of these sandboxes attached to it. The controller contains a port look up table and memory to store state information for the hardware process. See Figure 24.

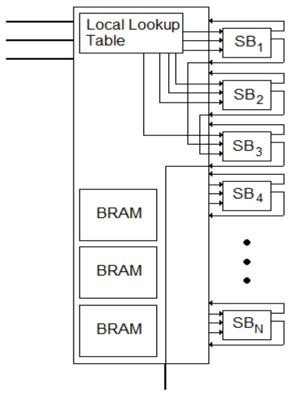


Figure 24: Sandbox Controller with Block RAMs

A hardware process is implemented on the FPGA through the following flow. First a computer system software process sends a request to the daemon running on the machine, which controls access to the FPGA. The daemon sends the hardware configuration plus required inputs to the logic board. Next the FPGA takes the hardware configuration, decodes it, and sends it to the OTF (On-The-Fly) Layout Planner. This module consults the resource manager to determine which sandbox controllers are currently busy. From here the layout is determined and the inputs and local layout configurations are passed to the sandbox controllers that are needed for the process. The sandbox controllers set their connections and execute the task based on the inputs. A response is sent back to the daemon and relayed to the requesting software process. The connections on the system are then released and ready for the next operation.

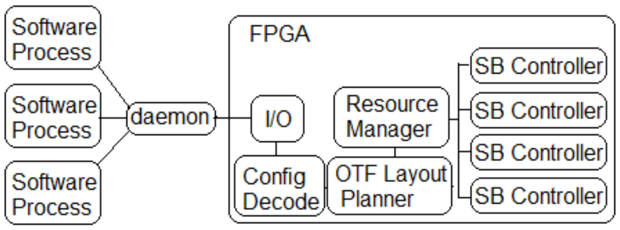


Figure 25: Sandbox Model System

The expected benefits from this design include the benefits of dynamic hardware systems with faster reconfiguration loading due to no hardware being changed (e.g., no new bitstream loaded), and increased security which will be discussed below.

The driving motive that led to this new run-time reconfiguration implementation was the need for secure systems. Next we will discuss the passive and active security features of this implementation. The most notable passive security feature is that any individual or group attempting to tamper with the circuit to obtain valued information would not see the true circuit. Other security features include encrypted bitstreams, sensitivity control, and quick connection flushing.

One of the major security features that the sandbox approach takes advantage of is the anti-tamper protection. Any individual who has the resources to extract the bitstream would have to decrypt it—as FPGAs often allow their bitstreams to have triple DES protection—and then reverse engineer it. Assuming they were able to reverse engineer the circuit they would not have any valuable information. The only thing that would be visible would be the unconnected sandboxes. Essentially, there is no hardware signature. This is due to the fact that the design is created and destroyed at run-time. The hardware process is created through the configuration, it executes, and then it dies through resetting the sandbox interconnections.

Another prominent security feature of partially reconfigurable FPGAs is that they contain SRAM controlled configurations. This means that the pre-synthesized bitstreams are held in volatile memory waiting to be swapped in at a nanosecond's notice. The benefit of this is that when the system is powered off, the circuit function disappears, which makes tampering much more difficult. Moreover, the functions built out of tiled sandboxes require dynamic instructions which are also held in volatile storage, further protecting the design from tampering.

Next, several active security features can be added to the system to further increase the overall safety. The connection configuration and data that is passed to the FPGA can be encrypted before it is sent and then decrypted when received on the FPGA. Also, depending on the size and security requirements of the design, the FPGA can be split into different self-contained areas each of which contains different levels of security. This will allow multiple users, with different security requirements, to use the PRC elements, in a secure seamless manner. The regions in which the processes are run can be separated based on classification level or other measures. On very large systems or systems that require even further safety, these different regions may be implemented on completely different FPGA boards.

An additional security feature that this approach makes easy to implement is a circuit self-destruct feature. A button or command could be added that instantly flushes all sandbox connections. This would destroy the circuit leaving it empty for anyone who tries to tamper with it later. One final security advantage to point out is that by design, sandboxes provide protection against many common attacks such as buffer overflows. All of these different passive and active security features make the sandbox model ideal for situations where security is of the utmost importance.

4.6.2 Checkpointing and Recovery

In order to support reconfiguration of some or all of an FPGA, one must determine what happens to any state associated with a previous configuration. In the case of the sandbox system discussed above, the functions in each sandbox can be memoryless (without side effects), so that swapping different sets of sandboxes in and out of an FPGA can be easily accomplished. On the other hand, if the sandboxes contain state that must be preserved for later function calls, then this greatly impacts our ability to reuse the sandbox. Similarly, if the sandbox control circuitry contains state that must be preserved (we expect this to be the normal situation), then there is a need to save this control state before reconfiguring the FPGA to support different sandbox configurations. To address this need, we developed an approach for checkpointing the state associated with an FPGA and enabling it to be recovered or restored after reconfiguration is complete.

In order to support this saving of state, we must have some storage that is not affected by the FPGA reconfiguration. We assume that this is RAM or some other memory located logically near the FPGA (perhaps on the same board). This storage may or may not be volatile, as long as it is not erased after a checkpoint. Next, we must annotate which memory block(s) we wish to checkpoint. We must also provide some controller whose responsibility it is to manage the transfer of this data from the memory block(s) to the external storage during a checkpoint. This controller could be implemented much like a TAP (Test Access Port) controller for IEEE 1149.1 boundary scan support, or it could be implemented to transfer data during execution, but to prevent incoherent data from being saved, it is best to freeze the function of a sandbox while its data is being checkpointed.

Each block of data is stored in memory using a directory scheme akin to the inodes used with Unix for managing files. In the case of the sandbox controller, it has responsibility for reloading the state of each sandbox that will be restored (we assume a flag is kept for each of them pertaining to whether or not they will be restored). When a new configuration is loaded, the checkpoint and recovery controller will take over from initialization and grab the equivalent of a bootstrap from a specific external memory location. It will then look up in this location the bootstrapping information for the controller (e.g., how many memory blocks, their size, location, etc.). Using this information it will address the required data, read it from external memory, and load it into the locations on the FPGA. Similarly, the checkpoint and recovery controller will work with the sandbox controller to transfer memory blocks from checkpointed sandboxes back to their memory location (which may be on a physically different part of the FPGA, or perhaps on a different FPGA in more complex systems).

To accomplish this functionality, one needs to provide a checkpoint and recovery controller. One means of doing this is to use a built-in PowerPC processor in many Xilinx FPGAs or perhaps a MicroBlaze or NIOS soft core. The controller must be able to access the memory to be saved. If sandboxes have state to be saved, they must provide access to the controller by standard design interfaces (this could be similar to scan chains of registers or paths to block RAMs).

Using this approach, one can save some or all of the critical state of an FPGA and restore it after being reconfigured. In the case of the sandboxes, one might wish to change the number or specific mix of types of sandboxes. By using this approach, one can save the state of interest, reconfigure the FPGA as needed, restore state, and then continue operation.

An idea we have not pursued is the potential to have some sandbox state loaded from biometrics, secure ID, or other input that could be used to authenticate users and restore specific capabilities when they use the secure computing system.

4.6.3 SANDBOXES SUMMARY

A simple prototype has been developed to act as a proof of concept. It implements the generic sandbox shown in Figure 25. With this proof of concept completed, we are investigating different sandboxes to create operational flexibility and to serve as a framework to maximize performance, reliability, security, and re-usability. A good tool to test the performance and reliability, is to run comparative analysis against static circuits with the same functionality. A verification metric would be suitable for future work as well to ensure reliability. As it stands, verifying a circuit model that constantly changes could be quite challenging. Creating a built in self-verification for dynamically loaded circuits would be a very cumbersome task, but one that needs to be explored. Our extensions of this work will address these issues.

The flexibility and performance of FPGAs have made them ideal for the rapid prototyping of custom ASIC (application specific integrated circuit) designs. One of the most intriguing research areas has been the practical application of dynamic partial reconfigurable FPGAs. These dynamic configurations have many benefits and limitations. They provide better performance and flexibility while taking up less area, which is ideal for ASICs. However, their limitations include lack of design tools and intensive manual floor planning. This is also assuming that the board chosen for design can even be dynamically reconfigured. This paper proposed a new solution for run-time reconfiguration limitations that enhances features such as security and reconfiguration load times. An added benefit is that such a system is generic enough that it can be implemented on FPGAs that have not been designed for dynamic reconfiguration. Security is one of the motivators behind this new implementation, and as such, notable security enhancements include no hardware signatures, bitstreams that do not reveal any sensitive information about the circuit, and the ability to quickly flush all circuit connections effectively destroying the circuit.

Another key issue related to CAD and programming is an understanding and appreciation of the performance achieved by cognitive processing applications. The next chapter addresses performance evaluation approaches we have explored for these platforms.

5. PERFORMANCE MODELING

5.1 INTRODUCTION

The performance of a large parallel application on shared resources depends on the performance characteristics of the resources, the division and scheduling of tasks, and the load balance across nodes of the computing platform. The *scheduling problem* has been well studied in the literature [124,125,126,127]. In this work, we focus on static scheduling of applications that follow a generalized Master-Worker paradigm (also called *task farming*). The generalized Master-Worker paradigm is easy to program and is especially attractive for grid platforms. In grid platforms, the availability of resources is typically in a state of flux and worker tasks in a Master-Worker paradigm can be easily re-assigned as needed when resource availability changes. Furthermore, many scientific and computationally intensive applications can be mapped naturally to this paradigm: N-body simulations [128], genetic algorithms [129,130], Monte Carlo simulations [131], and parameter-space searches, among many others. Cognitive processing applications can also share this structure of processing, with many types of signal and image processing, pub/sub processing of queries, distributed agent coordination, and neural network evolution/training or application following a master-worker form. In scheduling these applications, there are two main challenges: 1) how many workers should be allocated to the application and 2) how to assign tasks to the workers. In general, we would like to make the most efficient use of our computing resources while at the same time minimizing our runtime.

Networks or clusters of workstations can provide significant computational capabilities if effectively utilized. Similarly, large supercomputing platforms exist to provide high performance computing, but demand effective mapping of tasks to resources to exploit their resources. Adding reconfigurable computing (RC) devices to form a high-performance reconfigurable computing (HPRC) platform, introduces additional challenges for efficient utilization of all available resources. To address these issues, we have developed an analytic performance modeling methodology for *synchronous iterative algorithms*, a sub-class of *fork-join algorithms*, running on shared, heterogeneous resources which is completely developed in [112,132]. In this report, we present an overview of this model and investigate some example applied uses. If HPRC resources can be effectively utilized, users can garner increased performance and flexibility for a wide range of computationally demanding problems.

Execution time estimation plays an important role in parallel performance modeling. Given certain computation loads and resources, the execution time of each processor can be modeled as a random variable while that for the parallel system is determined by the last processor completing its task [133]. For example, Peterson and Chamberlain built a model for network stations [133,134]. The overall execution time consists of three parts: parallel work, serial work and overhead. When the number of tasks is big enough, execution time for each processor can be represented by Gaussian random variables X_i. Because of the effect of synchronization, the overall execution time is decided by the latest completed task. Therefore, the estimation of the system execution time depends on the calculation of the expectation of maximum value (EMV) $E(\max_{i=1}^{N} X_i)$. However, the solution in closed form usually cannot be derived for this term. This problem is even more challenging for heterogeneous environments, where the execution time for different processors has different distributions. Due to its importance in parallel computation modeling and other places, many researchers have tried various kinds of methods for this problem.

Monte Carlo (MC) methods can be used to compute EMV with any initial distributions. However, it has no analytical expression and is too computationally expensive for most of the evaluations. The use of order statistics is first suggested for analyzing parallel program performance by Weide [135]. For independent identically distributed (i.i.d.) random variables with known distribution functions, extreme theory [136,137] can be applied to approximate the distribution of extreme values. The approximation becomes exact when the number of random variables increases. This method cannot cover all distributions because it is too complicated to derive asymptotical extreme distribution functions for some distributions by extreme theory.

Agrawal [138] evaluates the performance of synchronous logic circuits simulation by applying the Binomial distribution to determine the number of events at each processor. The amount of active gates, which needs to be simulated for each processor, is randomly distributed. Order statistics is used to calculate the expectation of execution time when gates are equally assigned to each processor. In this case, the processor loads are independent and identically distributed binomial random variables. For imbalanced cases, processors are divided into different subsets. The processors in one subset are equally statically loaded, so the order statistics can be applied to calculate the expectation of execution time for each subset. Because there is no analytical method for the non-identical random variables, the maximum subset execution time is considered as the overall execution time [138]. As we will discuss later, this method brings large estimation error.

Despite of all these work above, the problem to compute the expectation of the maximum value is still left unsolved for decades. First of all, current method cannot accurately compute EMV for heterogeneous initial distributions. Secondly, even when initial random variables are i.i.d., current methods either cannot cover all the common applied distributions in parallel computing or are not accurate enough. MC simulation can be general and accurate for all distributions. However, its expensive computational requirement is usually unacceptable in large scale parallel computation performance evaluation. Furthermore, MC simulation cannot provide an analytical form for this problem, which is important for the property analysis of the execution time.

This report includes an interesting property of extreme values we discovered, which provides an analytical method to accurately approximate the expectation of parallel execution time for both homogeneous and heterogeneous environments. Compared to previous methods, it is more accurate, computationally effective, and general to probabilistic distributions. Using complicated task graph analysis for parallel computation modeling is a well known challenge. We utilize probability tools to simplify the complexity in task graphs and utilize the EMMA(Expected Mean Maximum Approximation) method for execution time analysis. Interdependencies are usually unavoidable among parallel tasks but very difficult to calculate in performance modeling. To help solving these problems, we also discuss the effect of interdependence on execution time estimation and extend the EMMA method for interdependent computing tasks.

5.2 BACKGROUND

5.2.1 HPC, RC, AND HPRC

In RC architectures, the general-purpose processor performs operations that are not efficient in the reconfigurable logic such as data setup and organization, loops, branches, and other control

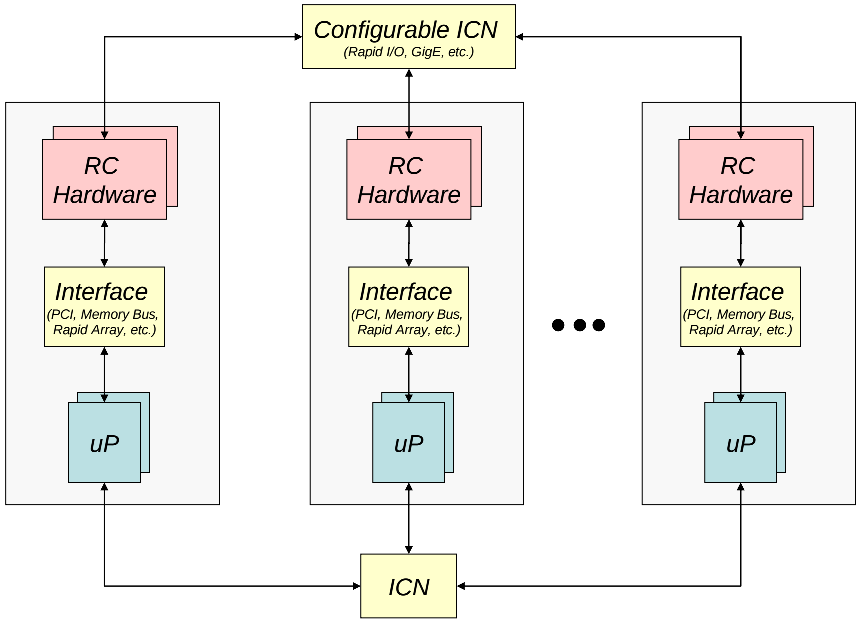

Configurable ICN
(Rapid I/O, GigE, etc.)
RC Hardware
Interface
(PCI, Memory Bus, Rapid Array, etc.)
uP
RC Hardware
Interface
(PCI, Memory Bus, Rapid Array, etc.)
uP
RC Hardware
Interface
(PCI, Memory Bus, Rapid Array, etc.)
uP
ICN

Figure 26: High-performance Reconfigurable Computer (HPRC) Architecture

structures, while computationally intense tasks are mapped to the reconfigurable hardware [4]. As shown in Figure 26, HPRC platforms consist of a number of computing nodes connected by some interconnection network (ICN); the computing nodes typically consist of a general-purpose processor coupled to the RC hardware via some communication interface (e.g. PCI, memory bus, rapid array, hypertransport, etc.). The HPRC platform allows users to exploit fine and coarse grain parallelism within the RC device and across the parallel compute nodes.

Several options currently exist for building or buying HPRC systems. In addition to a variety of RC cards such as the Pilchard [139] and others available from vendors such as Celoxica [140] and Nallatech [141], HPC vendors such as Cray with its XD1 [142] and SGI with their RASC^{TM} system [143] have entered the market with HPRC platforms featuring high-end processors tightly-coupled with reconfigurable devices. The advantage of these systems is that users are accustomed to programming in their environments (compilers, debugging tools, task managers, etc.). The disadvantage of these systems is the software stack specifically for the FPGA devices is not fully developed and difficult for scientists and end-users to utilize. Additionally, companies such as SRC with their line of MAPstations® [144], Opteron™-socket products from DRC Computers [5] and XtremeData [6], and systems using Intel's Accelerator Abstraction Layer support from DRC [5] and Nallatech [141] have entered the market with cluster based platforms. The advantages of systems from these vendors include their more sophisticated and developed software stack and that systems can be built incrementally (node by node) and in some cases, less expensively. The disadvantage of these systems is that the FPGA software development stack is often tightly coupled to the architecture and in the case of SRC, only works with their systems. One final advantage for the SRC system is that it is currently the only system offering Fortran support for FPGAs (Fortran is a commonly used language in scientific applications).

For all the HPRC systems mentioned thus far, the RC element of the system is in the form of an RC board or module and the primary architectural difference is the manner in which it is coupled with the rest of the system. Each of the above-mentioned platforms has a different communication interface between the general-purpose processors and the reconfigurable hardware devices in addition to other variations including the host processor, memory hierarchy, FPGA devices, clock frequency, etc. The model presented later provides parameters for describing these various differences enabling the performance comparison of an application running on a variety of different architectures from current to future systems.

5.2.2 Synchronous Iterative Algorithms (SIAs)

As the name implies, *Synchronous Iterative Algorithms* (SIAs), also known as *multi-phase algorithms* in the literature, are iterative in nature with each processor performing some portion of the required computation during each iteration. SIAs are a sub-class of a much broader set of algorithms known as fork-join algorithms where a main process or thread forks off some number of other processes or threads that then continue in parallel to accomplish some portion of the overall work before rejoining the main process or thread. These algorithms can be hierarchical and recursive, thus supporting a wide range of distributed and parallel applications. Many scientific algorithms fall into this large class of problems such as N-body simulations, Monte Carlo simulations, many discrete-event simulations, numeric optimizations, Gaussian elimination, FFTs, equation solvers, data encryption, sorting algorithms, and others.

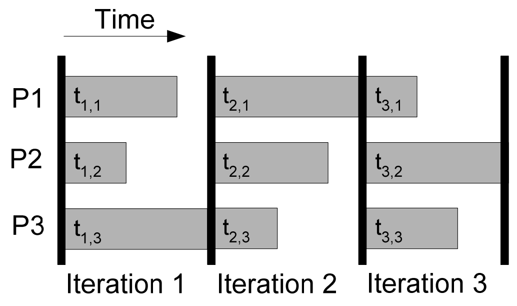


Figure 27: Timing of a synchronous iterative algorithm

In [112,132], we developed a performance model for fork-join type algorithms that takes into account the division of computation between the workstation processor and the reconfigurable device, task balance across the set of compute nodes, background loading (due to shared resources), and workstation heterogeneity. The development of an analytic performance model is a significant contribution that facilitates resource management optimization by helping users understand the performance tradeoffs in the computer architecture. With the knowledge and understanding of the combined system and application performance gained from the model, one can quickly determine the optimum application mapping for a set of constraints or identify the best set of computing resources to optimize runtime.

In order to effectively utilize our shared resources, we will employ this modeling methodology along with a usage policy with a measurable goal(s). Specifically, our objective is to choose an appropriate set of compute nodes on which to execute a given application that best meets our usage policy goals. The scheduling decisions must account for the individual processor performance characteristics as well as the existing background load. The desired usage policy is based on the relative importance of parallel applications to the existing background load (priority) and individual workstation characteristics such as processing power, type of or lack of reconfigurable hardware, current workload, or other factors. To implement the desired usage policy into a schedule for the parallel application, we build a cost function that represents the policy goals and using optimization techniques to minimize that cost function, we find a schedule for the parallel application that will most effectively utilize the available resources. It is well known that the optimization of a general cost function is an NP-hard problem [115] therefore true optimization is limited to restricted classes of problems which have efficient solutions or the use of heuristics to find near-optimal solutions.

5.3 PERFORMANCE MODEL OVERVIEW

In fork-join algorithms and SIAs, the time required to complete a given iteration is equal to the time required for the slowest or longest task to complete. The join processes in fork-join algorithms, or barrier synchronizations in SIAs, ensure that all processors start a given iteration together and that processors who complete their tasks early sit idle until the end of the iteration. Within a given iteration, a parallel algorithm normally includes some serial calculations (operations that cannot be parallelized), the parallelized operations, and some additional overhead (i.e. setup, synchronization). For the applications considered, each iteration requires roughly the same amount of computation making iterations similar enough that we consider the computations required for a “typical” iteration. This characteristic is valid for a large class of problems and is a reasonable assumption for many real-world algorithms. This tradeoff is used here to make the mathematics of the model less complex and more tractable for analysis. References to individual iterations could be maintained at the expense of a more complex model in the end.

First, we will assume we have a segment of an application having *I* iterations that will execute on parallel nodes with RC hardware accelerators and we will assume that all iterations exhibit similar behavior with respect to performance. Software tasks are distributed across parallel computing nodes and hardware tasks are distributed to the RC devices(s) at each individual node. For an SIA, we know the time to complete an iteration is equal to the time for overhead operations plus the time for the longest (slowest) task, which in the case of HPRC could be computed in either hardware or software depending on the task distribution. Therefore, we represent this iteration time as the overhead operations plus the maximum of the hardware or software task times:

$$\begin{gathered}\text{iteration time} = \text{overheads} + \\ \max(\text{hardware runtimes, software runtime})\end{gathered} \tag{5}$$

Looking at these terms in more detail, within each iteration of the algorithm there are some calculations which cannot be executed in parallel or accelerated in hardware and are denoted $t_{serial,i}$. There are other serial and overhead operations required by the RC hardware and they are denoted $t_{RC\text{-}ovhd,i}$. Other overhead processes that must occur such as synchronization and exchange of data between parallel computing nodes are denoted $t_{par\text{-}ovhd,i}$. The time to complete the tasks executing in parallel on the processor (software) and RC unit (hardware) are $t_{SW,i,k}$ and $t_{HW,i,j,k}$ respectively. For *I* iterations of the algorithm where *n* is the number of hardware tasks at node *k* and *m* is the number of processing nodes, the parallel runtime, R_P, can be represented as shown in (6), where in the second line, the $t_{SW,i,k}$ and $t_{HW,i,j,k}$ terms are combined to form the general task completion time for the RC node, $t_{RC,i,k}$.

$$\begin{aligned} R_P &= \sum_{i=1}^{I} [t_{serial,i} + t_{RC-ovhd,i} + t_{par-ovhd,i} \\ &\qquad + \max_{1\le k\le m}(t_{SW,i,k}, \max_{1\le j\le n}[t_{HW,i,k,j}])] \\ &= \sum_{i=1}^{I} [t_{serial,i} + t_{RC-ovhd,i} + t_{par-ovhd,i} \\ &\qquad + \max_{1\le k\le m}(t_{RC,i,k})] \end{aligned} \tag{6}$$

Now we combine the serial and RC-overhead into one serial term along with other manipulations detailed in [112] and separate the parallel overhead term into the communication and synchronization overheads, we can rewrite the parallel runtime as:

$$R_P = I \begin{pmatrix} t_{serial} + \beta \cdot t_{HW} + E\left[\max_{1\le k\le m}(t_{RC,k})\right] \\ + t_{synch} \cdot \log m + t_{comm} \end{pmatrix} \tag{7}$$

where n tasks are divided across the nodes, a load imbalance occurs due to application workload distribution (β), background users (γ), or workstation heterogeneity (δ). The he application load imbalance exists when tasks are unevenly distributed across the parallel computing nodes either a priori or during application runtime. Background load imbalance results from other users of the shared workstations. These users consume clock cycles that would otherwise be available for the parallel application in a dedicated system. Finally, workstation heterogeneity, or workstations with different performance capabilities, induces another load imbalance since some workstations will have better performance capabilities than others. We represent this total load imbalance product as η [112,132]. Assuming that the RC system load imbalance at any node is independent of the other nodes, the runtime for an application running on a shared, heterogeneous HPRC platform becomes:

$$\begin{aligned} R_P = \gamma \cdot t_{serial,i} + \beta \cdot t_{HW} + \frac{\eta \cdot t_{work}}{m} \\ + \left[t_{synch} \cdot \log m \right] + t_{comm} \end{aligned} \tag{8}$$

load imbalance on shared, heterogeneous workstations can be represented using scaling factors for application (α) and background (β) loading. To model heterogeneous processors, the scaling factors δ_j representing the processing time per unit work of processor *j*, and ω the time per unit work of the baseline processor are introduced. (For homogeneous resources, $\delta_j = \omega = 1$ since processor *j* will have the same performance as the baseline processor.) Finally, *B* represents the average work for a processing node and β_j/B is the application load imbalance scale factor for processor *j*. Therefore, for heterogeneous processors, η is defined as [112,132]:

$$\begin{aligned} \eta &= \frac{1}{B} E\left[\max_{1 \le j \le n} [\frac{\eta_j}{W_j}] \right] \\ &= \frac{1}{\omega B} E\left[\max_{1 \le j \le n} [\beta_j \gamma_j \delta_j] \right] \end{aligned} \tag{9}$$

Assuming the application and background load imbalances are independent of each other and that the application and background load imbalances of any processor are independent of the other processors, the expected value is [112,132]:

$$\eta = \frac{1}{\omega B} \sum_{k=1}^{\infty} k \cdot \left(\begin{aligned} &\prod_{j=1}^{m} \sum_{\alpha=1}^{k} \Pr ob\{\beta_j = \alpha\} \Pr ob\{\gamma_j \le \left\lfloor \frac{k}{\alpha \delta_j} \right\rfloor\} \\ &-\prod_{j=1}^{m} \sum_{\alpha=1}^{k-1} \Pr ob\{\beta_j = \alpha\} \Pr ob\{\gamma_j \le \left\lfloor \frac{k-1}{\alpha \delta_j} \right\rfloor\} \end{aligned} \right) \tag{10}$$

The application and background load imbalances are modeled with discrete-valued scale factors since the loads consist of some number of processes or tasks (a non-negative integer). Full derivation of the load imbalance model can be found in [112,132].

5.4 OPTIMIZATION PROBLEMS

In this section we will look at the application of the performance model to optimizing HPRC resources. Three algorithms are used to demonstrate the optimizations: the Level 3 BLAS routine DGEMM [145], a cryptography algorithm for Advanced Encryption Standard (AES) [146], and a Boolean SAT (Satisfiability (Boolean Satisfiability) Solver [147].

5.5 EXPERIMENTAL SETUP

In the analysis experiments discussed in this section, there were two experimental systems. In the first analysis discussed, the experimental system consisted of an SRC MAPstation [144]. The particular MAPstation used for testing consisted of dual 2.8 GHz Xeon® microprocessors with a Series C MAP® (Multi-Adaptive Processor). The MAP board is connected to the microprocessor via the SNAP® interface cards which plug into the DIMM slot. The MAP consists of two Xilinx Virtex II XC2V6000 devices running at 100MHz, a pre-configured FPGA control processor, and six 4MB SRAM banks referred to as On-Board-Memory (OBM). Code for both the host processors and the MAP hardware is written in standard C or Fortran. Function or subroutine calls are used to execute code on the MAP and are compiled by an SRC-proprietary compiler that targets the MAP components. The CARTE® environment [144] builds and links a unified executable that binds the application and configuration(s) for the MAP hardware.

The second experimental setup, used in the remainder of the analysis, consists of six Pentium workstations running Linux OS populated with Pilchard boards [139]. The Pilchard architecture consists of a Xilinx Virtex 1000E FPGA interfaced to the processor via the SDRAM DIMM slot. The logic for the DIMM memory interface and clock generation is implemented in the FPGA and the user's application code is developed using VHDL (Very High speed integrated circuits hardware Description Langauge). The FPGA is configured using the download and debug interface, which is separate from the DIMM interface and as such, requires a separate host program to configure the FPGA. Application code for the host processor is developed using a standard C compiler. Both experiment platforms address the FPGA-to-processor bandwidth bottleneck of the PCI bus by interfacing the FPGA device to the processor via the memory bus.

The overhead components of the problems consist of $t_{RC\text{-}ovhd}$ and $t_{par\text{-}ovhd}$. The former is related the RC hardware and consists of the time necessary to configure the FPGA device and move data to and from the FPGA device. The latter overhead contribution, $t_{par\text{-}ovhd}$, consists of the time necessary to perform functions such as synchronizations and communications between parallel tasks.

Instrumentation of the applications is accomplished with timer functions in the C code and hardware counters in the FPGA designs. The hardware counters do not introduce overhead but only provide measurement of computation internal to the FPGA. The timer functions in the C code do introduce overhead and care has been taken to minimize their impact. Background loads were synthetically generated and introduced via *cron* scripts to simulate background users and heterogeneity in an otherwise dedicated, homogeneous cluster.

We point out that although this development and validation work directly address parallel processing systems and reconfigurable computing (e.g., HPC, RC, and HPRC), the same framework can be applied to systems using accelerators such as GPUs. In fact, we have done so with very accurate results, modulo the compiler optimizations and instruction scheduling for the specific GPU processors. We expect similar applicability and accuracy for other accelerators such as Clearspeed [148]. We have not considered this modeling framework for other architectures such as quantum computing or DNA/molecular computing.

5.6 APPLICATION-ARCHITECTURE ANALYSIS AND OPTIMIZATION

A straightforward use of the modeling methodology in optimization and scheduling is to predict the runtime performance of applications executing on our shared HPRC resources. We have investigated the characteristics of a particular application under various workload conditions and policies, but we could also use the model to evaluate the performance impact of varying the problem size, network size, and other effects. The modeling methodology provides for easy investigation of these issues by simply changing the values for such things as the problem size, overhead, communication time, etc. We can easily consider the addition of more computing nodes, faster nodes, more RC devices, larger or faster RC devices, etc. and using appropriately constructed cost functions, determine their cost-effectiveness and/or performance impact. Figure 28 shows an analysis of the effects of FPGA speed, FPGA logic density, and data streaming on the performance of the Level 3 BLAS [145] routine DGEMM running on the SRC Series C MAPstation [144]. The first trace listed in the legend represents the software-only atlas-tuned version of DGEMM running on the SRC dual 2.8 GHz Xeon host processors and provides a baseline comparison for our analysis. The next two traces listed in the legend are HPRC implementations of DGEMM on the SRC MAPstation. The first of these traces, which is the lowest performance on the graph, represents the implementation of DGEMM in hardware and includes $t_{RC\text{-}ovhd}$. The next trace is the same hardware implementation but in the model, the overhead $t_{RC\text{-}ovhd}$ is hidden by with DMA-streaming. This technique overlaps the computation with the data movement and as seen in the graph, offers a significant improvement especially in the smaller dimensions where overhead costs dominate. Analysis of other application implementations has confirmed this improvement.

The remaining two traces are model results predicting performance of the DGEMM implementation with FPGA devices running at 2x the clock rate. Assuming the data bandwidth supports the increased computing throughput, there is a significant improvement in performance and ‘time to solution’ as shown in Figure 28. The first of these traces includes the worst-case data transfer overhead $t_{RC\text{-}ovhd}$ where the computation stalls and waits for data. The last trace, which offers the best performance on the graph (better than the atlas-tuned software-only implementation) across the entire dimension studied, utilizes the DMA streaming technique mentioned earlier to hide the data transfer costs by streaming data in parallel with computations. Through this simple straightforward analysis we see that the modeling methodology is a powerful tool for investigating the performance of an application and the impact of the characteristics of a hardware platform and programming techniques. The results of this analysis have led to an increase in the FPGA clock rate, higher data bandwidth in and out of the FPGA devices, and improved DMA streaming capabilities in the succeeding SRC MAPstation systems. The study also provided useful information about the bottlenecks in the application, which centered on the data access in the algorithm; focused techniques were then used to extract the maximum parallelism achievable with the given application and hardware architecture combination. These techniques included loop unrolling, block matrix operations, data sharing, and loop fusing. Additionally, macros provided with the CARTE development tool were used to implement the most efficient multiply-accumulators in the FPGA hardware. Performance analysis with the model allowed us to isolate parameters to determine the location of the bottlenecks and eventually how to overcome them.

5.6.1 Minimum runtime

To determine the schedule with the minimum runtime for shared homogeneous HPRC resources, we use the algorithm shown in Figure 29(a) [149].

First the compute nodes are sorted based on the arrival rate of background jobs. This process orders the nodes based on background load (or highest performance potential) since all nodes are of equal unloaded performance. Then the set of nodes for the schedule is selected to contain only the compute nodes with the lowest background task arrival rate. The runtime for the application

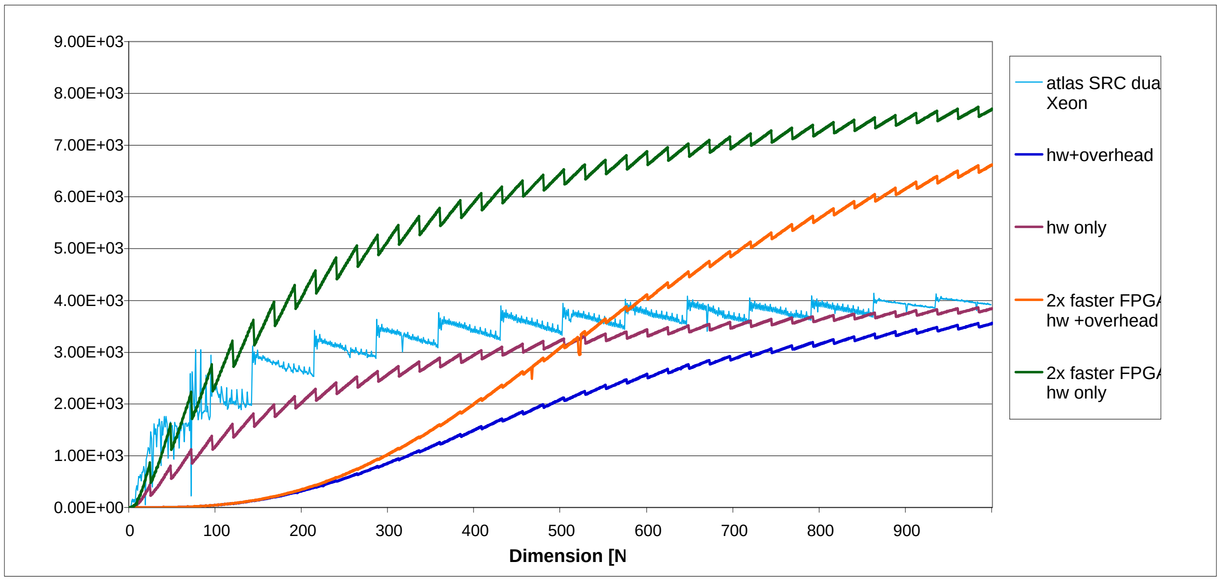


Figure 28: Performance analysis of DGEMM running on SRC MAPstation (NxN matrix)

is calculated for this computing node set and the next compute node is added in sorted order, repeating the runtime calculation until the added compute node no longer improves the runtime. In each calculation, an equal amount of work is assigned to each compute node in the set. This algorithm, similar to the one described in [150], gives the optimal solution for minimum runtime only when there is a minimum runtime, which is always true for SIAs where a barrier synchronization is required after each iteration. This optimization problem is applicable to all of the aforementioned HPRC platforms where any of them can be configured with either homogeneous or heterogeneous resources.

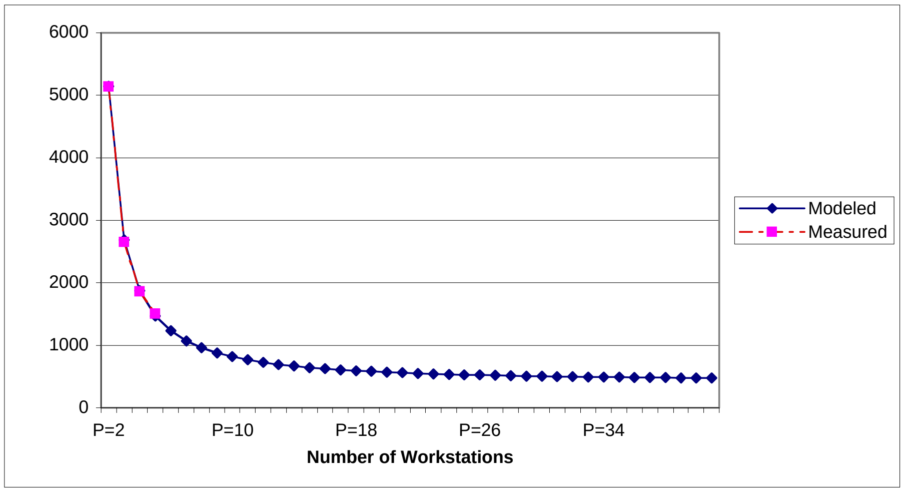
5.6.2 HOMOGENEOUS RESOURCES

For the AES algorithm, we see in Figure 29(b) that the runtime continues to improve from adding homogeneous workstations until we are using 25 computing nodes (in this case, we assume the same background loading model at each workstation, with an arrival rate of $\lambda = 0.2$). As more nodes are added, the additional overhead required for communication and synchronization of the new workstations negatively affects the performance. Additionally, due to the criteria used in the sorting algorithm, the background load is greater on each subsequently added node (the workstations are sorted based on background load), which contributes to a longer runtime. Therefore, we reach a threshold (approximately 25 nodes) where as nodes are added our performance no longer improves. Our measured results are limited by the size of our experimental platform - 6 nodes.

Next we look at the SAT Solver Application and investigate how changing the number of FPGA processing elements per computing node (parallel copies of the compute function in the FPGA device) and speed of the computing node resources will affect the application runtime. As shown in Figure 30(a), as one would expect, increasing the number of FPGA processing elements per computing node reduces the application runtime. However, as the total number of computing nodes increases, the three cases (4, 8, and 16 PEs/Node) approach one another and the performance advantage of extra processing elements per node is not as significant. In Figure 30(b) we again see that increasing the speed of the FPGA processing elements reduces the application runtime. However, as the total number of computing nodes increases, the three cases again approach one another and the performance advantage of faster hardware is not as significant.

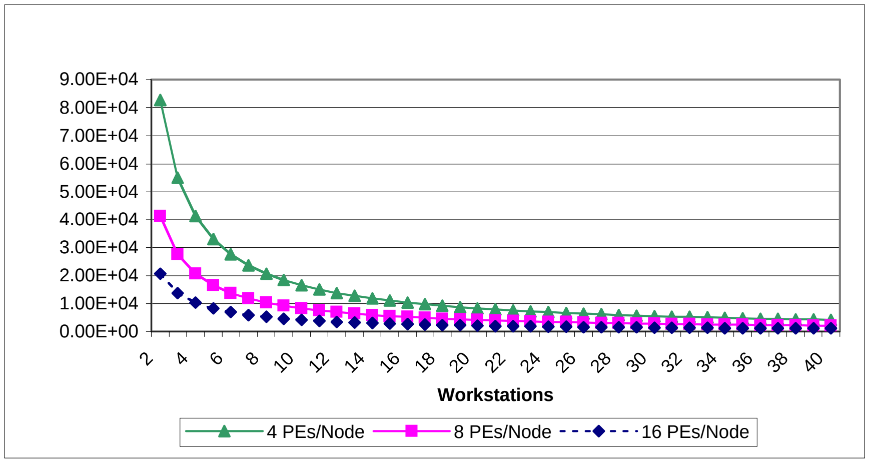
```
Sort available compute nodes $S$ by background job arrival rate $\lambda$
$P \leftarrow \{j\}$, where $j$ is compute node in $S$ with lowest arrival rate
while not done do
        $P' \leftarrow P \cup \{k\}$, where $k$ is next node in $S$
        if $R_{P'} < R_P$
            $P \leftarrow P'$
        else
            done $\leftarrow$ TRUE
endwhile
return optimal set $P$
```

(a)

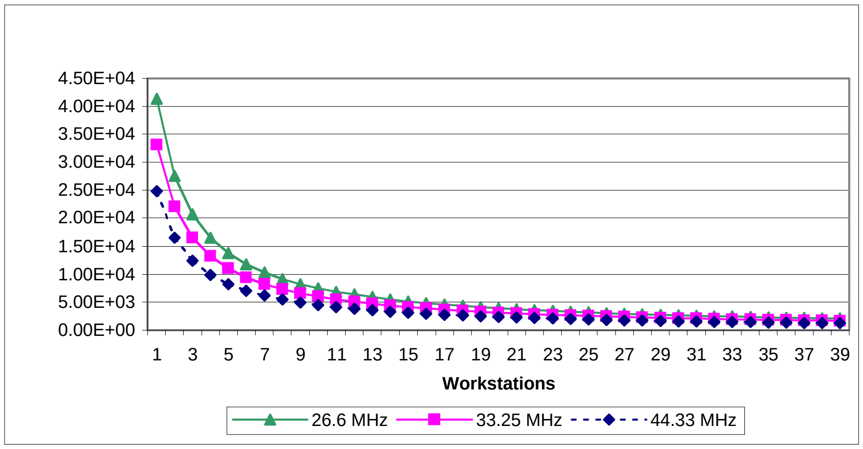


(b)

Figure 29: Homogeneous Resources: (a) Algorithm for minimum runtime on homogeneous resources (b) Optimum set of homogeneous resources for AES algorithm



(a)



(b)

Figure 30: Optimum Set of Homogeneous Resources for SAT Solver: (a) Compare number of PEs/Node (b) Compare FPGA Clock Frequency

5.6.3 Heterogeneous Resources

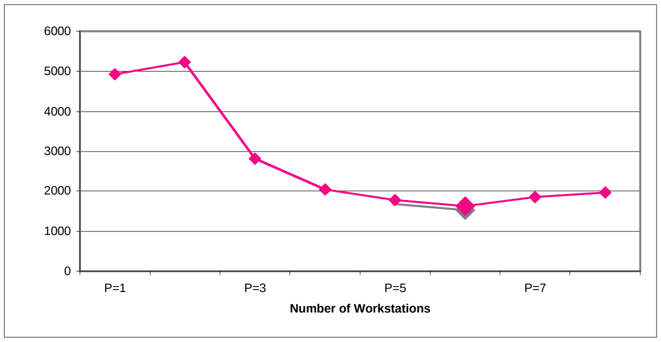
For applications running on heterogeneous resources, the performance model and algorithm are more complex and we resort to heuristics to find a near optimal solution since an efficient optimal solution does not exist [149]. Greedy heuristics are efficient to implement and in this case an acceptable choice even though they can get stuck in local extrema and therefore may not provide the best global solution.

If we assume that the application code is divided equally among the computing nodes, then we only need to consider the background load and heterogeneity when determining the optimum set of nodes. For the HPRC runtime model in [112], the background load is represented by a processor sharing queuing model and assuming that the service distribution of background tasks is Coxian [149,151], the expected number of background tasks at compute node j is $\rho_j/(1-\rho_j)$.

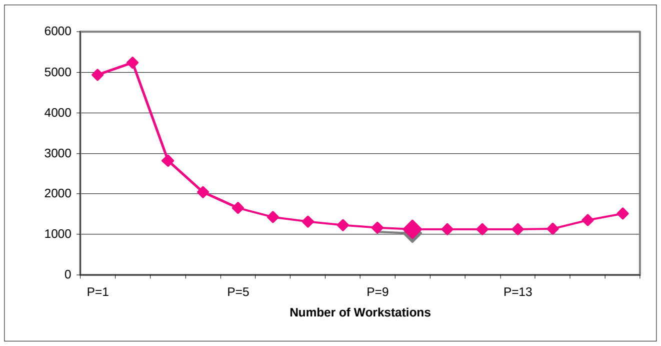
Adding the current application to the workstation load, the expected number of background tasks at processor j becomes $1/(1 - \rho_j)$. Since the compute nodes are heterogeneous, from the model in (8) and (9), we scale the runtime by δ_j/ω, where δ_j represents the processing time per unit work of compute node j, and ω the time per unit work of the baseline compute node. Multiplying by the background load (γ) and heterogeneity scale factors (δ_j), we find that compute node j is expected to take $\delta_j/[\omega(1 - \rho_j)]$ times as long as if the application ran on a dedicated baseline computing node. Using the heuristic algorithm shown in Figure 31, we sort the computing nodes based on the values of this scale factor rather than simply the background arrival rates as was done in the homogeneous case. We start with at least two workstations in the set and test the addition of more workstations as in the homogeneous example to find a near-optimal set of workstations P.

```
Sort available compute nodes $S$ by $\delta_j/[\omega(1 - \rho_j)]$, the scale factor for compute node $j$
$P^0 \leftarrow \{j\}$, where $j$ is first compute node in $S$     (serial case)
$P \leftarrow P^0 \cup \{j+1\}$, where $j$ is first compute node in $S$ (make sure we test at least 2 nodes)
while not done do
    $P' \leftarrow P \cup \{k\}$, where $k$ is next compute node in $S$
    if $R_{P'} < R_P$
        $P \leftarrow P'$
    else
        done $\leftarrow$ TRUE
endwhile (done when exhausted all nodes)
if $R_P < R_{P0}$
    return $P$ (near-optimal set)
else
    return $P^0$    (serial case faster than parallel)
```

Figure 31: Greedy Heuristic for Minimum Runtime on Heterogeneous Resources



(a)



(b)

Figure 32: Near-Optimum Set of Heterogeneous Resources for AES Algorithm, choose from set: (a) 8 nodes or (b) 16 nodes

In our first example, we model the AES algorithm running on eight heterogeneous nodes with four different speeds as denoted by the value δ_j for compute node *j*. Two of the compute nodes have $\delta_j = 1$, two have $\delta_j = 2$, two have $\delta_j = 4$, and two have $\delta_j = 6$. In Figure 32(a), we see that for our set of eight heterogeneous nodes, the runtime is at a minimum at six nodes.

In the second example we have a set of sixteen heterogeneous nodes with four different speeds. Two of the compute nodes have $\delta_j = 1$, four have $\delta_j = 2$, four have $\delta_j = 3$, four have $\delta_j = 4$, and two have $\delta_j = 6$. In Figure 32(b) we see that for our set of sixteen heterogeneous nodes, the runtime is at a minimum at ten nodes.

The algorithms we have discussed thus far ignore the impact on other users. By minimizing the runtime of the candidate application, without considering the impact on other users, the cost function essentially gives the candidate application priority over other users. In the next section, we will look at the use of a cost function to balance the runtime of the candidate application with the impact on other users while optimizing resources.

5.7 OPTIMIZATION SPACE ANALYSIS

To illustrate the power of the modeling methodology when used with cost function analysis, we will now analyze the optimization space using a fixed cost function and vary other parameters in the model. In our example, we will look at the SAT Solver Application running on a set of homogeneous resources with identical costs and a cost function representing the trade-off between maximizing the application performance and minimizing the impact to other users [149]:

$$C_P = xR_P + \sum_{j=1}^{P} c_j R_P = R_P \left(x + \sum_{j=1}^{P} c_j \right) \qquad (11)$$

The cost function reflects the usage policies encouraging the use of particular compute nodes for the candidate application based on the c_j values. Higher c_j values equate to higher cost or less desirable nodes. The x term represents the desire to minimize the execution time of the candidate application regardless of the number of nodes (higher x gives higher priority to the candidate application).

First, we will vary the number of processing elements of the SAT Solver engine implemented at each node (i.e. per FPGA) and determine how that number affects both the runtime and the cost. Figure 33 plots the optimal set of homogeneous resources for the SAT Solver application as we vary the x/c ratio. In Figure 34 we select the values of x and c such that x/c = 0.0001 to enact a usage policy that gives more priority to the background users relative to the candidate application. From the three plots we see that by changing the number of processing elements per FPGA at each computing node we reduce the overall runtime (right axis) as expected but the change also affects the cost analysis results. As the number of processing elements per node increases, the number of computing nodes in the optimum set for our cost function decreases (from 6 to 4 to 3 in this example). Although our overall runtime would be shortened with a larger set of nodes, our cost function is optimized with fewer nodes (see Figure 34).

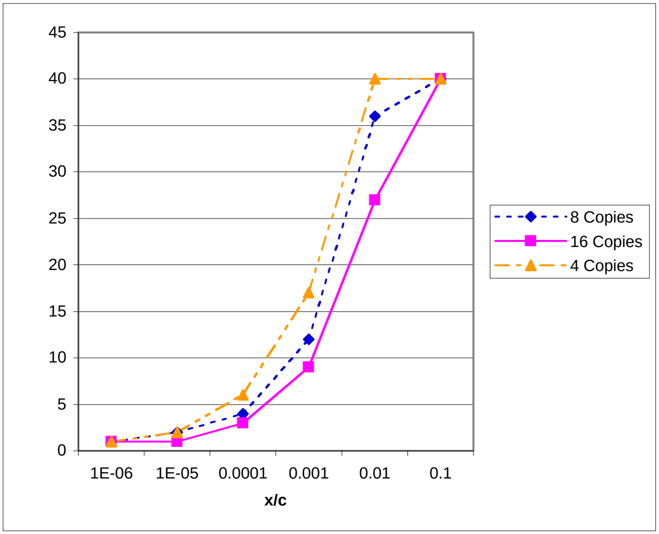
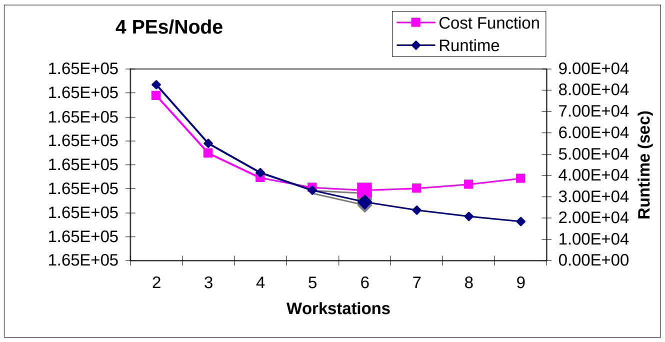
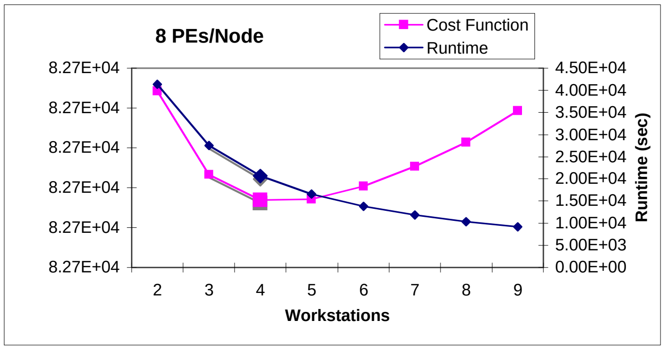


Figure 33: Optimal Set of Homogeneous Resources for SAT Solver, Varying the Number of Processing Elements per Node





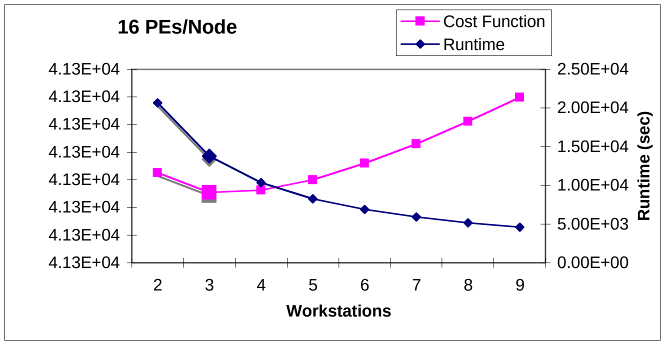


Figure 34: SAT Solver: Varying the Number of Processing Elements per Node (x/c = 0.0001)

5.8 COMPUTING THE EXPECTED MAXIMUM VALUE

To compute the execution time using the model developed above, we have to calculate the expectation for the maximum parallel execution time per iteration, which can be computed by numerical or analytical methods. An analytical solution of this problem is very helpful for performance analysis and optimization. If the individual runtimes are i.i.d., the distribution function of S is:

$$F_S(s) = \left(F_{t_{parallel,j}}(s)\right)^P \tag{12}$$

The density function of S is:

$$\begin{aligned} f_S(s) &= \frac{d(F_S(s))}{ds} \\ &= P\left(F_{t_{parallel,j}}(s)\right)^{P-1} \frac{d\left(F_{t_{parallel,j}}(s)\right)}{ds} \\ &= P\left(F_{t_{parallel,j}}(s)\right)^{P-1} f_{t_{parallel,j}}(s) \end{aligned} \tag{13}$$

The expected maximum is then:

$$\begin{aligned} E(S) &= \int_a^b s f_S(s) ds \\ &= \int_a^b sP\left(F_{t_{parallel,j}}(s)\right)^{P-1} f_{t_{parallel,j}}(s) ds \end{aligned} \tag{14}$$

Where a and b are the lower and upper bounds of random variable s. EMV (Expected Maximum Value) in equation (14) could be analytically solved for some simple distributions. However, numerical methods normally have to be used. The resulting computational load is unacceptable for many applications, such as dynamic load balancing and scheduling.

Extreme theory [136] could approximate EMV when the initial random variables are i.i.d. and follow certain distributions. For normally distributed random variables with mean μ and variance σ^2:

$$\begin{aligned} &E\left(\max_{1 \le j \le P} t_{parallel,j}\right) \\ &\approx \mu + \sigma\left[(2\ln m)^{1/2} - \frac{\ln\ln m + \ln 4\pi}{2(2\ln m)^{1/2}} + \frac{\gamma}{(2\ln m)^{1/2}}\right] \end{aligned} \tag{15}$$

Where γ is Euler's constant (0.5772).

Note that extreme theory gives asymptotic approximations as the number of random variables grows. It can only work for certain distributions. To find a general and effective execution time approximation for parallel performance evaluation, we introduce the EMMA method in the next section.

5.8.1 THE EMMA MEHTOD

Current methods cannot calculate EMV for many commonly used distributions. At the same time, it is normally a tedious task to derive the approximation functions from extreme theory. The EMMA method solves this problem by using a short function and constant. This section will also describe a novel extension to the EMMA method for non-identical initial distributions.

4.1.1.1 THE EMMA METHOD FOR I.I.D. RANDOM VARIABLES

To solve the mean of maximum random variable problem presented above, we introduce the EMMA method for i.i.d. (independent and identically distributed) tasks. For simplicity, we give the conclusions without explanation first. The mathematical proofs and extensions of the method are described in the next part.

Method I: Let X_i $(1 \le i \le n)$ be i.i.d. random variables, and $\mathrm{Y_n} = \max_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{X_i}$. Then $\mathrm{P}(\mathrm{X_i} \le \mathrm{E}(Y_n))^{\mathrm{n}} \approx \varphi$, where $E(Y_n)$ is the mean of Y_n and φ is a constant taken as 0.57. If X_i has distribution function F_i with inverse function F_i^{-1}, then $E(Y_n)$ can be approximated by $F_i^{-1}(\varphi^{1/n})$.

According to this theorem, equation (14) can simply be calculated by

$$F_S\left(E(S)\right) = \left(F_{t_{parallel,j}}\left(E(S)\right)\right)^P = 0.57 \qquad (16)$$

$$E(S) = F_{t_{parallel,j}}^{-1}\left(0.57^{1/P}\right) \qquad (17)$$

Compared to previous work, this theorem gives a much more effective approach for the EMV problem. By using $\varphi = 0.57$, the EMMA method replaces the complicated extreme distribution forms in order statistics. Mathematical explanation and proof will be given in the next part.

Assume $\mathrm{X_i}$ $(1 \le i \le n)$ are i.i.d. Gaussian random variables with mean μ, variance σ^2, and $\mathrm{Y}_n = \max_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{X_i}$. Here, we take μ=30, σ^2 =9. For each value of n, we use a MATLAB MC random number generator to produce the n Gaussian random variables and find the maximum value. We repeat this operation 500 times and get the expectation by taking the average of these 500 maximum values. MATLAB provides reverse distribution functions for many distributions. For the Gaussian distribution, the approximated EMV value $E(Y_n)$ for each n can be simply computed as:

```
30+3*sqrt(2)*erfinv(2*((0.57)^(1/n))-1)
```

Where erfinv is the inverse error function for Gaussian distribution.

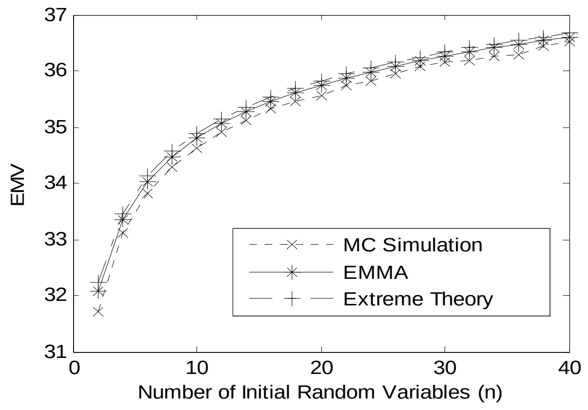


Figure 35: EMV by EMMA, Extreme Theory and MC simulation for Gaussian distributions

In Figure 35, we compare EMMA results to extreme theory [152] and MC simulations. Figure 35 shows that EMV from EMMA theorem accurately matches the MC simulation results. We repeat the above experiment many times and change the values of μ, σ^2, and n. We observe that EMMA approximates the simulation results consistently and accurately.

Binomial is another common distribution used in computer performance modeling. For example, in logic simulation, the number of gates to be simulated may follow a binomial distribution [133,153]. Assume X_i ($1 \le i \le n$) is binomially distributed with parameters M=5000 and p=0.02 (activity level in logic gate simulation). By using similar methods in example 1, we illustrate implementing method 1 for i.i.d. binomial distributions. The result is compared to MC simulation in Figure 36. The approximation accuracy increases with the amount of random numbers. However, there are some exceptions, which is because the inverse function of binomial distribution is discrete while the MC simulation gives continuous real numbers.

For both Gaussian and binomial distributions, the EMMA method gives similar results as MC simulation. Note that the approximation becomes more accurate when the number of random numbers grows. We compare EMMA and MC simulation for many common used distributions with arbitrary parameters, and get promising results in all tests. Usually, the approximation error is already very small when n is as small as 3.

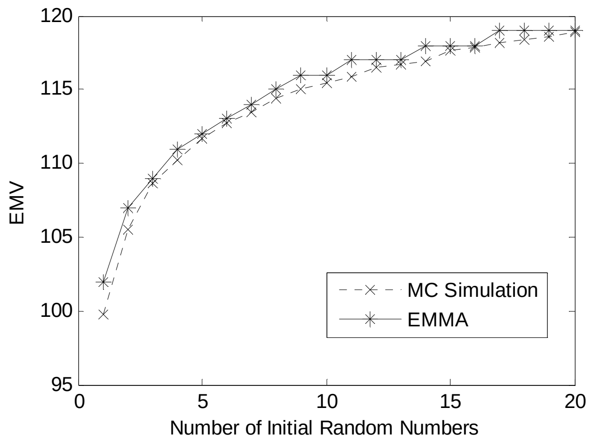


Figure 36: EMV by MC simulation and EMMA (Binomial distribution)

Mathematical Proof and Extensions

It is well known in order statistics that there are three types of distributions for extreme values: type I, type II, and type III [152]. These three types of distributions cover the asymptotic extreme distributions for most initial distributions. Most common initial distributions, such as normal, exponential and Rayleigh distributions belong to type I. Here we explain the EMMA method by using the properties of extreme distributions.

Theorem 1: For a Type I distribution with mean μ_n and cumulative distribution function $F_{Y_n}(\cdot)$, the following property exists: $F_{Y_n}(\mu_n) \approx 0.57$.

Proof: For a Type I distribution, the probability density function (PDF) and cumulative distribution function (CDF) for the maximum values are [152]:

$$f_{Y_n}(y) = \frac{1}{k}\exp\left[-\frac{y-\alpha}{k} - \exp\left(-\frac{y-\alpha}{k}\right)\right] \tag{18}$$

$$F_{Y_n}(y) = \exp\left[-\exp\left(-\frac{y-\alpha}{k}\right)\right] \tag{19}$$

The mean of this distribution above is:

$$\mu_n = \alpha + \gamma k \ ,$$

Where γ is the Euler-Mascheroni constant. Substituting μ_n into the CDF function, we get

$$\begin{aligned} F_{Y_n}(\mu_n) &= \exp\left[-\exp\left(-\frac{(\alpha+\gamma k)-\alpha}{k}\right)\right] \\ &= \exp[-\exp(-\gamma)] \\ &\approx 0.57 \end{aligned} \tag{20}$$

Theorem 2: For a Type II distribution with parameters v_n and k, let μ_n and $F_{Y_n}(\cdot)$ be the mean and cumulative distribution function. The following property exists: $F_{Y_n}(\mu_n) \underset{k\to\infty}{\rightarrow} 0.57$

Proof: For Type II distribution, the probability density function (PDF) and cumulative distribution function (CDF) for the maximum random variable are:

$$f_{Y_n}(y) = \frac{k}{v_n}\left(\frac{v_n}{y}\right)^{k+1}\exp\left[-\left(\frac{v_n}{y}\right)^{k}\right] \tag{21}$$

$$F_{Y_n}(y) = \exp\left[-\left(\frac{v_n}{y}\right)^{k}\right] \tag{22}$$

Where v_n is the characteristic largest value of the initial random variables and k is the shape parameter ($1/k$ is a measure of dispersion). The mean for this distribution is:

$$\mu_n = v_n\Gamma(1-1/k)$$

Where $\Gamma(\cdot)$ is the gamma function. Substituting μ_n into the CDF function, we get

$$\begin{aligned} \underset{k\to\infty}{F_{Y_n}}(\mu_n) &= \underset{k\to\infty}{\exp}\left[-\left(\frac{v_n}{v_n\Gamma(1-1/k)}\right)^k\right] \\ &= \underset{k\to\infty}{\exp}\left[-\left(\frac{1}{\Gamma(1-1/k)}\right)^k\right] \\ &\to 0.57 \end{aligned} \tag{23}$$

In order statistics, Type I and Type II are the extreme distributions for the initial distributions unlimited in the directions of the relevant extremes (maximum). In contrast, Type III represents the limiting distribution for initial distributions with a finite upper bound or lower bound value (we are only interested in upper bounds for this report).

Theorem 3: For Type III distribution with parameters w_n and k, let μ_n and $F_{Y_n}(\cdot)$ be the mean and cumulative distribution function. The following property exists: $\underset{k\to\infty}{F_{Y_n}}(\mu_n) \to 0.57$

Proof: For Type III distribution, the PDF and CDF for the maximum random variables are:

$$f_{Y_n}(y) = \frac{k}{\omega - w_n}\left(\frac{\omega - y}{\omega - w_n}\right)^{k-1}\exp\left[-\left(\frac{\omega - y}{\omega - w_n}\right)^k\right] \tag{24}$$

$$F_{Y_n}(y) = \exp\left[-\left(\frac{\omega - y}{\omega - w_n}\right)^k\right] \tag{25}$$

Where w_n is the characteristic largest value of the initial random variables, k is the shape parameter ($1/k$ is a measure of dispersion of X_n), and ω is the upper bound value of the initial distributions. The mean for this distribution is:

$$\mu_n = \omega - (\omega - w_n)\Gamma(1+1/k)$$

Where $\Gamma(\cdot)$ is the gamma function. Substituting μ_n into the CDF function, we get

$$\begin{aligned} \underset{k\to\infty}{F_{Y_n}}(\mu_n) &= \underset{k\to\infty}{\exp}\left[-\left(\frac{\omega - (\omega - (\omega - w_n)\Gamma(1+1/k))}{\omega - w_n}\right)^k\right] \\ &= \underset{k\to\infty}{\exp}\left[-(\Gamma(1+1/k))^k\right] \\ &\to 0.57 \end{aligned} \tag{26}$$

Figure 37 plots the CDF value at mean point μ_n ϕ_or three types of extreme distributions. It is always roughly 0.57 for Type I. With the growth of parameter k, the CDF values for Type II and Type III converge very quickly to 0.57 from above and below, respectively.

Based on the theorems above, we derive the following result.

Theorem 4: The CDF value at the mean point for Type I is always 0.57. For Type II and III, it converges to 0.57 quickly with the shape parameter k.

By theorems 1 to 3, we can derive the EMMA method for i.i.d. initial distributions, whose extreme distributions meet the following sufficient conditions:

i. Type I, or

ii. Type II/III with shape parameter k not too small,

For both Type II and III in extreme theory, the parameter k is the shape parameter, which is normally an increasing function of n and converges to constant when n approaches infinity. Note that this proof of the EMMA method uses extreme distributions in order statistics. Because extreme distributions are not exhaustive for all initial distributions, it is the same for this proof. That is, the conditions i and ii above are sufficient but not necessary. We do not need to worry about this exhaustiveness of the proof because all commonly used distributions are covered. Figure 38 gives approximation error on some commonly used distributions. Because of the lack of analytical methods for EMV computation for most of these distributions here, we compare EMMA with MC simulation. The results are listed when the number of processors is 5, 50 and 500. Note that the approximation becomes more accurate as the number of processor n increases.

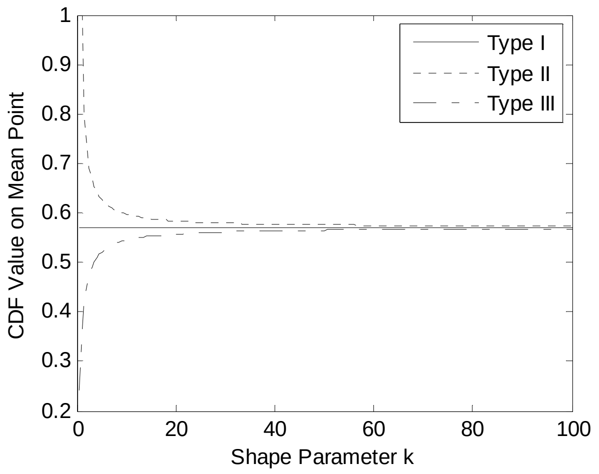


Figure 37: CDF values at maximum mean point converge to 0.57 for all three types of extreme distributions

For completeness, we now consider distributions that do not meet these two conditions although they could rarely be used in computation performance modeling. As with Figure 37, if for some certain initial distribution, the parameter k converges to a small value for a certain distribution, then a constant different than 0.57 should be used for φ to achieve more accurate approximation. However, if a certain approximation error can be tolerated, the constant 0.57 can still be used for simplicity. That is, the EMMA method is robust to parameter k. We describe this property by constructing a distribution converging to type III with shape parameter $k = 2$.

Example 3: Assume an initial distribution function has CDF and PDF as:

$$F_X(x) = 1 - ((10 - x)/10)^2 ;\ 0 \le x \le a \qquad (27)$$

And,

$$f_X(x) = (10 - x)/50 \qquad (28)$$

This distribution in type III, the asymptotic form for the maximum value is:

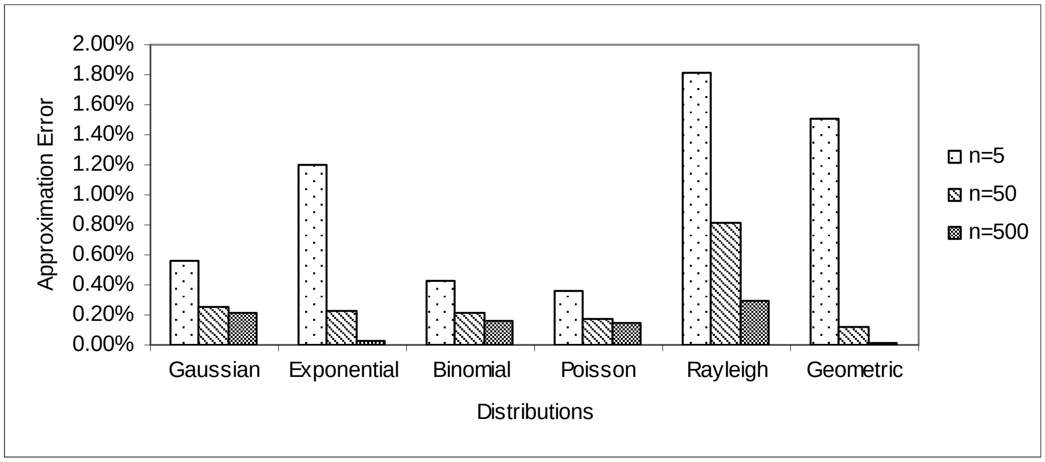


Figure 38: Approximation Error for Different Distributions

$$F_{Y_n}(y) = \exp[-n((10-y)/10)^2] \qquad (29)$$

With the parameters $k = 2$ and $\omega = 10$. The mean is

$$\mu_{Y_n} = 10\left(1 - 0.5\sqrt{\pi / n}\right) \qquad (30)$$

The shape parameter k very small and the related CDF value at maximum mean point $F_{X_n}(\mu_n)$ is around 0.46 in Figure 37. By extreme theory and the deduction of theorem III, we know that the EMMA method can accurately approximate the maximum mean for this distribution by taking the constant φ as 0.46. In this case, we are interested in the approximation error when φ is given 0.57. Figure 39 plots the approximation from the EMMA method when φ is 0.46 and 0.57.

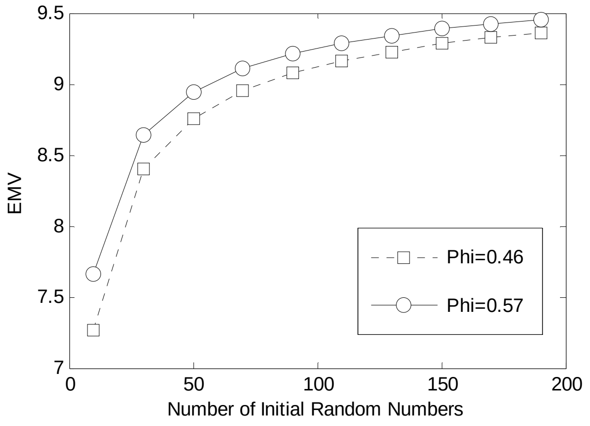


Figure 39: EMMA with different constants

Figure 39 shows that for a type III distribution with small shape parameter k, which does not meet the sufficient conditions, the EMMA method with constant 0.57 also follows the trend very well, but only with a little bigger approximation error when the amount of paralleled number of processors is low. We test the EMMA method for most commonly used distributions and find it is accurate for all distributions in large systems.

5.8.2 THE EMMA METHOD FOR HETEROGENEOUS DISTRIBUTIONS

Method 2: Let D be a set of independent random variables, and can be divided into m mutually exclusive subsets D_i $(1 \le i \le m)$. For each D_i, there are n_i i.i.d. random variables $X_{i,j}$ $(1 \le j \le n_i)$. Let $\mathrm{Y}_n = \max(\mathrm{X}_{\mathrm{i,j}})$ $(1 \le i \le m$ and $1 \le j \le n_i)$ for all the probability events. Then $\prod_{\mathrm{i}=1}^{m} \mathrm{P}(\mathrm{X}_{\mathrm{i,j}} \le \mathrm{E}(Y_n))^{\mathrm{n_i}} \approx 0.57$, where $E(Y_n)$ is the mean of Y_n. If $X_{i,j}$ $(1 \le j \le n_i)$ has distribution function F_i, then $E(Y_n)$ can be approximated by solving the function: $\prod_{\mathrm{i}=1}^{m} \mathrm{F_i}(\mathrm{E}(Y_n))^{\mathrm{n_i}} \approx \varphi$, where φ is a constant usually taken as 0.57.

The above is an extension of method 1 to non-identical random variables. Note that different subsets do not need to have the same kind of distribution in this method. This extends EMMA for heterogeneous computing environments.

Using method 2 to find EMV needs to solve an implicit function, where numerical methods can be used. The steps can be as the following:

1) Compute the EMV of any individual subset by method 1.
2) Let the result from step (1) as the initial value of $E(Y_n)$ and increase it by reasonable value each time until the approximation equation in theorem 2 is met.

Note that iteration times in step (2) can be reduced by choosing larger initial values in step (1). Since the overall extreme mean usually not much larger than the largest extreme mean of each subset, step (2) usually converges very fast.

We illustrate method 2 using a collection of Gaussian distribution. Assume there are three subsets, each with identically distributed random variables. The parameters are shown in Table 5:

Table 5: Subset Parameters

Parameter	Subset 1	Subset 2	Subset 3
Mean	40	45	50
Standard Deviation	12	9	6

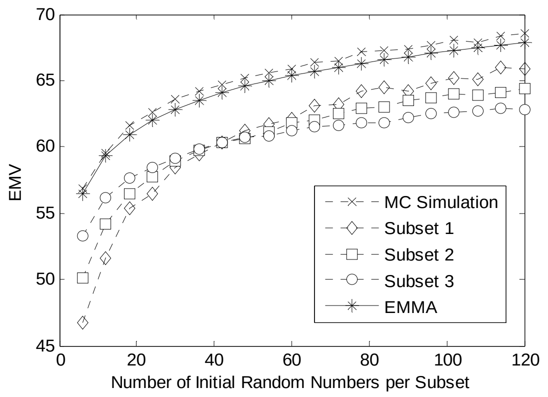


Figure 40: EMV from MC simulation and EMMA for heterogeneous environment (Gaussian distribution with different parameters)

In Figure 40, we assume each subset has the same number of tasks. The X-axis is the number of tasks for each subset. We can see EMV by Method 2 agrees accurately to MC simulation. EMV for each subset are also given by MC simulation. They are all under the overall maximum mean as expected. Because of a lack of analytical methods to calculate EMV for non-identical random variables, the largest execution time for individual subset is historically used as the overall execution time [138]. Figure 40 shows that this method can bring around ten percent errors even there are just three subsets of tasks.

We validate method 2 with various combinations of common used distributions and receive accurate approximation results for all of them. Figure 41 approximates the execution time when the tasks have different distributions as shown in Table 6.

Table 6: Subset Parameters

	Subset 1	Subset 2	Subset 3
Distributions	Gaussian	Gaussian	Exponential
Parameters	$\mu = 40, \sigma = 12$	$\mu = 45, \sigma = 9$	$\mu = 30$

Note that the parameter for Subset 3 stands for mean, instead of the parameter (one over mean) normally used in the density function of exponential distribution.

The X-axis in Figure 41 represents the number of processors per subset. We assume each subset has the same number of processors for simplicity. In this example, subset 3 is dominant and determines the maximum mean, which is also very accurately approximates by EMMA.

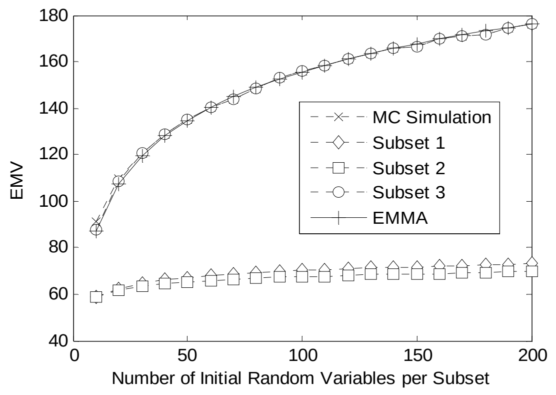


Figure 41: EMV from MC simulation and EMMA for heterogeneous environment (mixed distributions)

5.8.3 Utilization of the EMMA Method

The EMMA method provides an accurate and general mathematic tool for execution time approximation in parallel computing. It can also be conveniently used to analyze other characteristics of the system, such as speedup and optimal number of processors computation. This part describes using the EMMA method to analyze the system performance by an example in logic simulation, which is widely used to verify modern VLSI system design before fabrication. As the number of gates per VLSI chip increases, the simulation time becomes an important issue. We now apply the first model and Method 1 into an example in logic simulation.

An efficient logic simulation of circuits is possible by the event-driven method, where node voltages are represented by discrete values and their changes are restricted to discrete points in time [154,155]. The gates are modeled as functions to manipulate signals applied to their inputs and produce output signals. There is a finite delay for the gate operation depending on different gate types. On each clock cycle, plenty of the gates are inactive because their input signals keep unchanged. In event-driven method, only the active gates are simulated. For each of the iterations, the simulations for all the gates are independent and take roughly the same computational effort. Table 7 shows the active gates for some experimental circuits.

Table 7: Experimental Circuit Collections [156,157]

Circuits	Gate count	Average activity
CKT2	1754	0.03
8080	3439	0.001-0.005
Multiplier	5000	0.01-0.02

For event-driven method on parallel processors, tasks (gates) can be statically assigned to processors with approximately equal amount per processor. Due to the static allocation of gates to the processors, the numbers of potentially active gates for each processor are independent identical random variables. At the end of each of the iterations, the processors synchronize, share signal updates and proceed to the next iteration.

We first discuss the speedup characteristics of problems with stochastic execution time. The time used for synchronization, communication are neglected for simplicity. Now, equation (13) is simplified as:

$$R_P = I \cdot E\left[\max_{1\le j\le P} t_{parallel,j}\right] \qquad (31)$$

Assume the Multiplier circuit is simulated on 5 paralleled processors with 1000 gates per processor. If 0.02 is picked up as the average activity, the number of active gates per processor $\mathrm{n_i}$ ($1 \le i \le 5$) is binomial distributed with parameters 1000 and 0.02. That is, $n_i \sim B(1000,0.02)$.

Assume the computation effort for simulating each gate is one time unit and 300 iterations are needed. The expected execution time can be derived by equation (17), where the function F^{-1} is now the inverse function for binomial distribution:

$$R_P = 300 \cdot F^{-1}_{t_{parallel,j}}\left(0.57^{1/5}\right) \qquad (32)$$

By using the inverse Binomial distribution function, we get the execution time expectation: $R_P = 7800$.

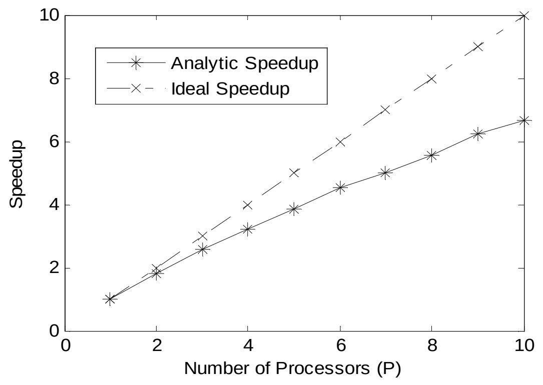


Figure 42: Ideal vs. analytic speedup without counting time on synchronization, communication and overheads

For simulation on a single processor, the execution time is

$$R_1 = 300 \cdot F^{-1}_{t_{parallel,j}}\left(0.57^{1/1}\right) = 30600 \qquad (33)$$

Note that $\mathrm{F}^{-1}_{\mathrm{t_{parallel,j}}}$ is now the inverse function for binomial distribution $B(5000,0.02)$. The speedup is:

$$Speedup = \frac{R_P}{R_1} = \frac{30600}{7800} = 3.92 \qquad (34)$$

where we can see the parallel speed up cannot achieve the ideal even when the time on synchronization, communication and overhead are not counted. The reason is for parallel computation on multiple processors:

$$E\left[\max_{1\le j\le P} t_{parallel,j}\right] > E\left[t_{parallel,j}\right] \tag{35}$$

Assuming this simulation task is assigned to a variety number of processors, Figure 42 plots the speedup with the number of processors. This example demonstrates that: for the problems with stochastic execution time on each processor, the speedup can never be ideal.

In real practice, the synchronization and communication times cannot be neglected in many cases and can be modeled as a function of the number of processors [158]. The following will introduce finding the optimal number of processors to achieve the minimum execution time by using the EMMA method.

For simplicity, we assume that the time for synchronization and communication is linear to the number of processor, that is, for equation (12):

$$t_{par\_overhead,i} = k(P-1) \tag{36}$$

k is a constant and taken as 2 in this example. Equation 13 becomes:

$$R_P = I\left(t_{serial} + E\left[\max_{1\le j\le P} t_{parallel,j}\right] + 2(P-1)\right) \tag{37}$$

After taking away the constants I and t_{serial}, which will not affect our optimization results, the cost function to minimize the execution time can be simplified as:

$$C = E\left[\max_{1\le j\le P} t_{parallel,j}\right] + 2(P-1) \tag{38}$$

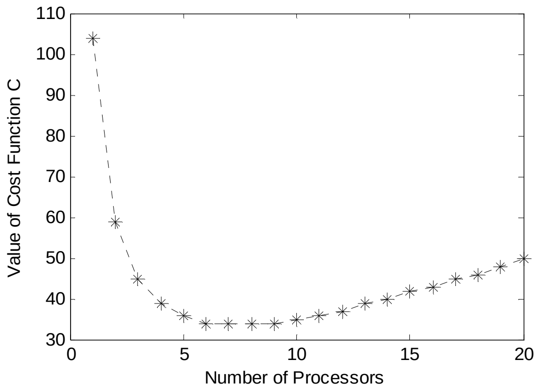


Figure 43: Simplified cost function for finding optimal number of processors

By using the EMMA method, we can easily plot this equation as shown in Figure 43. The optimal point is where the value of cost function C is the smallest. In this particular case, the cost function has similar values when the number of processors is 6, 7, or 8. If other factors like economics are considered, 6 would be expected to be the best selection.

5.8.4 EXECUTION TIME FOR TASK GRAPHS

Task graphs are often used to describe program executions. Lots of efforts have been delivered to analyze the execution time of task graphs by many researchers. In this part, we discuss the analysis on complicated task graphs by using the EMMA method. For simplicity, some results from probability are cited without proof.

Precondition 1: Let $X_1 \cdots X_n$ be random variables and $X = \sum_{i=1}^{n} X_i$, then $E(X) = \sum_{i=1}^{n} E(X_i)$.

This precondition is well known in probability theory, which says that the mean of the sum is equal to the sum of the mean. For a task graph in Figure 44a, the overall structure of the task graph is serial, where each phase could be parallel tasks. In such a paragraph, the overall execution time is equal to the sum of the execution time for all phases. For phase having parallel tasks, the mean execution time of that phase can be computed by the EMMA method.

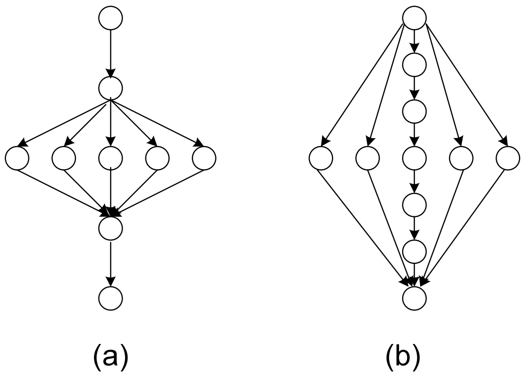


Figure 44: Serial and parallel task graphs

For the task graph shown in Figure 44b, the middle path consists of a series of tasks. To apply the EMMA method, we consider the overall task graph is parallel, so the distribution functions for all paths are required. We discuss finding distribution for the sum of serial tasks in the following.

Precondition 2: Let $X_i\ (i = 1, \cdots, n)$ be a normal random variable with mean μ_i and variance σ_i^2, $X = \sum_{i=1}^{n} X_i$. Then X is still a normal random variable with mean $\mu = \sum_{i=1}^{n} \mu_i$ and variance $\sigma^2 = \sum_{i=1}^{n} \sigma_i^2$.

Since the Gaussian distribution is usually used to model running time, it is important that the distribution can be accurately calculated for the sum of Gaussian distributions. Unfortunately, this might not be possible for other distributions. However, these non-Gaussian distributions can be approximated according to the central limit theorem.

Precondition 3: Let $X_i\ (i = 1, \cdots, n)$ be independent and $E(X_i) = \mu_i$, $Var(X_i) = \sigma_i^2$. Assume that $\sup_j (E|X_j|^{2+\varepsilon}) < \infty$ for some $\varepsilon > 0$. Let $X = \sum_{i=1}^{n} X_i$, then X converges to a Gaussian random variable with mean $\mu = \sum_{i=1}^{n} \mu_i$ and variance $\sigma^2 = \sum_{i=1}^{n} \sigma_i^2$.

The proof of precondition 3 can be found in [152]. Once the distribution functions of all the paralleled paths are available, the overall execution time in Figure 44b can be computed by using the EMMA method for heterogeneous cases. It might not be accuracy to apply the central limit law when the number of serial processes is too small. A more accurate method is to compute the distribution formula for the sum of random variables. However, it is usually too complicated and not easily extended.

5.8.5 Extension to Dependent Tasks

For parallel computation performance evaluations, independence is usually assumed for simplicity. However, dependencies usually exist due to many reasons. First of all, the tasks can be dependent themselves. For example, in logic gate simulations, the active gates might be related. Secondly, for some parallel computer architectures, parallel programs have to share some common hardware which brings dependence. Thirdly, some tasks might be dependent by sharing a common path. Communication and synchronization will also bring dependencies. It is very difficult to quantify dependencies, so normally they are just neglected for simplicity sake. For example, Madala approximates the execution time by assuming task paths are independent [137].

It is very necessary to analyze the inaccuracy caused by assuming independence. The following subsection will show that the EMMA model is an improvement on current methods, but is not a general solution for when dependencies exist. The improvements are due to the extension of the EMMA method for programs with interdependencies due to associated tasks.

ASSOCIATED TASKS

For parallel programs with dependencies due to associated tasks, the EMMA method can be applied by neglecting the dependencies. The following part will discuss the result in this case for associated parallel tasks. Two tasks are associated if the load increase of one affects that of the other. An example is that the number of active logic gates increase simultaneously on different parts of a circuit. A precise definition for association is as the following [159].

Definition 1: Random variables $X_1, \cdots, X_n$ are associated if

$$\operatorname{cov}[\Gamma(X), \Delta(X)] \geq 0 \qquad (39)$$

For all pairs of increasing binary functions Γ and Δ.

According to the theory of reliability, if X_i $(1 \leq i \leq n)$ are associated random variables then [159,160]

$$P[X_1 \leq y, \cdots, X_n \leq y] \geq \prod_{i=1}^{n} P[X_i \leq y] \qquad (40)$$

Let $Y_n = \max_{i=1}^{n} X_i$, then

$$F_{Y_n}(y) \geq \prod_{i=1}^{n} F_{X_i}[y] \qquad (41)$$

Corollary 1: For dependent associated parallel tasks, the result from EMMA theory by ignoring dependence is an upper bound of the real mean of the maximum.

Proof: according to theorem 4, the mean of the maximum can be computed by:

$$F_{Y_n}\left(E(Y_n)\right) = 0.57 \tag{42}$$

From equation (41), we have

$$\prod_{i=1}^{n} F_{X_i}\left[E(Y_n)\right] \le 0.57 \tag{43}$$

When we compute the mean of the maximum by ignoring the dependence, we take the inequality in equation (43) as equal. That is we compute by using

$$\prod_{i=1}^{n} F_{X_i}\left[(Y_n)\right] = 0.57 \tag{44}$$

Since the function sum and cumulative distribution function are both non-decreasing, the computed results are bigger than or equal to the real value. The equality is achieved when the random variables are mutually independent.

SHARING COMMON PATHS

The dependence between paths can also be caused by sharing common paths. Note that, although the dependence caused by sharing the common path appears to meet the definition of association, corollary 1 cannot be directly applied because of the synchronization effect. For the task graph in Figure 45, some previous work [160] concluded the execution time in task graph b is the upper bound of its counterpart with common paths (graph a). Here we give a counter example illustrating the error of previous methods.

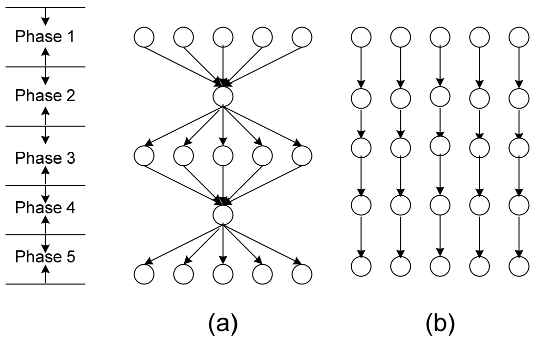


Figure 45: Task graph with common paths (a) and its independent counterpart (b)

Assume the sub tasks in Figure 45 represented by circles are identical. The running time of each subtask is Gaussian distributed with mean 30 and variance 9. By applying the EMMA method the mean execution time for phase 1, 3 and 5 in task graph (a) is computed into 34.033. Therefore the overall average runtime can be calculated as 34.033+30+34.033+30+34.033=162.099. For task graph (b), the running time distribution for each path is Gaussian distributed with mean 150 and variance 45, the overall execution time is 159.227. MC simulation results agree with our results. This example means that if we compute the execution time of task graph (a) by considering it as 5 independent paths like in task graph (b), the result is less than the real value (159.227 < 162.099). The reason is that paths in task graph (a) are not the same as in graph (b) because phase 1, 3 and 5 need to be synchronized before proceeding to the next phase and this synchronization costs extra time. Therefore, for the instance presented, the EMMA method assuming independence provides a more accurate upper bound than that of [160].

Accurate performance modeling of parallel applications faces difficulties due to the challenge of finding EMV. Although many efforts have been devoted, the problem is still left unsolved for decades especially for heterogeneous computing. Our work can be considered as an extension of Extreme Theory especially to heterogeneous distributions. By exploiting extreme value properties, we propose the EMMA method that is capable of finding fast, accurate solutions for the parallel execution time in both homogeneous and heterogeneous environments. We presented mathematical proofs and sufficient comparisons to MC simulation, which demonstrate the accuracy and generality of our method. EMMA can significantly improve the efficiency of parallel computation modeling.

5.9 PERFORMANCE EVALUATION CONCLUSIONS

High Performance Reconfigurable Computing (HPRC) platforms, and other accelerator technologies, offer the potential for cost-effective performance improvements for many computationally intensive applications provided the resources are used efficiently. We have developed a performance modeling methodology for fork-join and more specifically Synchronous Iterative Algorithms (SIAs) running on shared HPRC resources and demonstrated how to exploit those resources by optimizing scheduling of the parallel applications using the modeling results.

A sample cost function was considered to reflect different policy goals and priorities. We discussed how the model could be used to analyze different applications and their performance on the shared HPRC resources. We have only scratched the surface of the wide applicability and use of the model for problems in optimization, scheduling, and performance evaluation.

Future work will continue investigating the application of this modeling methodology to other dual and multi-paradigm computing platforms. These computing systems consist of multiple specialized devices tightly-coupled in a single environment. The specialized devices of interest include FPGAs, graphical processors, cell processors, PIMs, ClearSpeed [148], and others. The modeling methodology and application presented in this paper can be applied in these systems because the theoretical approach is analogous: *offload the intense computation to the specialized computing device*. Additional work will involve incorporating the model into CAD tools for scheduling and optimization for these systems.

6 CONCLUSIONS

Cognitive processing applications present a set of challenges for computational architectures. Given the diversity of applications that can be classified as "cognitive", one would not expect a single approach to be universally the best. Nonetheless, as the scale of cognitive processing application increases, previous architectural approaches seem unlikely to meet USAF needs because processor performance has stagnated in recent years. Similarly, embedded cognitive processing applications face further constraints on their size, weight, power, and related characteristics, so emerging technologies already can perform better than traditional computer architectures for many of these problems. This report aims to survey the field of cognitive processing applications, the computational technologies available to implement them, and the infrastructure for developing and tuning these applications.

A range of computational architectures and emerging technologies are now available for implementing cognitive processing applications. Serial processors have not significantly improved in their performance in the past few years because power and other constraints have rendered clock increases impractical and continued microarchitectural enhancements have remained elusive. Instead, multiple processor cores are now commonly deployed within the same processor die, making parallel processing architectures the norm. In the foreseeable future, the additional transistors posited by Moore's Law will be invested in doubling the number of processor cores every eighteen to twenty-four months. At the same time, emerging computing technologies show promise for providing performance improvements. For example, general purpose processing with graphical processing units (GPGPUs) has recently become quite popular for researchers interested in accelerating floating point computations with the parallel arithmetic units contained in moderns GPUs. Similarly, reconfigurable computing with FPGAs shows promise for faster and more efficient processing by allowing developers to create application-specific customized circuits. Other technologies such as quantum computing and DNA/biomolecular computing loom on the horizon as potential approaches to achieve performance gains, but most likely at least a decade remains before these technologies will be viable.

Cognitive processing applications span a range of domains with varying attributes. We surveyed a set of application types to explore their characteristics in order to provide insight into next-generation computational needs for cognitive processing applications. The artificial neural network approach to machine learning remains popular for many applications, so we explored mapping neural networks to reconfigurable computing as well as evolutionary computing techniques to improve their effectiveness over time. We then considered attempts to create much larger brain-like structures with neuron circuits that more closely approximate biological neurons. Scaling these neuron models to yield cognitive processing rivaling that of small mammals will require vast computational processing that will best be implemented using analog circuits. Hence, this type of cognitive processing will probably require the development of a new computational structure based on analog primitives.

Situational awareness demands a good understanding of the state of a wide array of systems, people, and environment. This can be achieved by culling important information from large numbers of disparate data sources and extracting salient situational knowledge. In order to provide flexible distribution and access to data, the fielding of systems such as the Joint Battlespace Infosphere requires that data sources must be able to flexibly publish their results while authorized users are empowered to subscribe to the data pertinent to their mission. We explored the processing associated with pub/sub and databases, in particular with managing large numbers of queries with the various computing technologies as well as managing the authorization and access to the data. With our previous research into bioinformatics applications manipulating data from large databases to extract information, we explored the potential for large parallel supercomputers, GPUs, and reconfigurable computing to help with these types of processing.

With the complexity of individual processing elements such as pipelined, superscalar processors supporting hyperthreading and multilevel caching, parallel systems now employing thousands of cores and marching towards millions of cores, or FPGA devices that enable new circuits to be employed, the ability to effectively program these systems for cognitive processing applications looms as a critical factor. We surveyed the vast array of languages and tools that address programming these different computing technologies. Because defense systems require support for decades, many commercial tools and techniques for systems development may not be appropriate. Hence, programming and related development tools remain a critically important aspect of the computing environment and will require ongoing investment by DoD/USAF to ensure the ability to meet defense needs.

With the sensitivity of critical or classified data or the recent widespread emergence of computer hackers attacking every computer system, security and reliability represent critical aspects of defense systems. Accordingly, we explored the use of reconfigurable computing to provide more robust processing capabilities so that the systems and information are both secure and reliable. The approach we used was to develop “sandboxes” for processing applications on virtualized reconfigurable hardware. It provides flexibility and the potential to share reconfigurable computing hardware among multiple processes or users concurrently. This approach also enables runtime reconfiguration to easily customize hardware for evolving mission needs. It also provides defense against tampering or reverse engineering, as the “flushing” of sandbox data and configuration information will provide a level of complexity above that of encrypted bitstreams protecting the FPGA circuit. Moreover, we explored software service support with the operating system and the sandbox controller to enable the system to operate. Finally, we developed an approach for saving the state of sandboxes or other FPGA circuit state. In this way the state can be restored after a reconfiguration to enable process migration, provide better robustness to failures, or enable runtime reconfiguration for better system performance.

The complex space of applications and architectures makes it difficult to determine the best approach to employ. In order to provide a mathematical foundation for any tradeoffs or optimization, we developed a performance modeling framework. Although this framework was originally developed for reasoning about parallel and distributed computer systems, we have extended it to include support for reconfigurable computing as well. Similarly, this modeling framework seems to perform well for GPGPU processing too. This framework has been used to explore performance and identify bottlenecks as well as to support optimization with respect to runtime, cost, power dissipation, and other constraints.

One aspect of the performance modeling framework is the analytical solution of the expected value of the maximum of a set of random samples. This represents the final task to reach a barrier synchronization or the last to reach a “join” in a fork-join task graph. In practice, solving this problem is mathematically intractable in most interesting cases. Others have explored the use of the statistics of extremes or order statistics, although this approach typically provides loose bounds. Many performance modeling efforts have faced this challenge, and there are numerous papers with incorrect solutions to this problem. To address this, we developed a theory that allows an accurate and easy approximation to the expected mean of the maximum. We discuss this theory and its application for homogeneous and heterogeneous mixes of distributions, thus allowing us to reason about complex computing systems.

Having surveyed this array of topics related to cognitive processing applications and the best architectural approaches for implementing them, we can summarize the results by noting the following. First, traditional processing with serial CPUs is no longer tenable. Second, the emerging GPU architectures show great promise for floating point computations, and are likely to merge with traditional processors. Third, reconfigurable computing promises to provide substantial performance improvements as well as more robust operation with respect to reliability and security. Finally, parallel processing will be a fundamental aspect of cognitive processing applications, so parallel program and related development environments should continue to be a topic of sustained research and development by DoD/USAF.

7 REFERENCES

1 John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition. Morgan Kauffman. 2009.

2 Moore, G. http://en.wikipedia.org/wiki/Moore%27s_law - _ref-Moore1965paper_1 Cramming more components onto integrated circuits. Electronics Magazine 1965.

3 Shameem Akhtar and Jason Roberts, Multi-Core Programming: Increasing Performance through Software Multi-Threading. Intel Press. 2006.

4 Compton, K. and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Comput. Surv.*, Vol. 34:2, pp. 171-210, June 2002.

5 DRC Computer Corp., http://www.drccomp.com

6 XtremeData, Inc., http://www.xtremedatainc.com

7 H. DeGaris, "Evolvable Hardware : Principles and Practice." http://www.cs.usu.edu/~degaris/papers/CACM-E-Hard.html," 1997.

8 H. de Garis, "Artificial brain: ATR's CAM-brain project aims to build/evolve an artificial brain with a million neural net modules inside a trillion cell cellular automata machine," *New Generation Computing,* vol. 12, pp. 215-221, 1994.

9 A. Thompson, "An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics," in Lecture Notes in Computer Science, No. 1259, "Evolvable Systems : From Biology to Hardware", First International Conference, ICES96 Tsukuba, Japan, 1996, pp. 390-405.

10 T. Higuchi, M. Murakawa, M. Iwata, I. Kajitani, L. Weixin, and M. Salami, "Evolvable hardware at function level," in *IEEE International Conference on Evolutionary Computation*, 1997, pp. 187-192.

11 J. Zhu and P. Sutton, "FPGA Implementation of Neural Networks - A Survey of a Decade of Progress," *Proc. 13th International Conf on Field-Programmable Logic and Applications*, Lisbon, Portugal, 2003.

12 C. E. Cox and W. E. Blanz, "GANGLION-a fast field-programmable gate array implementation of a connectionist classifier," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 288-299, 1992.

13 S. A. Guccione and M. J. Gonzalez, "Neural network implementation using reconfigurable architectures," *International workshop on field programmable logic and applications,* Oxford, UK, 1993.

14 P. Lysaght, et. al., "Artificial Neural Network Implementation on a Fine-Grained FPGA," *4th International Workshop on Field-Programmable Logic and Applications*, Prague, Czech Republic, 1994.

15 J. G. Eldredge and B. L. Hutchings, "Density enhancement of a neural network using FPGAs and run-time reconfiguration," *Proc. of IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

16 R. Gadea, et. al., "Artificial neural network implementation on a single FPGA of a pipelined on-line back-propagation," *Proc. 13th International Symposium on System Synthesis,* 2000.

17 M. van Daalen, et. al., "Emergent activation functions from a stochastic bit-stream neuron," *Electronics Letters,* vol. 30, pp. 331-333, 1994.

18 B. Noory and V. Groza, "A reconfigurable approach to hardware implementation of neural networks," IEEE *Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2003.

19 R. Sutton, "Two problems with back-propagation and other steepest-descent learning procedures for networks," in *Eighth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ, 1986, pp. 823-831.

20 C. Jacob, J. Rehder, J. Siemandel, and A. Friedmann, "XNeuroGene: a system for evolving artificial neural networks," in Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '95, 1995, pp. 436-439.

21 S. Baluja, "Evolution of an artificial neural network based autonomous land vehicle controller," *Systems, Man and Cybernetics, Part B, IEEE Transactions on,* vol. 26, pp. 450-463, 1996.

22 H. Seung-Soo and G. S. May, "Optimization of neural network structure and learning parameters using genetic algorithms," in *Proceedings Eighth IEEE International Conference on Tools with Artificial Intelligence.* , 1996, pp. 200-206.

23 P. G. Harrald and M. Kamstra, "Evolving artificial neural networks to combine financial forecasts," *IEEE Transactions on Evolutionary Computation,* vol. 1, pp. 40-52, 1997.

24 X. Yao and Y. Liu, "A new evolutionary system for evolving artificial neural networks," *Neural Networks, IEEE Transactions on,* vol. 8, pp. 694-713, 1997.

25 C. Zhang, Y. Li, and H. Shao, "A new evolved artificial neural network and its application," in *Proceedings of the 39th IEEE Conference on Decision and Control.*, 2000, pp. 1065-1068 vol.2.

26 S. W. Moon and S. G. Kong, "Block-based neural networks," *IEEE Transactions on Neural Networks,* vol. 12, pp. 307-317, 2001.

27 K. Kyung-Joong and C. Sung-Bae, "Evolving artificial neural networks for DNA microarray analysis," in *The 2003 Congress on Evolutionary Computation, CEC '03*, 2003, pp. 2370-2377 Vol.4.

28 D. A. Miller, R. Arguello, and G. W. Greenwood, "Evolving artificial neural network structures: experimental results for biologically-inspired adaptive mutations," in *Congress on Evolutionary Computation. CEC2004*, 2004, pp. 2114-2119 Vol.2.

29 M. N. H. Siddique, and M. O. Tokhi, “Training neural networks: back-propagation vs. genetic algorithms,” in *Proceedings of International Joint Conference on Neural Networks (IJCNN '01)*, 2001, pp. 2673-2678 vol4.

30 J. N. H. Heemskerk, "Overview of Neural Hardware.," in *Neurocomputers for Brain-Style Processing. Design, Implementation and Application*: Unit of Experimental and Theoretical Psychology, Leiden University,The Netherlands., 1995.

31 J. D. Hadley and B. L. Hutchings, "Design methodologies for partially reconfigured systems," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines.* , 1995, pp. 78-84.

32 R. Gadea, J. Cerda, F. Ballester, and A. Macholi, "Artificial neural network implementation on a single FPGA of a pipelined on-line back-propagation," 2000, pp. 225-230.

33 P. James-Roxby and B. A. Blodget, "Adapting constant multipliers in a neural network implementation," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 335-336.

34 A. Perez-Uribe and E. Sanchez, "Speeding-up adaptive heuristic critic learning with FPGA-based unsupervised clustering," in *IEEE International Conference on Evolutionary Computation.*, 1997, pp. 685-689.

35 A. Perez-Uribe and E. Sanchez, "FPGA implementation of an adaptable-size neural network," in *Proceedings of The VI International Conference on Artificial Neural Networks, ICANN96*, 1996.

36 A. Perez-Uribe and E. Sanchez, "Implementation of neural constructivism with programmable hardware," in *International Symposium on Neuro-Fuzzy Systems, AT'96.*, 1996, pp. 47-54.

37 K. R. Nichols, M. A. Moussa, and S. M. Areibi, "Feasibility of Floating-Point Arithmetic in FPGA based Artificial Neural Networks," in *15th International Conference on Computer Applications in Industry and Engineering*, San Diego, California, 2002.

38 M. Marchesi, G. Orlandi, F. Piazza, and A. Uncini, "Fast neural networks without multipliers," *Neural Networks, IEEE Transactions on*, vol. 4, pp. 53-62, 1993.

39 A. F. Murray and A. V. W. Smith, "Asynchronous VLSI neural networks using pulse-stream arithmetic," *Solid-State Circuits, IEEE Journal of*, vol. 23, pp. 688-697, 1988.

40 G. P. K. Economou, E. P. Mariatos, N. M. Economopoulos, D. Lymberopoulos, and C. E. Goutis, "FPGA implementation of artificial neural networks: an application on medical expert systems," in *Proceedings of the Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems.*, 1994, pp. 287-293.

41 V. Salapura, "Neural networks using bit stream arithmetic: a space efficient implementation," in *IEEE International Symposium on Circuits and Systems. ISCAS '94.*, 1994, pp. 475-478 vol.6.

42 L. M. Reyneri, "Theoretical and implementation aspects of pulse streams: an overview," 1999, pp. 78-89.

43 H. Hikawa, "A new digital pulse-mode neuron with adjustable activation function," *Neural Networks, IEEE Transactions on*, vol. 14, pp. 236-242, 2003.

44 L. Mintzer, "Digital filtering in FPGAs," in 1994 Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers., 1994, pp. 1373-1377 vol.2.

45 T. Szabo, L. Antoni, G. Horvath, and B. Feher, "A full-parallel digital implementation for pre-trained NNs," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, IJCNN 2000.*, 2000, pp. 49-54 vol.2.

46 J. L. Holt and T. E. Baker, "Back propagation simulations using limited precision calculations," 1991, pp. 121-126 vol.2.

47 Xilinx Inc. http://www.xilinx.com/

48 S. Guccione and M. J. Gonzalez, "Classification and Performance of Reconfigurable Architectures," in *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*: Springer-Verlag, 1995.

49 D. F. Wolf, R. A. F. Romero, and E. Marques, "Using Embedded Processors in Hardware Models of Artificial Neural Networks," in *Proceedings of SBAI - Simpósio Brasileiro de AutomaoInteligente*, 2001, pp. 78-83.

50 M. Krips, T. Lammert, and A. Kummert, "FPGA implementation of a neural network for a real-time hand tracking system," in *Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications*, 2002, pp. 313-317.

51 S. W. Moon and S. G. Kong, "Pattern recognition with block-based neural networks," in *Proceeding of International Joint Conference on Neural Networks (IJCNN-2002)*, 2002, pp. 992-996.

52 S. G. Kong, "Time series prediction with evolvable block-based neural networks," in *Proceeding of International Joint Conference on Neural Networks (IJCNN-2004)*, 2004, pp. 1579-1583 vol.2.

53 W. Jiang, S. G. Kong, and G. D. Peterson, "ECG Signal Classification using Block-based Neural Networks," in *Proceeding of International Joint Conference on Neural Networks (IJCNN-2005)*, 2005, pp. 992-996.

54 S. Kothandaraman, "Implementation of Block-based Neural Networks on Reconfigurable Computing Platforms," in *Electrical and Computer Engineering Department*. vol. MS Knoxville: University of Tennessee, 2004.

55 S. Merchant, G. D. Peterson, and S. G. Kong, "Intrinsic Embedded Hardware Evolution of Block-based Neural Networks," in *Proceedings of Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, Nevada, 2006.

56 S. Merchant, G. D. Peterson, S. K. Park, and S. G. Kong, "FPGA Implementation of Evolvable Block-based Neural Networks," in *Proceedings of IEEE Congress on Evolutionary Computation*, Vancouver, Canada, 2006.

57 Amirix Systems Inc. http://www.amirix.com/

58 J. Peterson, Petri Net Theory and the Modeling of Systems: Prentice Hall, 1981.

59 W. Reisig, *A Primer in Petri Net Design*: Springer-Verlag, 1992.

60 R. Fisher, "The use of multiple measurements in taxonomic problems," *Annals Eugen.*, vol. 7, pp. 179-188, 1936.

61 E. R. Hruschka and N. F. F. Ebecken, "Rule extraction from neural networks: modified RX algorithm," 1999, pp. 2504-2508 vol.4.

62 Y. Jewajinda and P. Chongstitvatana, "A Cooperative Approach to Compact Genetic Algorithm for Evolvable Hardware," 2006, pp. 2779-2786.

63 J. C. Gallagher, S. Vigraham, and G. Kramer, "A family of compact genetic algorithms for intrinsic evolvable hardware," *IEEE Transactions on Evolutionary Computation,* vol. 8, pp. 111-126, 2004.

64 Jeff Hawkins and Sandra Blakeslee, *On Intelligence*. Times Books. 2004.

65 D. Wei and J. G. Harris, "Signal reconstruction from spiking neuron models." in IEEE International Symposium on Circuits and Systems (*ISCAS), vol 5*, 2004, pp. 353–356.

66 J. Wijekoon, P. Dudek, "Spiking and Bursting Firing Patterns of a Compact VLSI Cortical Neuron Circuit." *Proceedings of International Joint Conference on Neural Networks*. Orlando, FL. 2007.

67 A. Van Shaik, "Building blocks for electronic spiking neural networks" Neural Networks 14(2001) 617-628.

68 Djurfeldt, M., Lundqvist, M., Johansson, C., Rehn, M., Ekeberg, Ö., & Lansner, A. (2008). Brain-scale simulation of the neocortex on the IBM Blue Gene/L supercomputer. *IBM J. Res. & Dev*., 52 (1/2), 31–41.

69 Joseph Lancaster, Jeremy Buhler, and Roger D. Chamberlain, "Acceleration of ungapped extension in Mercury BLAS" *Microprocessors and Microsystems*. Volume 33, Issue 4, June 2009, Pages 281-289.

70 Eddy, SR. Profile hidden Markov models. Bioinformatics, 14:755-763, 1998.

71 Landman, J, Ray, J, Walters, JP. Accelerating HMMer searches on Opteron processors with minimally invasive recoding. IEEE AINA, 2006.

72 Krogh, A, Brown, M, Mian, IS, Sjolande, K, and Haussler D. Hidden Markov models in computational biology. Application to protein modeling. Journal of Molecular Biology 235:1501-31, 1994.

73 Hughey, R, and Krogh A. Hidden Markov models for sequence analysis: extension and analysis of basic method. Comput. Appl. Biosci. 12:95-107, 1996.

74 Oehmen, C, and Nieplocha, J. ScalaBLAST: A scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis.

75 Costa, RLC, and Lifschitz, S. Database allocation strategies for parallel BLAST evaluation on clusters. Distributed and Parallel Databases, 13, 99-127, 2003.

76 National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/

77 Bateman A, Birney E, Cerruti L, Durbin R, Etwiller l, Eddy SR, Griffiths-Jones S, Howe KL, Marshall M, Sonnhammer ELL. The Pfam protein families database. Nucleic acids Research, 30:276-280, 2002.

78 Message Passing Interface Forum, MPI: A Message Passing Interface Standard. May 1994.

79 Sunderam, V.S. PVM: A Framework for Parallel Distributed Computing. *Concurrency, Practice and Experience*, 1990.

80 Srinivasan, U, Che, PS, Diao, Q, Lim CC, Li, E, Chen, Y, Ju, R, and Zhang Y. Characterization and analysis of HMMER and SVM-RFE parallel bioinformatics applications. IEEE 2005.

81 Wun, B, Buhler, J, and Crowley, P. Exploiting coarse-grained parallelism to accelerate protein motif finding with a network processor. Parallel Architecure and compilation Techniques, 173-184, IEEE, 2005.

82 Horn, DR, Houston, M, and Hanrahan, Pat. ClawHMMER: A streaming HMMer-Search implementation. Proc. IEEE Supercomputing 2005.

83 Madisetty, RP, Buhler J, Chamberlain, RD, Franklin, MA, and Harris, B. Accelerator design for protein sequence HMM search. ICS, 2006.

84 Oliver, TF, Schmidt, B, Yanto, J, and Maskell, DL. Accelerating the Viterbi Algorithm for Profile Hidden Markov Models Using Reconfigurable Hardware. Lect. Notes Comput. Sci., vol. 3991, 2006, pp. 522-29.

85 Walters, JP, Meng, X, Chaudhary, V, Oliver, T, Yeow, LY, Schmidt, B, Nathan, L, and Landman, J. MPI-HMMER-Boost: Distributed FPGA Acceleration. *Journal of VLSI signal processing.* 2007

86 Darole,R, Walter, JP, and Chaudhary V. Improving MPI-HMMER’s Scalability With Parallel I/O. Technical Report #2008-11, May 22, 2008.

87 Chukkapalli, G, Guda, C, and Subrananiam, S. SledgeHMMER: A Web Server for Batch Searching the Pfam Database. Nucleic Acids Res., Vol. 32, 2004.

88 Rekapalli, B.P. Automated Genome-Wide Protein Domain Exploration. Ph.D. dissertation, The University of Tennessee, Department of Electrical Engineering and Computer Science. Dec. 2007.

89 Top500 Supercomputers of the world. http://www.top500.org/

90 Smith, M. C., J. Vetter, and S. Alam, “Scientific Computing Beyond CPUs: FPGA Implementations of Common Scientific Kernels,” *MAPLD*, 2005.

91 Melissa C. Smith, Steven L. Drager, Lt. Louis Pochet, Gregory D. Peterson, “High Performance Reconfigurable Computing Systems.” In *Proceedings of 2001 IEEE Midwest Symposium on Circuits and Systems*. August 2001.

92 Gregory D. Peterson, “Programming High Performance Reconfigurable Computers” In *SPIE’s International Symposium on The Convergence of Information Technologies and Communications*. Denver, CO, August 2001.

93 Keith D. Underwood, “FPGAs vs. CPUs: trends in peak floating-point performance.” *FPGA*. 2004.

94 Junqing Sun, Gregory D. Peterson, and Olaf Storaasli “High Performance Mixed-Precision Linear Solver for FPGAs.” *IEEE Transactions on Computers*, **57**(12):1614-1623, December 2008

95 Akila Gothandaraman, Gregory D. Peterson, G. Lee Warren, Robert J. Hinde, Robert J. Harrison, “FPGA Acceleration of a Quantum Monte Carlo Application”, *Parallel Computing*, **34**(4-5):278-291,May 2008.

96 E. A. Lee and A. Sangiovanni-Vincentelli, ``A Framework for Comparing Models of Computation,", *IEEE Transactions on CAD,* Vol. 17, No. 12, December 1998.

97 E. A. Lee, ``Overview of the Ptolemy Project", ERL Technical Report UCB/ERL No. M98/71, University of California, Berkeley, CA 94720, November 23, 1998. *This paper has been superseded by Technical Memorandum UCB/ERL M01/11, March 6, 2001*, also entitled Overview of the Ptolemy Project

98 Edward A. Lee, "Overview of the Ptolemy Project," *Technical Memorandum UCB/ERL M01/11*, University of California, Berkeley, March 6, 2001.

99 Edward A. Lee, "Computing for Embedded Systems," *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 21-23, 2001.

100 http://ptolemy.berkeley.edu

101 *OpenMP Application Programming Interface*. 2008, OpenMP Architecture Review Board.

102 Wall, D.W., *Limits of Instruction-Level Parallelism*. 1993, Digital Equipment Company: Palo Alto.

103 John D. Owens, M.H., David Luebke, Simon Green, John E. Stone, and James C. Phillips, *GPU Computing.* Proceedings of the IEEE, 2008. **96**(5): p. 879-899.

104 AMD Stream Computing User Guide. 2008, AMD.

105 Ian Buck, T.F., Daniel Horn, Jeremy Sugerman, *Brook for GPUs*. 2003.

106 Dally, W. *Stream Computing*. [Lecture] 2008 June 10, 2008 [cited 2009 March 23]; Available from: http://www.youtube.com/watch?v=8x7OqjUNbyo.

107 NVIDIA CUDA home page, http://www.nvidia.com/object/cuda_home.html

108 Association for Computing Machinery Student Research Contest Grand Finals, See http://www.ece.utk.edu/%7Ejsun5/acmsrc07.htm

109 Association for Computing Machinery Student Research Contest Grand Finals, See http://www.acm.org/src/gothandaraman/gothandaraman.html

110 Anderson, J., Becker, T., Blodget, B., Lysaght, P., and Sedcole, P. Modular Dynamic Reconfiguration in Virtex FPGAs. *IEE Proceedings online no. 20050176. IEE Proc. Comput. Digit. Tech*., Vol. 153, No. 3, May 2006.

111 Xilinx JBITS, http://www.xilinx.com/products/jbits/index.htm

112 Smith, M. C., and G. D. Peterson, "Parallel Application Performance on Shared High Performance Reconfigurable Computing Resources," *Perform. Eval.*, Vol. 60/1-4, pp. 107-125, May 2005.

113 Junqing Sun and Gregory D. Peterson, “Effective Execution Time Estimation for Heterogeneous Parallel Computing.” *International Conference on Parallel and Distributed Computing Systems.* Sept 2006.

114 Kwok, Y., “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors,” *ACM Comput. Surv.*, vol. 31:4, pp. 406-471, Dec. 1999.

115 Garey, M. R. and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman & Co., NY, 1979.

116 Blodget, B., Keller, E., McMillan, S., James-Roxby. P., and Sundararajan, Prasanna. A Self-Reconfiguring Platform. *In Proceedings of Field Programmable Logic and Applications, (2003)* 565-574.

117 Becker, T., Cheung, P.Y.K., and Luk, W. 2007. Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration. *2007 International Symposium on Field Programmable Custom Computing Machines*. IEEE -7695-2940-2/07. DOI 10.1109/FCCM.2007.51

118 Dorairaj, N., Goosman, M., and Shiflet, E. How to Take Advantage of Partial Reconfiguration in FPGA Designs. Xilinx Corp 2006. http://www.pldesignline.com/179100555

119 Athanas, P. M., Craven, S. D., High-Level Specification of Runtime Reconfigurable Designs. *Proceedings of the 2007 International Conference on Engineering of Reconfigurable Systems Algorithms (ERSA).* 2007, Las Vegas Nevada, USA, June 24-28, 2007.

120 Conger, C. George, A. D., and Gordon-Ross, Ann. Design Framework for Partial Run-Time FPGA Reconfiguration. *Proceedings of 2008 Internation Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, July 14-17, 2008.

121 Mosley, R. Reconnetics: A System for the Dynamic Implementation of Mobile Hardware Processes in FPGAs. *Communicating Process Architectures 2002.* 177-190.

122 D. Bell and L. La Padula, “Secure Computer Systems: Mathematical Foundations and Model.” MITRE Report, MTR-2547 v2. Nov. 1973.

123 Charles P. Pfleeger and Shari Lawrence Pfleeger, *Security in Computing*, 4th edition, Prentice Hall, 2007.

124 Casavant, T. L., "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. on Software Eng.*, vol. SE-14:2, pp. 141-154, Feb.1988.

125 El-Rewini, H. and T. G. Lewis, *Task Scheduling in Parallel Distributed Systems*, Prentice Hall, 1994.

126 Kwok, Y., “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors,” *ACM Comput. Surv.*, vol. 31:4, pp. 406-471, Dec. 1999.

127 Topcuoglu, H., S. Hariri, and M. Wu, “Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing,” *IEEE Trans. Parallel Distr. Syst.*, vol. 13:3, pp. 260-274, Mar. 2002.

128 Govindan, V. and M. A. Franklin, “Application Load Imbalance on Parallel Processors,” *Proc. IPPS96*, 1996.

129 Cantu-Paz, E., “Designing Efficient Master-Slave Parallel Genetic Algorithms,” *Proc. 3rd Annual Genetic Programming: Conference,* San Francisco, CA, 1998.

130 Hou, E., N. Ansari, and H. Ren, “A Genetic Algorithm for Multiprocessor Scheduling”, *IEEE Trans. Parallel Distr. Syst.,* Vol. 5:2, pp. 113-120, Feb. 1994.

131 Basney, J., B. Raman, and M. Livny, “High Throughput Monte Carlo,” *SIAM PPSC99,* San Antonio, TX, 1999.

132 Smith, M.C., "Analytical Modeling of High Performance Reconfigurable Computers: Prediction and Analysis of System Performance." Doctor of Philosophy University of Tennessee, Knoxville, TN, 2003

133 G. D. Peterson and R. D. Chamberlain, Parallel Application Performance in a Shared Resource Environment. IEE Distributed Systems Engineering Journal, August 1996.

134 G. D. Peterson and R. D. Chamberlain, Sharing Networked Workstations: A Performance Model. 6th IEEE Symposium on Parallel and Distributed Processing. pp. 308-315, Dallas, TX, October 1994.

135 B. W. Weide. Analytic Models to Explain the Anomalous Behavior of Parallel Programs. In International Conference on Parallel Processing, pp. 183-187, 1981.

136 A.H. –S. Ang and W. H. Tang, Probability Concepts in Engineering Planning and Design Vol. II. Rainbow Bridge, 1984.

137 S. Madala and J. B. Sinclair, Performance of Synchronous Parallel Algorithms with Regular Structures. IEEE Transactions on Parallel and Distributed Systems, 2(1): 105-116, January 1991.

138 V. D. Agrawal and S. T. Chakraadhar, Performance analysis of synchronized iterative algorithm on multiprocessor systems, IEEE Trans. Parallel and Distributed Systems, VOL. 3, NO. 6, 739-745, Nov. 1992.

139 Leong, P. H. W., Leong, M. P., Cheung, O. Y. H., Tung, T., Kwok, C. M., Wong, M. Y., and Lee, K. H., “Pilchard - A Reconfigurable Computing Platform With Memory Slot Interface,” *IEEE FCCM*, 2001.

140 Celoxica, Inc., http://www.celoxica.com

141 Nallatech, Inc., http://www.nallatech.com

142 Cray, Inc., http://www.cray.com.

143 SGI, Inc., http://www.sgi.com

144 SRC Computers, Inc., http://www.srccomp.com

145 BLAS (Basic Linear Algebra Subprograms), http://www.netlib.org/blas/

146 NIST, “Guideline for Implementing Cryptography in the Federal Government,” NIST SP800-21, http://csrc.nist.gov/publications.

147 Cook, S.A., “The Complexity of Theorem-proving Procedures,” *ACM Symp. Theory of Comp.,* 1971.

148 ClearSpeed, Inc., http://www.clearspeed.com

149 Peterson, G. D., "Parallel Application Performance on Shared, Heterogeneous Workstations." Doctor of Science Washington University, Saint Louis, MO, 1994.

150 Atallah, M. J., C. L. Black, D. C. Marinescu, H. J. Segel, and T. L. Casavant, "Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations," *J. Parallel Distr. Comp.*, vol. 16, pp. 319-327, 1992.

151 Kant, K., Introduction to Computer System Performance Evaluation, McGraw-Hill, Inc., NY, 1992.

152 J. Jacod and P. Protter, Probability Essentials, Springer, 2000.

153 G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors, Volume I, Prentice Hall, 1988.

154 M. L. Bailey, Jr. J. V. Briner, and R. D. Chamberlain. "Parallel Logic Simulation of VLSI Systems." ACM Computing Surveys, 26(3): 255--294, September 1994.

155 K.T. Cheng and V. D. Agrawal, Unified Methods for VLSI Simulation and Test Generation. Boston: Kluwer Academic, 1989.

156 P. Agrawal, “Concurrency and communication in hardware simulators,” IEEE Trans. Comput. –Aided Design, vol. CAD-5, pp. 617-623, Oct. 1986.

157 L. Soule and T. Blank, “Statistics for parallelism and abstraction level in digital simulation,” in Proc. 24th ACM/IEEE Design Automat. Conf., 1987, pp. 588-591.

158 M. A. Driscoll and W. R. Daasch, “Accurate Predictions of Parallel Program Execution Time,” Journal of Parallel and Distributed Computing. Vol. 25, No. 1, Feb, 1995.

159 R. E. Barlow and F. Proschan, Statistical Theory of Reliability and Life Testing. New York. 1975

160 Nihal Yazici-Pekergin and Jean-Marc Vincent, “Stochastic Bounds on Execution Times of Parallel Programs,” *IEEE Trans. on Software Eng.*, vol. 17, No. 10, Oct. 1991.

8 ACRONYMS USED

AA	Amino Acid
ACS	Adaptive Computing Systems
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
API	Application Programmers Interface
ASIC	Application Specific Integrated Circuit
AWACS	Airborne Warning and Control System
BbNN	Block-based Neural Network
BYU	Brigham Young University
C4ISR	Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance
CAD	Computer Aided Design
CAL	Computer Abstraction Layer
CDF	Cumulative Distribution Function
CPLD	Complex Programmable Logic Device
CPS	Computations Per Second
CSP	Communicating Sequential Processors
CUDA	Computer Unified Device Architecture
DMA	Direct Memory Addressing
DNA	Deoxyribonucleic Acid
DSP	Digital Signal Processor
ECG	ElectroCardioGram
EDK	Embedded Development Kit
EHW	Evolvable Hardware
EMMA	Expected Maximum Mean Approximation
EMV	Expected Maximum Value
FPGA	Field Programmable Gate Array
FPSLIC	Field Programmable System Level Integrated Circuits

GA	Genetic Algorithm
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HMM	Hidden Markov Model
HPC	High Performance Computing
HPCS	High Productivity Computing Systems
HPRC	High Performance Reconfigurable Computing
HT	Hyper-Transport
i.i.d.	Independent identically distributed
I/O	Input/Output
ICN	Inter-Connection Network
IL	Intermediate Language
ILP	Instruction Level Parallelism
JBI	Joint Battlespace Infosphere
JHDL	Java Hardware Description Language
JTIDS	Joint Tactical Information Distribution System
LUT	Look Up Table
MC	Monte Carlo
MCPS	Millions of Connections Per Second
MIMD	Multiple Instructions Multiple Data
MIPS	Millions of Instructions Per Second
MKL	Math Kernel Library
MLP	Multi-Layer Perceptron
MMX	Multi-Media eXtension, Matrix Math eXtension
nr	non-redundant
OBM	On Board Memory
OPB	On-Chip Peripheral Bus
ORNL	Oak Ridge National Laboratory

P/T	Place/Transition
PAL	Programmable Arrays of Logic
PCI	Peripheral Component Interconnect
PDF	Probability Density Function
PE	Processing Element
PLB	Programmable Local Bus
PLD	Programmable Logic Device
PPC	Power PC
PRC	Partially Reconfigurable Circuit
PSOC	Programmable System On a Chip
PVM	Parallel Virtual Machine
QPI	Quick Path Interconnect
RAM	Random Access Memory
RC	Reconfigurable Computing
RS3	Runtime Support for Reliable, Secure, Reconfigurable Systems
RTL	Register Transfer Language
SAT	Satisfiability (Boolean Satisfiability)
SDRAM	Synchronous Dynamic Random Access Memory
SIA	Synchronous Iterative Algorithm
SIMD	Single Instruction Multiple Data
SLAAC	System Level Applications of Adaptive Computing

SMP	Symmetric Multi-Processing
SOC	System On a Chip
SRAM	Static Random Access Memory
SSE	Streaming SIMD Extensions
TAP	Test Access Port
TLP	Thread Level Parallelism
TRIPS	Tera-op Reliable Intelligently advanced Processing System
UAV	Unmanned Aerial Vehicle
UCAV	Unmanned Combat Aerial Vehicle
UT	University of Tennessee
VHDL	Very High Speed Integrated Circuits Hardware Description Langauge
VLSI	Very Large Scale Integration
VSIPL	Vector, Signal and Image Processing Library