DISTRIBUTED CONFIGURATION MANAGEMENT FOR RECONFIGURABLE CLUSTER COMPUTING

By

AJU JACOB

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2004

This document is dedicated to the God for giving me the strength to finish it.

ACKNOWLEDGMENTS

I thank Dr. Alan George, Ian Troxel, Raj Subramaniyan, Burt Gordon, Hung-Hsun Su, Dr. Sarp Oral, and all other members of the HCS Lab at the University of Florida for all their guidance and direction. I thank Dr. Herman Lam, Dr. Renato J. Figueiredo, and Dr. Jose A. B. Fortes for serving on my thesis committee. I thank my parents and sister for the support and encouragement. I would also like to thank Alpha-Data, Tarari, and Celoxica for their RC Boards. I thank Dolphin Inc. for its donation of SCI cards. I thank Intel and Cisco for their donation of Cluster resources.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

DISTRIBUTED CONFIGURATION MANAGEMENT
FOR RECONFIGURABLE CLUSTER COMPUTING

By

Aju Jacob

December 2004

Chair: Alan D. George
Major Department: Electrical and Computer Engineering

Cluster computing offers many advantages as a highly cost-effective and often scalable approach for high-performance computing (HPC) in general, and most recently as a basis for hardware-reconfigurable systems, known as Reconfigurable Computing (RC) systems. To achieve the full performance of reconfigurable HPC systems, a run-time configuration manager is required. Centralized configuration services are a natural starting point but tend to limit performance and scalability. For large-scale RC systems, the configuration manager must be optimized for the system topology and management scheme. This thesis presents the design of a configuration manager within the Comprehensive Approach to Reconfigurable Management Architecture (CARMA) framework created at the University of Florida, in addition to several distributed configuration management schemes that leverage high-speed networking. The experimental results from this thesis highlight the effects of the design of this configuration manager and provide a comprehensive performance analysis of three of the

proposed management schemes. In addition, the experiments explore and compare the scalability of these management schemes. These results show the configuration manager designs have little overhead on the system, when the system is unstressed. Finally, larger system-sizes are explored with an analytical model and scalability projections that investigate performance beyond available testbeds. The model shows that a hierarchical management scheme is needed for the configuration manager to provide lower bandwidth consumption and completion latency.

CHAPTER 1
INTRODUCTION

Traditional computing spans a variety of applications due to its high flexibility, but is rather inefficient dealing with fine-grain data manipulation. Reconfigurable Computing (RC) attempts to perform computations in hardware structures to increase performance while still maintaining flexibility. Research has shown that certain applications, such as cryptanalysis, pattern matching, data mining, etc., reap performance gains when implemented in an RC system. However, this approach is valid only if there is an efficient method to reconfigure the RC hardware that does not overshadow the performance gains of RC.

Loading a different configuration file alters the computation of an RC device, typically an Field-Programmable Gate Array (FPGA). To increase the efficiency of RC systems, the RC device is typically coupled with a General-Purpose Processor (GPP). GPPs are apt at control and I/O functions while the RC hardware handles fine-grain computations well. In a typical Commercial-off-the-Shelf (COTS) environment, the RC hardware takes the form of FPGA(s) on boards connected to the host processor through the PCI bus. Computation, control communication, and configurations are transferred over the PCI bus to the board. This coupling of RC board(s) to the GPP in a conventional computing node yields a single-node COTS RC system.

A trend toward high-performance cluster computing based largely on COTS technologies has recently developed within the reconfigurable computing community. The creation of a 48-node COTS RC cluster at the Air Force Research Laboratory in

Rome, NY [1] is evidence of this trend. Indeed, High-Performance Computing (HPC) has the potential to provide a cost-effective, scalable platform for high-performance parallel RC. However, while the addition of RC hardware has improved the performance of many stand-alone applications, providing a versatile multi-user and multitasking environment for clusters of RC and conventional resources imposes additional challenges.

To address these challenges in HPC/RC designs, the HCS Research Lab at the University of Florida proposes the Comprehensive Approach to Reconfigurable Management Architecture (CARMA) framework [2, 3, 4]. CARMA provides a framework to develop and integrate key components as shown in Figure 1. With CARMA, the RC group at Florida seeks to specifically address key issues such as: dynamic RC-hardware discovery and management; coherent multitasking in a versatile multi-user environment; robust job scheduling and management; fault tolerance and scalability; performance monitoring down into the RC hardware; and automated application mapping into a unified management tool. This thesis focuses on the configuration management portion of CARMA's RC cluster management module in order to highlight the design and development of distributed configuration management schemes.

Many of the "new" challenges CARMA addresses bear a striking resemblance to traditional HPC problems, and will likely have similar solutions, however others have very little correspondence whatsoever. The task of dynamically providing numerous configurations to distributed RC resources in an efficient manner is one such example. At first glance, changing computation hardware during execution in RC systems has no

traditional HPC analog. However, future development of an RC programming model to allow the reuse of configuration files through run-time libraries resembles the concept of code reuse in tools such as the International Mathematical and Statistical Library (IMSL). This collection of mathematical functions abstracts the low-level coding details from developers by providing a means to pass inputs between predefined code blocks [5]. While tools like IMSL provide this abstraction statically, future RC programming models will likely include run-time library access. Core developers are providing the basis for such libraries [6].

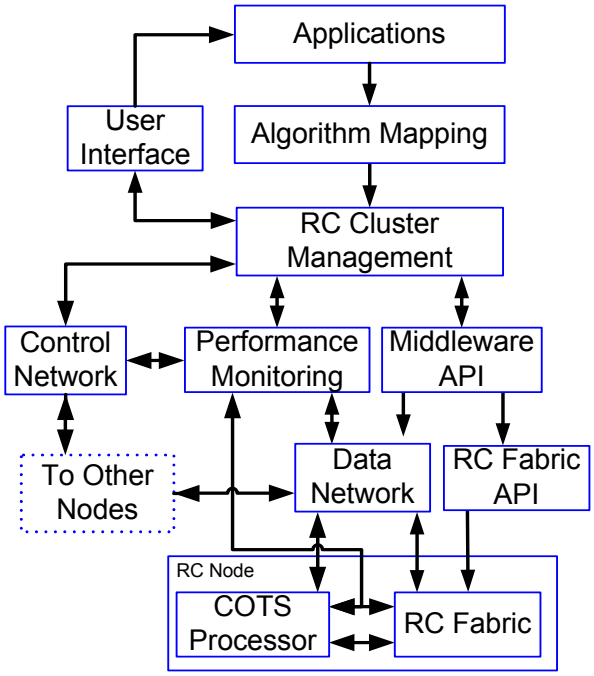

Figure 1. The CARMA Framework. This Thesis focuses on Configuration Management located in the RC Cluster Management block.

While the programming model aspect of configuration files could relate to traditional HPC, other aspects may not. For example, one might equate the transmission of configuration files to data staging by assuming they are simply another type of data necessary to be loaded onto a set of nodes at run-time. One reason this approach does not

hold is the adaptable nature of configuration files. Recent work has suggested an adaptive-algorithms approach to RC in which tasks may require the run-time use of hundreds of configuration-file versions [7]. Traditional HPC systems today are not designed to recompile code at run-time much less do so hundreds of times per job.

Another reason configuration files differ is that, while relatively small, the amount of high-speed cache dedicated to their storage is typically small, if existent. While some custom FPGAs allow up to four configurations to be stored on chip [8], systems that do not allocate on-chip or on-board memory cannot preemptively "stage" configurations at all. As demonstrated, configuration management and other issues need to be considered to achieve a versatile platform for RC-based HPC, especially if such systems may one day include grid-level computation where communication latencies can lead to significant performance penalties.

The remaining chapters of this thesis are organized as follows. Chapter 2 provides a background of past work in configuration management services while Chapter 3 describes the design and development of CARMA's configuration manager. Chapter 4 provides a detailed discussion of the experimental setup while Chapter 5 presents and analyses the experimental results. Chapter 6 presents the projected scalability of CARMA's configuration manager. Finally, Chapter 7 describes conclusions and future work.

CHAPTER 2
BACKGROUND

Traditionally, computational algorithms are executed primarily in one of two ways: with an Application-Specific Integrated Circuit (ASIC) or with a software-programmed General-Purpose Processor (GPP) [9]. ASICs are fabricated to perform a particular computation and cannot be altered, however, ASIC designs provide fast and efficient execution. GPPs offer far greater flexibility because changing software instructions changes the functionality of the GPP. This flexibility comes at the overhead cost of fetching and decoding instructions. RC is intended to achieve the flexibility of GPPs and the performance of ASICs simultaneously. Loading a different configuration file alters the computation that an RC device, typically a FPGA, performs. Moreover, RC devices, such as FPGAs, are composed of logic-blocks that operate at hardware speeds.

Configuration files dictate the algorithm or function that FPGA logic-blocks perform. The FPGA-configuration file is a binary representation of the logic design, used to configure the FPGA. Configuration file sizes have increased considerably over the last ten years. Xilinx's Virtex-series FPGAs have configuration-file sizes that range from 80 kB files to 5.3 MB for its high-end Virtex-II Pro [10]. The configuration-file size in general is a constant for any given device, regardless of the amount of logic in it, since every bit in the device gets programmed in full-chip configurations, whether it is used or not. However, the Virtex series supports partial reconfiguration, which is a mechanism where only logic on part of the FPGA is changed. Virtex-series chips can perform partial configuration without shutting down or disturbing processing on other regions of the

FPGA. Albeit partial configuration is supported at the chip level, there exist no COTS-based boards that have software-API support for partial reconfiguration.

Applications that exhibit parallelism and require fine-grain data manipulations are accelerated greatly by RC solutions. Data encryption is one such area that has benefited by RC. Serpent [11], DES [12], and Elliptic-Curve Cryptography [13] applications have all shown speedups as implemented in FPGAs when compared to conventional processors. Another application area that exhibits significant speedup when using RC is space-based processing. Hyperspectral imaging, a method of spectroscopy from satellites, can show frequency details that reveal images not visible to the human eye. Hyperspectral sensors can now acquire data in hundreds of frequency windows, each less than 10 nanometers in width, yielding relatively large data cubes for space-based systems (i.e. over 32 Mbytes) [14]. It is predicted that an RC implementation would have tremendous speedup over the today's conventional processor using Matlab.

As mentioned in Chapter 1, the Air Force Research Laboratory in Rome, NY has created a 48-node RC cluster [1], thereby combining HPC and RC. Another example of the combination of HPC and RC exists at Virginia Tech with their 16-node "Tower of Power" [15]. More recently, the HCS Lab at the University of Florida has implemented an 9-node RC cluster [2]. A System-Area Network (SAN), more specifically Myrinet in Rome and Virginia Tech and Scalable Coherent Interconnect (SCI) in Florida's RC cluster, interconnects the nodes of these clusters. Myrinet and SCI as well as other High-Performance Networks (HPNs) including QsNet, and Infiniband, provide a high-throughput, low-latency network between HPC nodes. These networks are ideal to

transfer latency-critical configuration files, which can be several MBs for high-end FPGAs.

In addition to the cost advantages of COTS-based cluster computing, COTS-based RC boards facilitate the creation of RC systems. There are a variety of RC boards commercially available, varying in FPGA size, on-board memory, and software support. The most common interface for COTS-based RC boards is PCI. Two PCI-based RC boards presented in this chapter are Celoxica's RC1000 [16] and Tarari's Content Processing Platform (CPP) [17].

The RC1000 board provides high-performance, real-time processing capabilities and provides dynamically reconfigurable solutions [16]. Figure 2 depicts the architecture of the RC1000. The RC1000 is a standard PCI-bus card equipped with a Xilinx Virtex device and SRAM memory directly connected to the FPGA. The board is equipped with two industry-standard PMC connectors for directly connecting other processors and I/O devices to the FPGA. Furthermore, a 50-pin unassigned header is provided for either inter-board communication or connecting custom interfaces. Configuration of the RC1000 is through the PCI bus directly from the host.

Tarari has developed the dynamically reprogrammable content-processing technology to tackle the compute-intensive processing and flexibility requirements of the Internet-driven marketplace [17]. Figure 3 depicts the architecture of the Tarari CPP. The Tarari CPP has two Content-Processing Engines (CPE), each of which is a Xilinx Virtex-II FPGA. In addition, a third FPGA is a Content-Processing Controller (CPC), which handles PCI and inter-CPE communication, as well as configuration and on-board memory access. The DDR SDRAM is addressed by both CPEs, thus creating a shared-

memory scheme for inter-FPGA communication. The two CPEs enable parallelism and more complex processing, compared to the RC-1000. The Tarari boards have the ability to store configurations in on-board memory thereby decreasing the configuration latency by eliminating costly PCI-bus transfers.

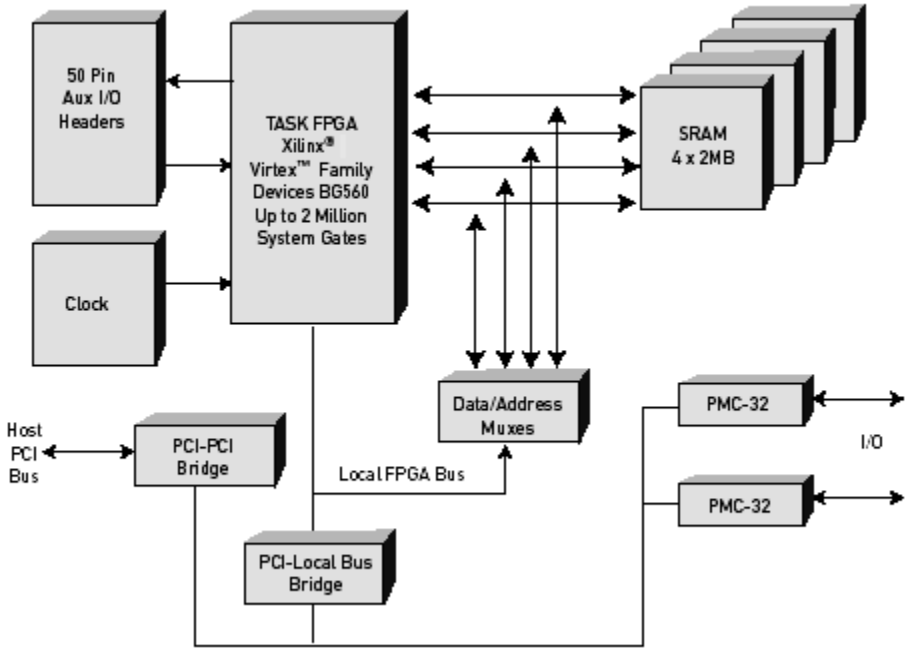

Figure 2. Celoxica's RC1000 Architecture.

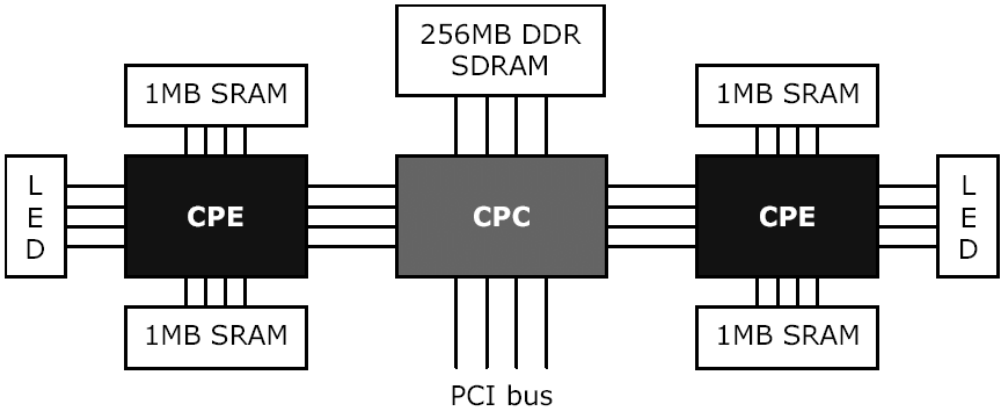

Figure 3. Tarari CPP Architecture.

FPGA configuration is one of the most critical processing components that must be handled with care to ensure an RC speedup over GPP. As mentioned in Chapter 1, configuring RC resources is pure overhead in any RC system and thus has the potential to overshadow RC-performance gains. Reusing RC hardware during a process's execution, known as Run-Time Reconfiguration (RTR), has substantial performance gains over static configuration. One such example of this performance gain is demonstrated by RRANN at BYU. RRANN implements a backpropagation training algorithm using three time-exclusive FPGA configurations. RRANN demonstrated that RTR was able to increase the functional density by 500% compared to FPGA-based implementation not using RTR [18].

The dynamic allocation of RC resources results in multiple configurations per FPGA and consequently yields additional overhead compared to static systems. Dynamic allocation of configurations on a distributed system requires the RC system to maintain a dynamic list of where configuration files reside in the system. Furthermore, RTR systems must handle coordination between configurations, allowing the system to progress from one configuration to the next as quickly as possible.

Moreover, methods such as configuration compression, transformation, defragmentation and caching can further reduce configuration overhead [9]. For example, using configuration compression technique, presented in [19], results in a savings of 11-41% in memory usage. The use of transformation and defragmentation has been shown to greatly reduce the configuration overhead encountered in RC, by a factor of 11 [20]. Configuration caching, in which configurations are retained on the chip or on the board until they are required again, also significantly reduces the reconfiguration

overhead [21]. A well-designed RC system should be able to handle these overhead-reduction methods efficiently.

Some RC systems implement a Configuration Manager (CM) to handle the issues that arise from RTR and configuration overhead reduction. There have been a few noteworthy designs upon which CARMA's configuration manager builds, two of which are discussed in detail: RAGE from the University of Glasgow [22] and a reconfiguration manager from Imperial College, UK [23].

The RAGE run-time reconfiguration system was developed in response to management methods that cater to one application and to one hardware setup [23]. The RAGE system provides a high-level interface for applications to perform complex reconfiguration and circuit-manipulation tasks. Figure 4 shows the dataflow of the RAGE system. A Virtual Hardware Manager (VHM) orchestrates the system by accepting application descriptions. The VHM requests circuit transforms from the Transform Manager if configurations do not currently fit in the FPGA. The VHM also manages the circuit store by converting hardware structures submitted by applications into circuits. The configuration manager loads circuits onto devices, in addition to passing state information and interrupts to the VHM. The device driver handles the board-specific functions and hides the programming interface of the FPGA from higher levels. The functionalities handled by the device driver include writing and reading to and from the FPGA, setting clock frequencies, and even monitoring the FPGA.

Imperial College developed its configuration manager to exploit compile-time information, yet remain flexible enough to be deployed in hardware or software on both partial and non-partial reconfigurable FPGAs. The reconfiguration manager framework

from Imperial College is shown in Figure 5. This framework is composed of three main components: the Monitor, Loader, and Configuration Store. When applications on the system advance to the next configuration, they notify the monitor. The monitor maintains the current state of the system including which FPGAs are in use and with what configurations. In some applications the state of the system can be determined at compile time, thereby reducing the complexity of the monitor. The loader, upon receiving a request from the monitor, loads the chosen configuration onto the FPGA using board-specific API functions. The loader retrieves the needed configuration file from the configuration store. The configuration store contains a directory of configuration files available to the system, in addition to the configuration data itself. A transform agent could be employed to compose configuration at run-time that fit appropriately into the FPGA.

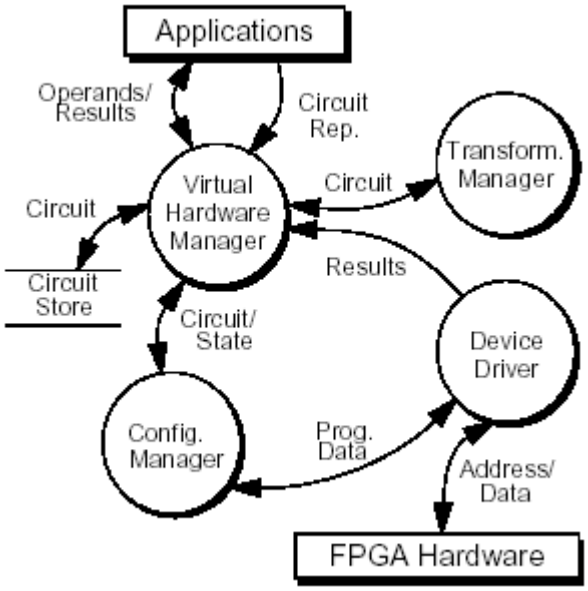

Figure 4. RAGE System Data Flow [22].

Imperial College developed its configuration manager to exploit compile-time information, yet remain flexible enough to be deployed in hardware or software on both

partial and non-partial reconfigurable FPGAs. The reconfiguration manager framework from Imperial College is shown in Figure 5. This framework is composed of three main components: the Monitor, Loader, and Configuration Store. When applications on the system advance to the next configuration, they notify the monitor. The monitor maintains the current state of the system including which FPGAs are in use and with what configurations. In some applications the state of the system can be determined at compile time, thereby reducing the complexity of the monitor. The loader, upon receiving a request from the monitor, loads the chosen configuration onto the FPGA using board-specific API functions. The loader retrieves the needed configuration file from the configuration store. The configuration store contains a directory of configuration files available to the system, in addition to the configuration data itself. A transform agent could be employed to compose configuration at run-time that fit appropriately into the FPGA.

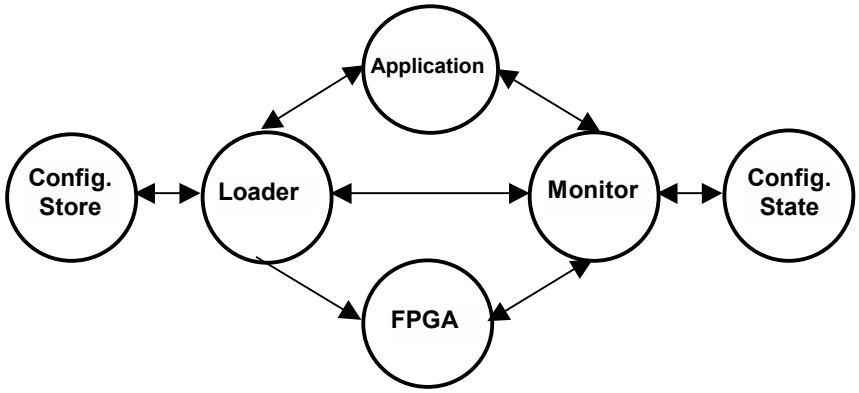

Figure 5. Imperial College Framework [23].

The recent trend toward COTS-based distributed RC clusters requires the deployment of a distributed configuration manager such as the CARMA configuration manager located in the *RC Cluster Management* box in Figure 1. CARMA's execution

manager is analogous to the VHM in RAGE, since both components coordinate the execution of configuration commands. CARMA's CM extends the configuration store of [23] by maintaining a distributed configuration store. The distributed store requires new methods for transporting and accounting configuration files. Furthermore, transformation of configuration files presented in both [22] and [23] can be implemented in the CARMA configuration manager. CARMA's configuration manager employs the high degree of device independence of the RAGE as well as the functional capability of [23]'s Loader. Furthermore, CARMA's configuration manager supports multiple boards, some of which include those previously presented in this chapter, with the Board Interface Module (BIM). The BIM is functionally similar to the device driver of RAGE in that it handles low-level details of board configuration and communication. CARMA's configuration manager extends the Monitor in Shiraz et al. [23] to bring robust, scalable, and highly responsive monitoring down into the FPGAs resources by the use of Gossip-Enabled Monitoring Service (GEMS) [24–29] developed at Florida. A more detailed description of CARMA's configuration manager is given in Chapter 3.

CHAPTER 3
CONFIGURATION MANAGER DESIGN

CARMA's configuration manager incorporates a modular-design philosophy from both the RAGE [22] and the reconfiguration manager from Imperial College [23]. CARMA's configuration manager separates the typical CM operations of configuration determination, management, RC-hardware independence, and communication into separate modules for fault tolerance and pipelining. CARMA establishes a task-based flow of RC-job execution. Consequently, the CARMA's configuration manager encompasses different operating layers, which carry out sub-tasks to complete configuration of the RC Hardware. The CARMA configuration manager supports and improves on current levels of board independence and heterogeneity. In addition, CARMA's configuration manager institutes distributed configuration management to increase scalability, which results in the emergence of multiple management and communication schemes. A description of each of these features follows.

Configuration Management Modules and Interfaces

Figure 6 shows an overview of CARMA's configuration manager with its modules interconnected within a node and between nodes. All modules have been developed as separate processes, rather than inter-related threads, in order to increase the fault tolerance of the system. The execution manager handles the configuration determination while the configuration manager module handles the management of configuration files. The Board Interface Module (BIM) implements board independence to the application and to higher layers. A communication module handles all inter-node communication,

including both the control network and the configuration-file transfer network. The communication module is interchangeable and can be tailored for specific System-Area Networks (SANs).

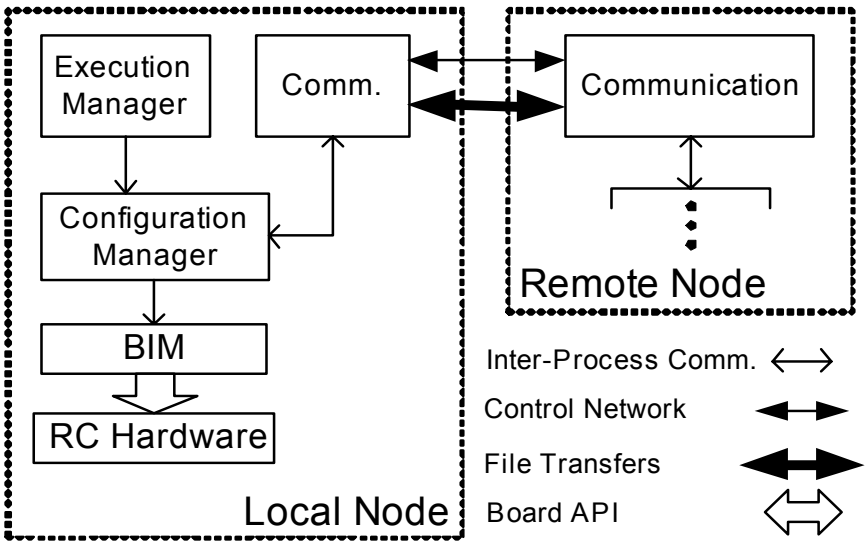

Figure 6. CARMA's Configuration Manager Overview. The figure shows Functional Modules and inter-node and intra-node Communication.

Although the control and file transfer communication can reside on the same network, the current implementation leverages SAN interconnects for large file transfers. TCP sockets (e.g. over IP over Gigabit Ethernet) comprise the control network, while SCI currently serves as the data network for configuration file transfers. Modules within a node use a form of inter-process communication (i.e. message queues) to pass requests and status.

Configuration Manager Initialization and Configuration Determination

At initialization, the CM creates a directory of available RC boards and BIMs are forked off for each board to provide access. After the RC boards have been initialization, the configuration file-caching array is initialized. Next, the CM attempts to retrieve network information. Due to its distributed nature, the CM requires the network

information of other CMs in order to communicate. The CM creates a network object from a file, which contains network information such as the IP address and SCI ID of nodes. Finally, the CM waits for transactions from the execution manager.

Configuration determination is completed once the execution manager receives a task that requires RC hardware. A configuration transaction request is then sent to the CM. From the execution manager's point-of-view, it must provide the CM with information regarding the configuration file associated with the task it is preparing to execute. The CM loads the configuration file on the target RC hardware in what is called a configuration transaction. Although the configuration transaction is the primary service of the CM, the CM also performs release transactions. The execution manager invokes release transactions when tasks have completed and the RC hardware can be released. Releasing the RC hardware allows it to be configured for another task, however the previous configuration is not erased in an attempt to take advantage of temporal locality of configuration use.

Configuration Manager's Operating Layers

A functional description of how CARMA manages configuration files and executes configuration transactions is given in Figure 7. As described before, the CM receives configuration requests from the execution manager. Upon receiving a request, the File-Location layer attempts to locate the configuration file in a manner depending on the management scheme used. A more detailed description of CARMA's distributed management schemes is provided later in this chapter. The File-Transport layer packages the configuration file and transfers it over the data network. The File-Managing layer is responsible for managing FPGA resource access and defragmentation [9], as well as configuration caching [21], relocation and transformation [30]. Furthermore, the File-

Managing layer provides configuration information to the monitor (not shown) for scheduling, distributing and debugging purposes. The File-Loading layer uses a library of board-specific functions to configure and control the RC board(s) in the system and provide hardware independence to higher layers.

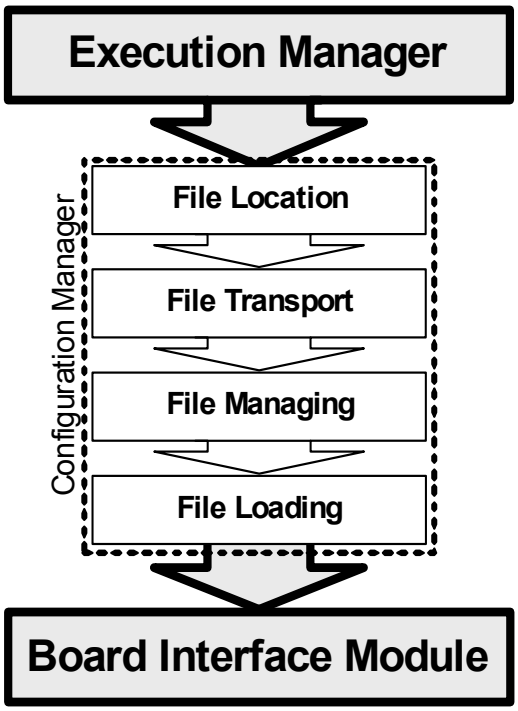

Figure 7. Configuration Manager's Layered Design. This Figure shows the Layers inside the CM block, which implement the Location and Management of Configuration Files in CARMA.

The transformation of configuration files and the CM's distributed nature requires a configuration store that is dynamic in both content and location. File location begins by searching the node's local configuration-file cache. If there is a miss, a query is sent to a remote CM. Locating a configuration file varies in complexity depending on the management scheme, since in some schemes there is a global view of the RC system, while in others there is not.

Due to CARMA's initial deployment on cluster-based systems, the CM typically has access to SANs which are widely deployed in clusters. SANs, such as SCI [31] and Myrinet [32], provide high-speed communication ideally suited for latency-dependent service such as configuration-file transport. To further diminish the transportation latency, the CM can exploit collective-communication mechanisms such as multicast supported by SANs [33, 34].

The CM's file-managing layer would deal with configuration-file caching and transformation, sometimes-called relocation, of configuration files in addition to defragmentation of the FPGA. Caching of configuration files is implemented by storing recently used configuration files in memory located on the RC board. For RC boards that do not support on-board configuration file storage, the files can be stored in RAM disk. CARMA's configuration manager currently does not support relocation as described in [9] and shown in Figure 8a, because current COTS-based RC-board software does not support partial reconfiguration. Defragmentation, shown in Figure 8b, is also not supported in the current CARMA version due to the inability to partially configure the RC boards.

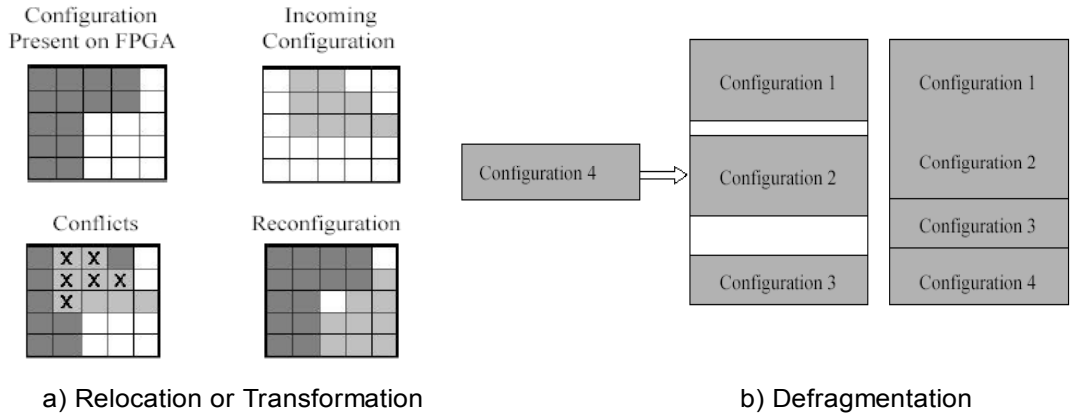

Figure 8. Illustration of Relocation (Transformation) and Defragmentation [9].

Board Interface Module

A key feature of CARMA's configuration manager is that it provides board independence to higher layers. Board independence has not effectively been implemented in today's RC run-time management tools. CARMA's file-loading layer achieves this board independence with the creation of a BIM for each board. The BIM provides both the application and the CM's higher layers a module that translates generic commands into board-specific instructions. Each board type supported by CARMA's CM has a specific BIM tailored using that board's API.

Figure 9 depicts the communication between the application and RC hardware through the BIM. At initialization, the CM spawns off a BIM for each of the boards within the node. The BIM remains dormant until the application requires use of the board, at which time the CM uses the BIM to configure the board. The application then sends data destined to the RC hardware to the BIM. The BIM then forwards the information in an appropriate format, and using the board-specific API, passes it to the board. After the application is finished accessing the board, the BIM goes back to its dormant state.

Although the primary feature the BIM provides is board independence, the BIM also yields other advantageous features. As described the BIM provides access to the RC board for a local application, however the BIM also allows seamless and secure access to the RC board from remote nodes. Furthermore, the use of the BIM increases the reliability of the system, since applications do not access the boards directly. A security checkpoint could be established inside the BIM to screen data and configuration targeted to the RC board. However, these additional features do come at a slight overhead cost

(roughly 10 μs), decreased control of the board by the application, and furthermore, minor code additions to the application.

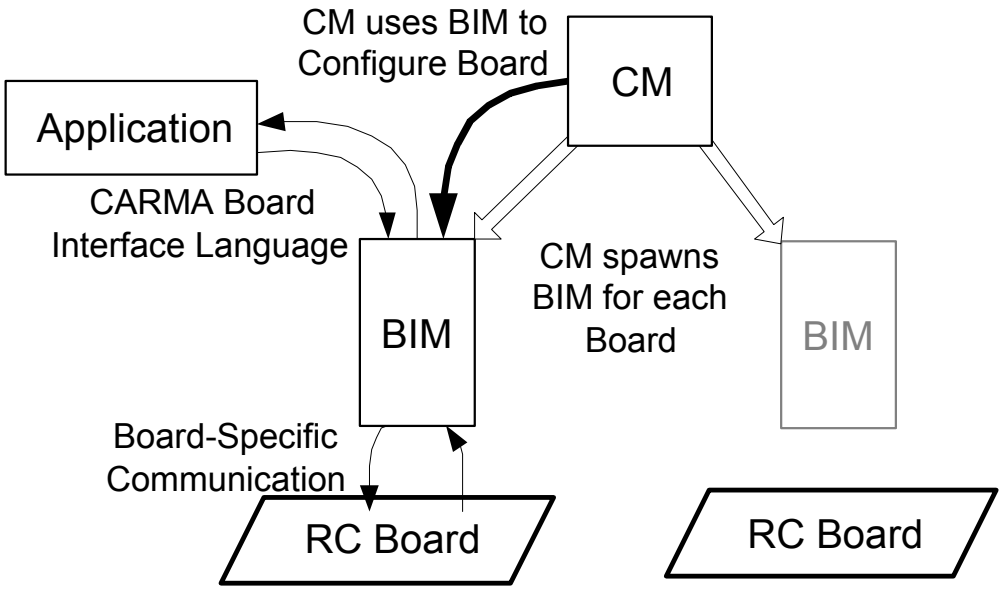

Figure 9. Board Interface Modules.

Distributed Configuration Manager Schemes

In order to provide a scalable, fault-tolerant configuration management service for thousands of nodes (one day) the CARMA configuration manager is fully distributed. The CARMA service is targeted for systems of 64 to 2,000 nodes and above, such as Sandia's Cplant [35] and Japan's Earth Simulator [36]. Such large-scale systems and grids will likely include RC hardware one day. In creating the distributed CM model, four distributed management schemes are proposed: Master-Worker (MW), Client-Server (CS), "Pure" Peer-to-Peer (PPP), and "Hybrid" Peer-to-Peer (HPP). Figure 10 illustrates these four schemes. While CARMA configuration management modules exist in different forms on various nodes in the four schemes, in all cases the CMs still use the communication module to communicate with one another.

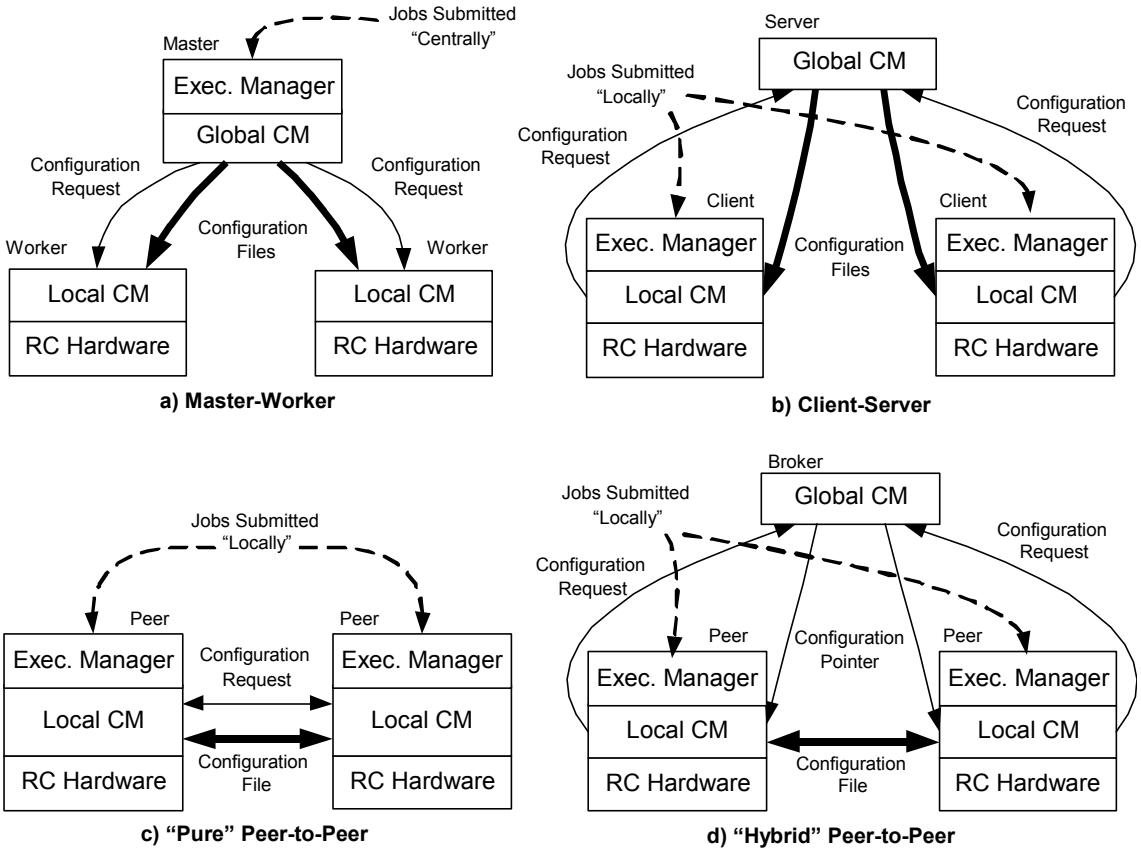

Figure 10. Distributed Configuration Management Schemes.

The MW scheme (Figure 10a) is a centralized scheme where the master maintains a global view of the system and has full control over job scheduling and configuration management. This scheme is representative of currently proposed CMs discussed in Chapter 2. While a centralized scheme is easy to implement, there will be performance limitations due to poor scalability for systems with a large number of nodes. The other three schemes in Figure 10 assume a distributed job-scheduling service. For the CS scheme, (Figure 10b) local CMs request and receive configurations from a server. Although this scheme is likely to exhibit better performance than MW for a given number of nodes, there will also be scalability limitations as the number of nodes is increased.

The PPP scheme (Figure 10c) contains fully distributed CMs where there is no central view of the system. This scheme is similar to the Gnutella file-sharing network [37] and is described academically by Schollmeier at TUM in Munich, Germany [38]. This scheme will likely provide better latency performance since hot spots have been removed from the system. However, the bandwidth consumed by this scheme would likely be unwieldy when the number of nodes is rather large. The HPP scheme (Figure 10d) attempts to decrease the bandwidth consumed by PPP by consolidating configuration-file location information in a centralized source. The HPP scheme resembles the Napster file-sharing network [39]. The HPP scheme is also a low-overhead version of CS because local CMs receive a pointer from the broker to the client-node that possesses the requested configuration. This scheme will likely further reduce the server bottleneck by reducing service time. Having multiple servers/brokers may further ease the inherit limitations in these schemes.

A hierarchical-layered combination of these four schemes will likely be needed to provide scalability up to thousands of nodes. For example, nodes might be first grouped using CS or HPP and then these groups might be grouped using PPP. The layered group concept has been found to dramatically improve scalability of HPC services for thousands of nodes [24]. Hierarchical layering and variation of these schemes are presented with analytical-scalability projections in Chapter 6. The following chapter details the experimental setup used to evaluate and experimentally compare the MW, CS and PPP schemes up to 16 nodes; HPP has been reserved for future study.

CHAPTER 4
EXPERIMENTAL SETUP

To investigate the performance of the CM in the MW, CS, and PPP schemes, a series of experiments are conducted. The objectives of these performance experiments are to determine the overhead imposed on the system by the CM, determine the components of latency in configuration transactions, and to provide a quantitative comparison between MW, CS, and PPP schemes. The performance metric used to compare these schemes is defined as the completion latency from the time a configuration request is received from the execution manager until the time that configuration is loaded onto the FPGA.

Configuration Manager Protocols

Figures 11, 12, and 13 illustrate the individual actions that compose a configuration transaction indicating the points at which latency has been measured for MW, CS, and PPP, respectively. The experimental setup includes system sizes of 2, 4, 8, and 16 nodes. In the MW and CS schemes, one node serves as the master or server while the remaining nodes are worker or client nodes. In the PPP scheme all the nodes are peer nodes. The Trigger block in Figures 11, 12, and 13 acts in place of the CARMA's execution manager and stimulates the system in a controlled periodic manner.

A MW configuration transaction is composed of the following components as shown in Figure 11. The interval from 1 to 2 is defined as *Creation Time* and is the time it takes to create a configuration-request data structure. The interval from 2 to 3 is defined as *Proxy Queue Time* and is the time the request waits in the proxy queue until it

can be sent over a TCP connection to the worker, while the *Proxy Processing Time* is the time it takes the Proxy to create a TCP connection to the worker and is the interval from 3 to 4. These connections are established and destroyed with each transaction because maintaining numerous connections is not scalable. The interval from 4 to 5 is defined as *Request Transfer Time* and is the time it takes to send a configuration request of 148 bytes over TCP. This delay was observed to average 420ms using TCP/IP over Gigabit Ethernet with a variance of less than 1%.

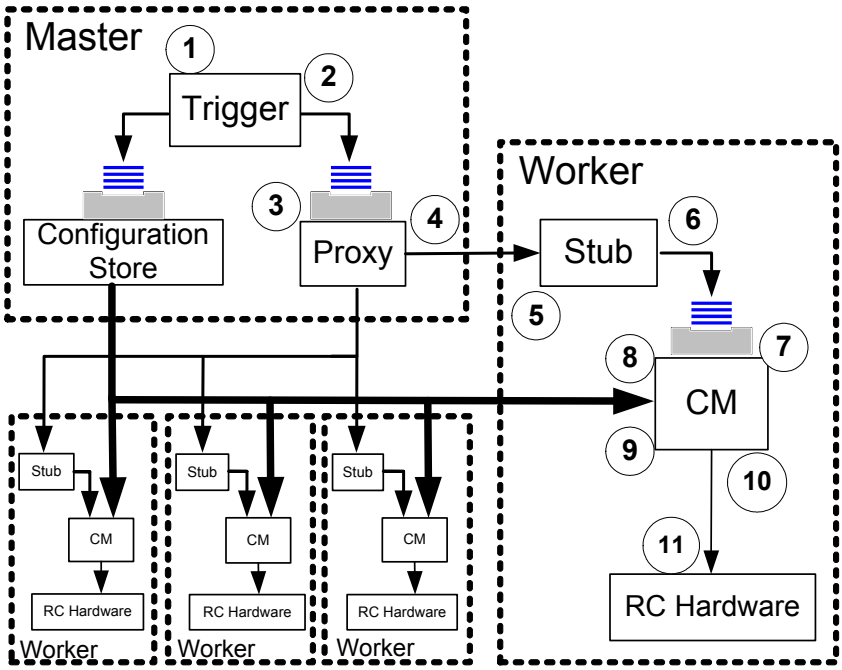

Figure 11. Master-Worker Configuration Manager Scheme.

The interval from 5 to 6 is defined as *Stub Processing Time* and is the time it takes the Stub to read the TCP socket and place it in the CM queue, while the interval from 6 to 7 is defined as the *CM Queue Time* and is the time the request waits in the CM queue until the CM removes it. The *CM Processing Time* is the time required to accept the configuration request and determine how to obtain the needed configuration file and is the interval from 7 to 8. The interval from 8 to 9 is defined as *File Retrieval Time*, the

time it takes to acquire the configuration file over SCI, including connection setup and tear down, whereas the interval from 9 to 10 is defined as *CM-HW Processing Time* and is the time to create a request with the configuration file and send the request to the BIM. Finally, the interval from 10 to 11 is defined as *HW Configuration Time* and is the time it takes to configure the FPGA.

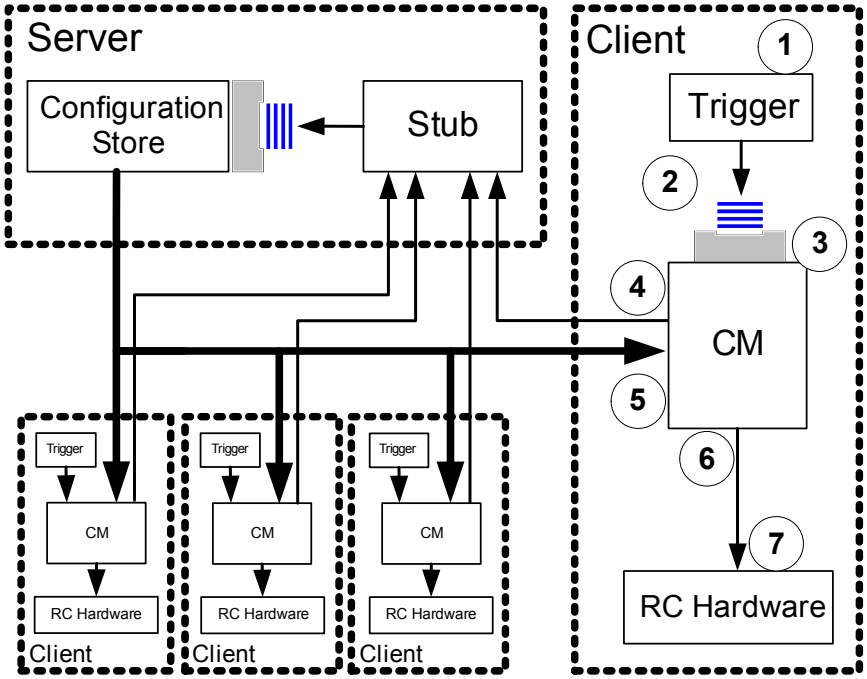

Figure 12. Client-Server Configuration Manager Scheme.

CS configuration transactions are composed of the following components as shown in Figure 12. Note that jobs (via the Trigger block) are created on the client nodes in the CS scheme rather than on the master in the centralized MW scheme. The intervals from 1 to 2, 2 to 3 and 3 to 4 are the *Creation Time*, *CM Queue Time* and *CM Processing Time*, respectively and are defined as in MW. The interval from 4 to 5 is defined as *File Retrieval Time*, which is the time required for a client to send a request to the server and receive the file in response. The intervals from 5 to 6 and 6 to 7 are the *CM-HW Processing Time* and *HW Configuration Time*, respectively and are defined as in MW.

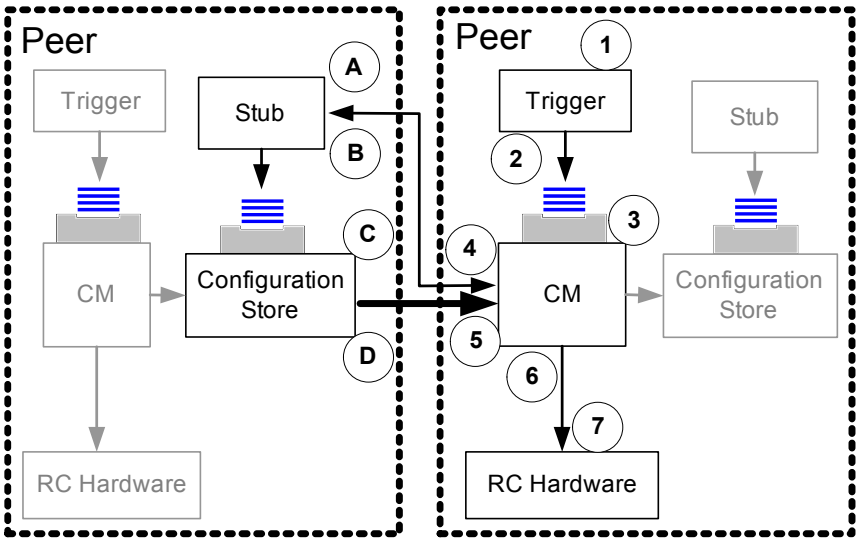

Figure 13. "Pure" Peer-to-Peer Configuration Manager Scheme.

PPP configuration transactions are composed of the following components as shown in Figure 13. Note that jobs (via the Trigger block) are created on the requesting peer nodes in the same way as the CS scheme. The intervals from 1 to 2, 2 to 3 and 3 to 4 are the *Creation Time*, *CM Queue Time* and *CM Processing Time*, respectively and are defined as they are in the two previous schemes. On the requesting peer, the interval from 4 to 5 is defined as *File Retrieval Time*, which is the time required for one peer to send a request to another peer and receive the file in response. The *File Retrieval Time* includes the time it takes for the requesting peer to contact each peer individually over TCP to request the needed configuration file. In order to explore the best-case and worst-case scenarios of configuration file distribution, the experiment was performed in two forms. The worst-case scenario was one in which the requesting node must contact all other peers for the needed configuration file and the last peer is the one that has the configuration file. In the best-case PPP, the first peer the requesting node contacts contains the configuration file. The intervals from 5 to 6 and 6 to 7 are the *CM-HW*

Processing Time and *HW Configuration Time*, respectively and are defined as in the two previous schemes.

On the responding side of the peer, the interval from A to B is defined as *Query Response Time*, which is the time required for a peer to determine if it has the requested configuration file. The stub sends a message to the configuration store to send the file to the requesting node, in addition to responding back to the requesting node that the configuration file was found. The interval from B to C is defined as the *Configuration Store Queue Time*, and is the time the request waits in the configuration store queue until the configuration store can service it. The configuration store establishes an SCI connection to the requesting peer and sends the configuration file, and is referred to as *Configuration Store Processing Time*, and is defined as the interval between C and D.

Experimental Setup

Experiments are performed with one instance of the CM running on a node, with the host system consisting of a Linux server with dual 2.4GHz Xeon processors and 1GB of DDR RAM. The control network between nodes is switched Gigabit Ethernet. The data network is 5.3 Gb/s SCI connected in a 2D torus with Dolphin D337 cards using the SISCI-software release 2.1. Each computing node contains a Tarari HPC board (i.e. CPX2100) [17] housed in a 66MHz, 64-bit PCI bus slot.

For this study, the number of nodes is varied from 2, 4, 8, and 16 nodes. The 2D topology of the SCI network is shown in Figure 14. In experiments with the schemes of MW and CS, the CM is executed with one node serving as master/server and the others as computing nodes. In the 2, 4, 8, and 16-node cases the master/server always resides in the same node.

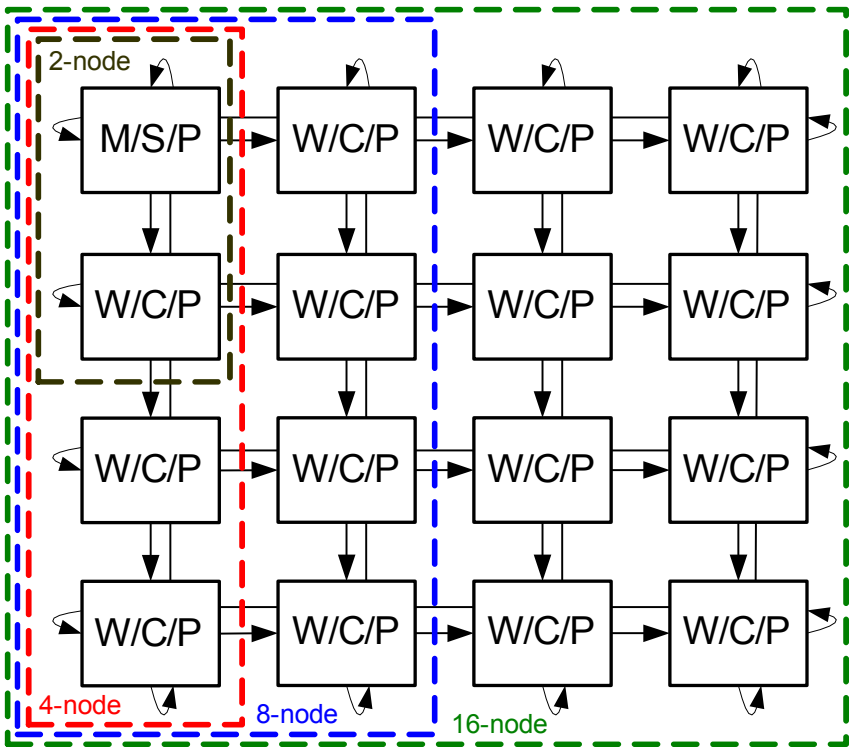

Figure 14. Experimental Setup of SCI nodes. Nodes are labeled Master (M), Servers (S), Workers (W), Clients (C), and Peers (P).

The interval between requests, known as Configuration Request Interval, is chosen as the independent variable and is varied from 1s to 80ms. The value of 80ms is selected as the lower bound because this value is determined to be the minimum-delay time to retrieve a Virtex-II 1000 configuration file (~20ms) and configure a Tarari board (~60ms). The configuration of the board holds constant throughout the experiments, and is a function of the board's API, independent of CARMA's CM. In each trial, 20 requests were submitted with constant periodicity by the trigger, each requesting a different configuration file. Thus the trigger mimics the execution manager's ability to handle configuration requests for multiple tasks that are running in the system. Since the execution manager is serialized, the trigger only sends one request at a time and is designed to do so in a periodic manner. This setup mimics the multitasking ability of the

execution manager for 20 tasks running on the node each requiring a different configuration file. The timestamps for measuring all intervals is taken using the C function *gettimeofday()*, which provides a 1μs resolution. Each experiment for each of the 3 schemes was preformed at least three times, and the values have been found to have a variance of at most 11%. The following chapter presents the final run of the experiment to be representative results of the experiment.

CHAPTER 5
EXPERIMENTAL RESULTS

Each of the completion-latency components are measured across all computing nodes in the MW, CS, and PPP schemes, and are summarized along with maximum, minimum, and average values in Figures 15, 16, and 17, respectively. All schemes have four computing nodes and the master/server are located in an adjacent SCI ring. PPP is performed in the worst-case scenario, as defined in Chapter 4. The results for request intervals larger than 250 ms are not shown because these values are found to be virtually identical to the 250 ms trial, for all schemes. Values in this region are found to be relatively flat because the master/server/peer is able to service requests without imposing additional delays. In this stable region, requests are only delayed by fixed values that are the sum of *File Retrieval Time* and *HW Configuration Time*, which is approximately equal to 130 ms, 140 ms, and 190 ms for MW, CS, and PPP, respectively.

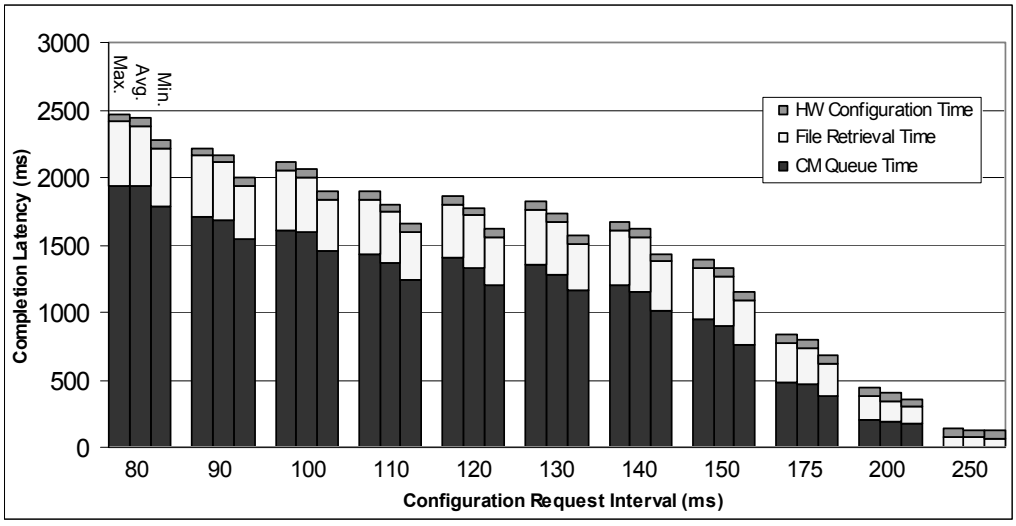

Figure 15. Completion Latency for Four Workers in MW.

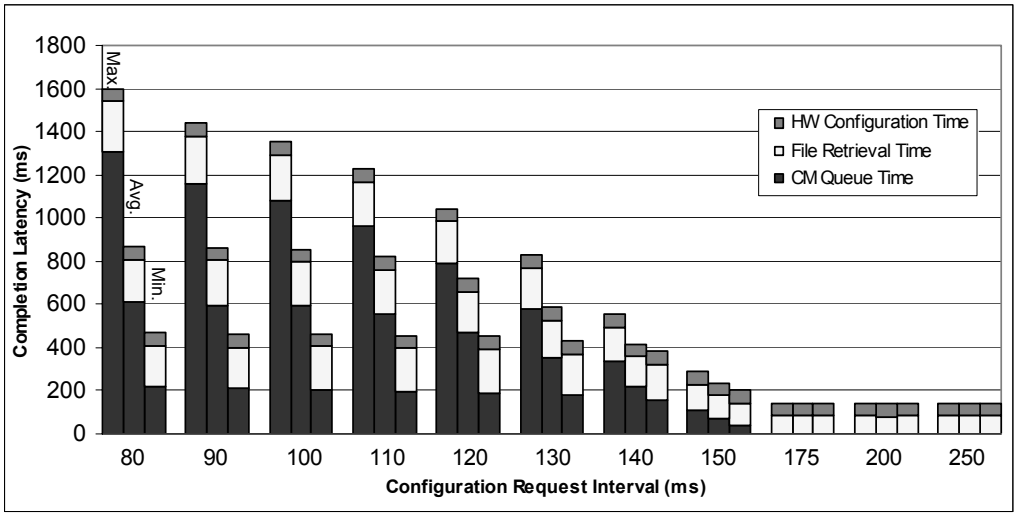

Figure 16. Completion Latency for Four Clients in CS.

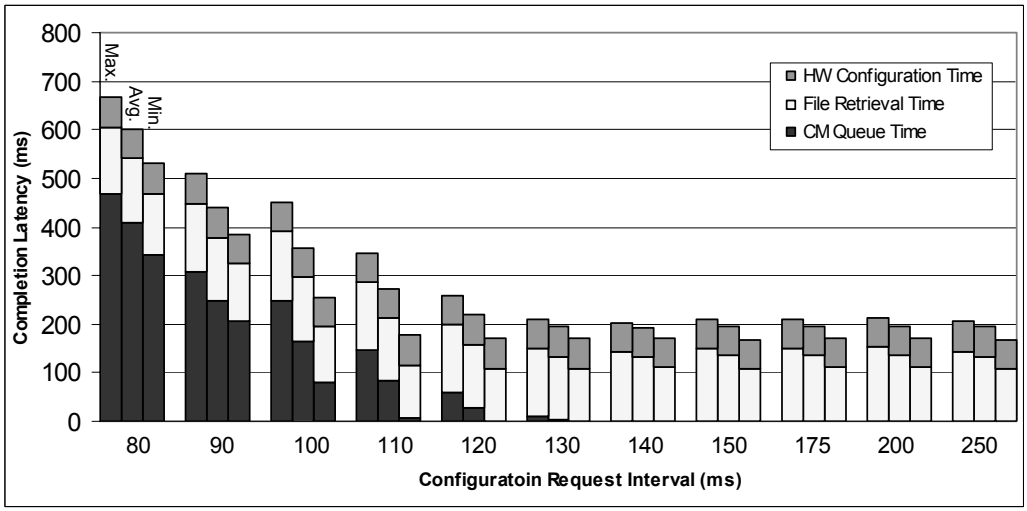

Figure 17. Completion Latency for Four Peers in PPP.

Several observations can be made from the values in Figures 15, 16, and 17. In MW, CS, and PPP, the major components of completion latency are *CM Queue Time*, *File Retrieval Time* and *HW Configuration Time*, the largest of these naturally being *CM Queue Time* as resources become overloaded (i.e. decreasing configuration request interval). The remaining components combined are in the worst case less than 2% of the total completion latency, and thus are not shown in Figures 15, 16, and 17. The data

demonstrates that the CM design in CARMA imposes virtually no overhead on the system in all cases. Furthermore, the data shows that when the request interval is large, *File Retrieval Time* and *HW Configuration Time* are the dominant factors. *CM Queue Time* and *File Retrieval Time* are the most dominant components of completion latency when the request interval is small. In Figure 15, *CM Queue Time* increases rapidly while *File Retrieval Time* grows steadily. In Figure 16, both *CM Queue Time* and *File Retrieval Time* grow steadily. However, in the PPP scheme, as seen in Figure 17, *File Retrieval Time* holds steady and *CM Queue Time* increases only after the configuration request interval is less than the *File Retrieval Time*. This phenomenon is a result of how the schemes are designed. Since PPP is designed with no single point of contention, the only factor in its queuing delay is CM-processing time dominated by *File Retrieval Time*. When the arrival rate is greater than the processing time, the *CM Queue Time* increases steadily. In the MW and CS schemes the *File Retrieval Time* is dependent on a centralized master or server, respectively. Another useful observation shown in Figures 15, 16 and 17 is the spread between the maximum and minimum values. In MW and PPP the spread is relatively small, at worst a 14% deviation from the average for MW and at worst 23% deviation for PPP. CS on the other hand experiences a worst-case 98% deviation from the average. The following figures investigate this phenomenon between the MW and CS schemes.

Figure 18 and 19 present a per-node view of MW and CS respectively, which shows how the two schemes differ significantly. In CS, there is a greater variability in the completion latency each node experiences due to server access. Nodes closer to the server on the SCI-torus experience smaller file-transfer latencies, which allow them to

receive and therefore request the next configuration file faster. Completion latencies in the MW scheme have less variability between nodes because all requests are serialized in the Master. Therefore, no one node receives an advantage based on proximity.

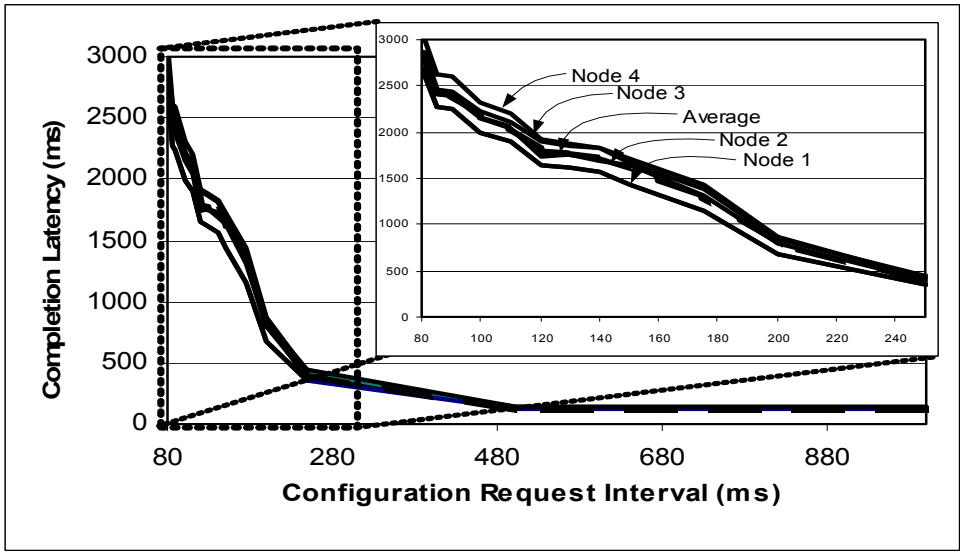

Figure 18. Completion Latency of Four Workers with Master in Adjacent SCI Ring.

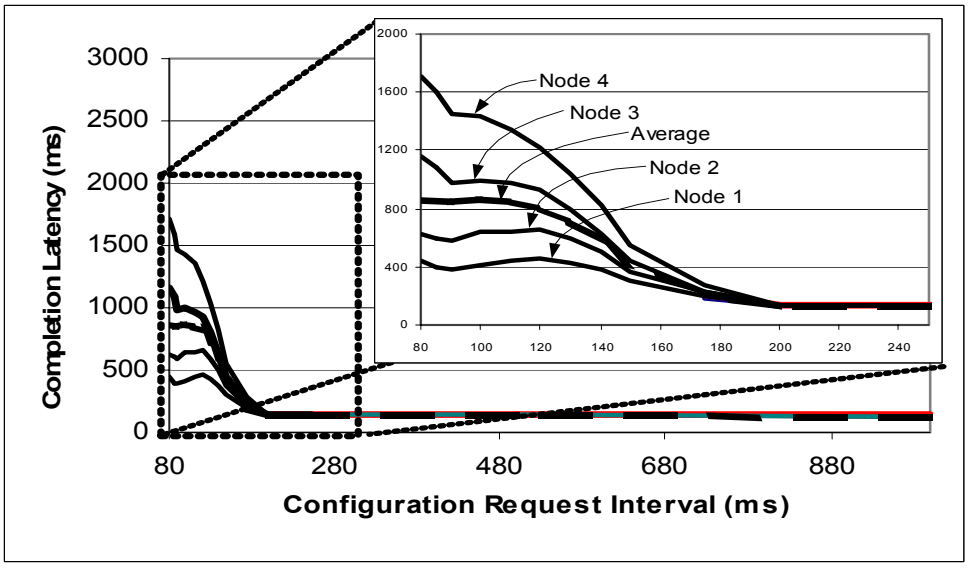

Figure 19. Completion Latency of Four Clients with Server in Adjacent SCI Ring.

Figure 20 illustrates the experiment where the server is moved into the same SCI ring as 3 clients. In the experimental results shown in Figure 19, the server resides on the

adjacent SCI ring requiring a costly dimension switch to transfer files to clients. The data shows that the client closest to the server experienced a *File Retrieval Time* lower than that of other nodes at the stressed configuration-request interval of 100 ms. However, in the experiment shown in Figure 20 all the clients experienced similar *File Retrieval Time* at the stressed configuration-request interval of 100 ms.

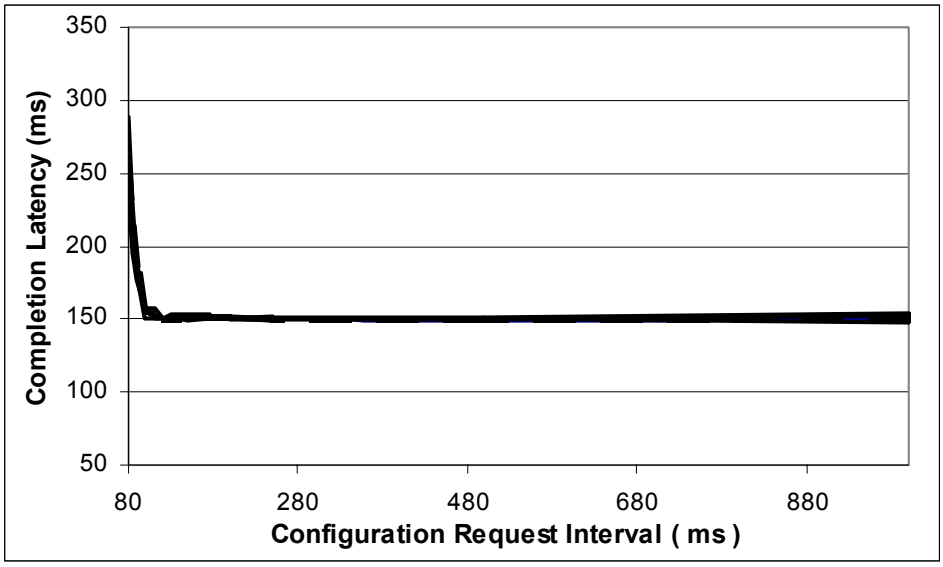

Figure 20. Completion Latency of Three Clients with Server in the Same SCI Ring.

This dependency on the client-to-server proximity is further illustrated in Figure 21, which shows the CS scheme for 8-nodes. Figure 21 shows results from the 8-node case where 3 clients reside on the same SCI ring as the server and 4 clients are on the adjacent ring. The spread between nodes is approximately 500 ms at a configuration-request interval of 80 ms.

Figures 22, 23, and 24 present the scalability of MW, CS, and PPP (worst-case), respectively. Figure 22 shows that MW is not scalable since completion latency dramatically increases as the number of nodes increases, on average triples from 2 to 4 nodes and 4 to 8 nodes.

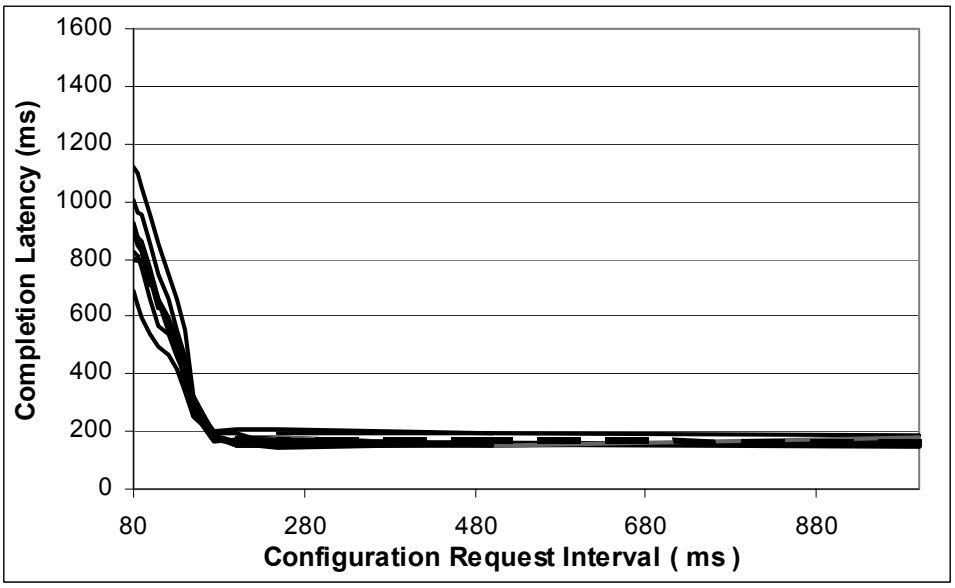

Figure 21. Completion Latency of Eight Clients in CS.

Particularly, the 16-node MW trial incurs a queuing delay when the configuration-request interval is 1 s, which no other scheme incurs. These results clearly show that traditional centralized-managed schemes developed to date cannot be used for even relatively small distributed systems. Figure 23 shows that CS has a low completion latency for small numbers of nodes (i.e. 2 and 4 nodes), however rapidly increases for larger numbers of nodes (i.e. 8 and 16). For all system sizes, the completion latency of CS was roughly only 10% of MW. Figure 24 reveals that worst-case PPP has a constant increase in completion latency over 2, 4, 8, and 16 nodes. PPP has higher completion latency for small numbers of nodes (i.e. 2 and 4 nodes) being roughly 1.5 times that of CS. However, for larger numbers of nodes (i.e. 16) PPP has a lower average completion latency than CS, measuring close to 500 ms lower at configuration-request intervals less than 200 ms.

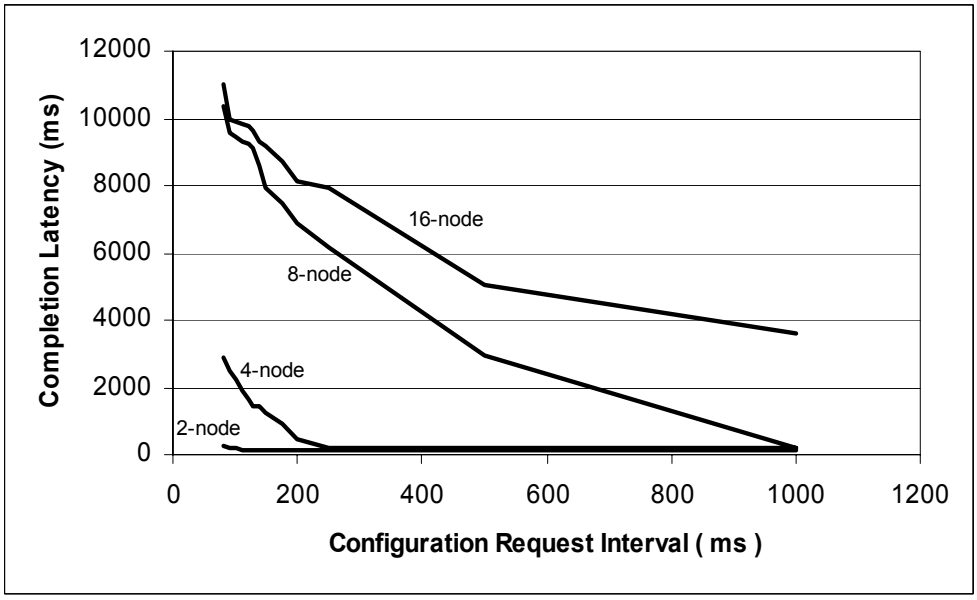

Figure 22. Completion Latency of MW Scheme for 2, 4, 8, and 16 nodes.

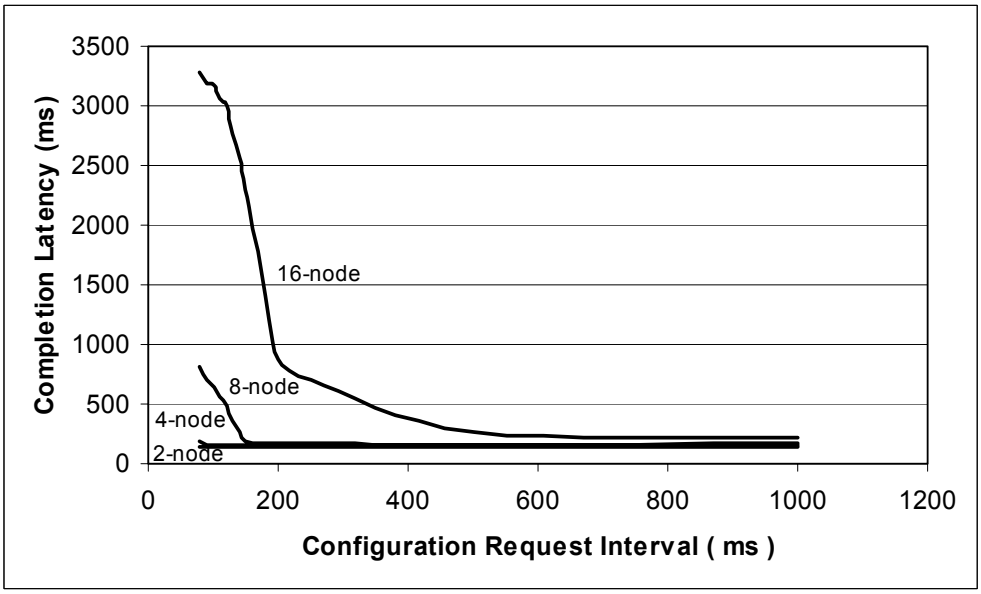

Figure 23. Completion Latency of CS Scheme for 2, 4, 8, and 16 nodes.

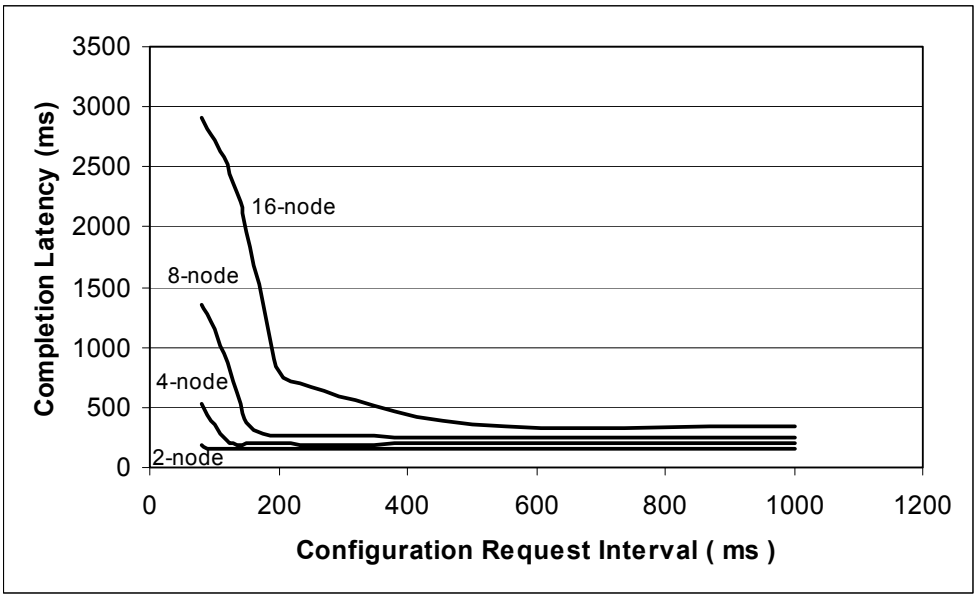

Figure 24. Completion Latency of Worst-Case PPP Scheme for 2, 4, 8, and 16 nodes.

Figures 22-24 show that "Pure" Peer-to-Peer and CS provide better scalability than MW. However, larger testbeds are needed to investigate scalability of these schemes at larger system sizes. As described earlier in this chapter, PPP has no single point of contention or hot spot, thus *File Retrieval Time* is relatively constant regardless of configuration-request interval. Furthermore having a fully distributed CM service also provides a great deal in terms of fault tolerance. The next chapter will provide an analytical scalability study of MW, CS, and PPP based on the experimental data shown above as well as possible extensions to these schemes.

CHAPTER 6
PROJECTED SCALABILITY

The current trend of cluster-based RC systems scaling to larger numbers of nodes will likely continue to progress, as RC technology advances. One day RC computing clusters might rival the size and computing power of Sandia National Laboratories Cplant [35], a large-scale parallel-computing cluster. Coupling parallel-computing power of this level with RC hardware yields a daunting task of system management. CARMA's configuration manager is one solution for allocating and staging configuration files in such large-scale distributed systems, however its performance on such system sizes should be explored. Based on the results presented in the previous chapter, CS and PPP schemes scale the best up to 16 nodes. Taking the experimental results for 2-, 4-, 8-, and 16-node systems, the completion latencies for larger systems can be projected.

Completion Latency Projections

Since searching for configuration files in large-scale RC system has no precedent, using current lookup algorithms available for PPP networks can yield an approximate number of nodes contacted in a query. The Gnutella lookup algorithm is a good starting point because of its simplicity and low overhead. Another lookup algorithm for PPP networks, Yappers, proposed by the CS department at Stanford University [40] uses hash tables, however this would yield weighty query processing. CARMA's PPP scheme would lie in the middle of these algorithms. A Gnutella-style algorithm would contact all nodes, and Yappers using the minimum number of hash buckets, resulting in minimum overhead, would contact roughly 25% of nodes.

For the projections presented in this thesis, a typical-case PPP is assumed to be 25% of worst-case (i.e. all nodes contacted) PPP. A projection to larger node sizes is done with curves of the form $y = m\,x + b$ for PPP and $y = m\,x^2 + b$ for CS. These curves are chosen for each since they are best-fit curves to the experimental data presented in Chapter 5. Both *m* and *b* are calculated mathematically and their values are found to be *m*=187.37 and *b*=138.6 for PPP, and *m*=10.85 and *b*=122.4 for CS. Figure 25 shows the results for CS, worst-case PPP (i.e. all nodes contacted), and typical-case PPP (i.e. 25% of nodes contacted) projected to 4096 nodes, since Dolphin's SCI hardware scales to system sizes of 4096 [41]. The values used for the projection are at a configuration-request interval of 100 ms, since it is at this point (see Figures 16-18) that the CM is adequately stressed.

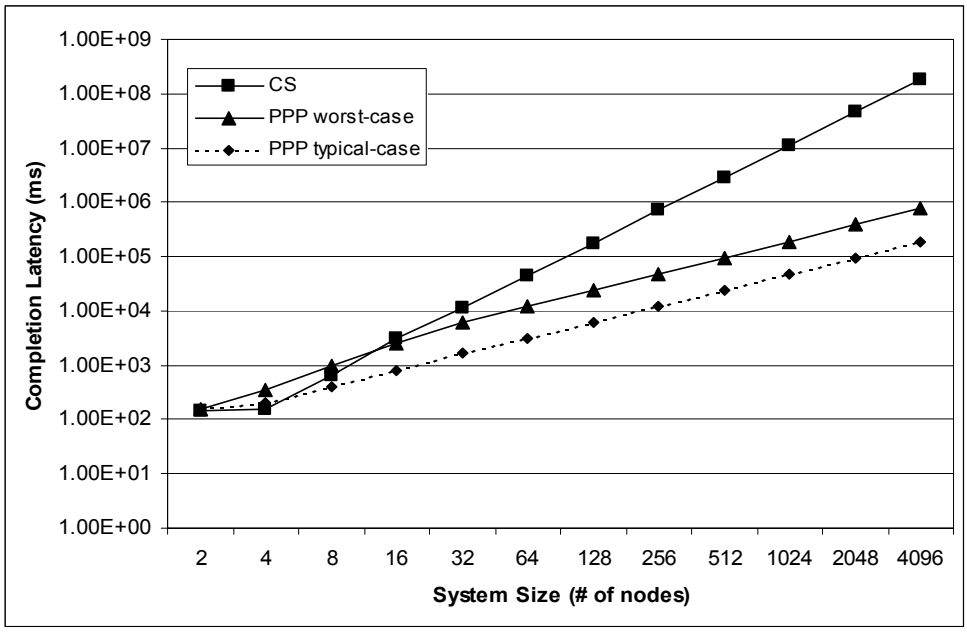

Figure 25. Completion Latency Projections for Worst-Case PPP, Typical-Case PPP, and CS. Note: Logarithmic Scales.

Figure 25 shows that CS has greater completion latencies than worst-case PPP when group sizes are greater than 12 and greater than typical-case PPP when group sizes

are greater than 6. Another observation from Figure 25 is that the schemes do not scale well for system sizes larger than 32. The completion latency of the CS scheme is of the order of 10^8 seconds for a system size of 4096, while a 32-node system yields completion latencies of a more reasonable 2 seconds. The data shows that in order to achieve reasonable completion latencies and scale to larger systems, CM nodes should be grouped into a layered hierarchy.

Hierarchy Configuration Managers

For this analytical study a two-layered hierarchy is chosen, where each layer could be implemented in either CS or typical-case PPP schemes. Typical-case PPP is used to more closely model a real system. Consequently the following four permutations are investigated, PPP-over-CS, CS-over-CS, CS-over-PPP, and PPP-over-PPP, shown in Figure 26.

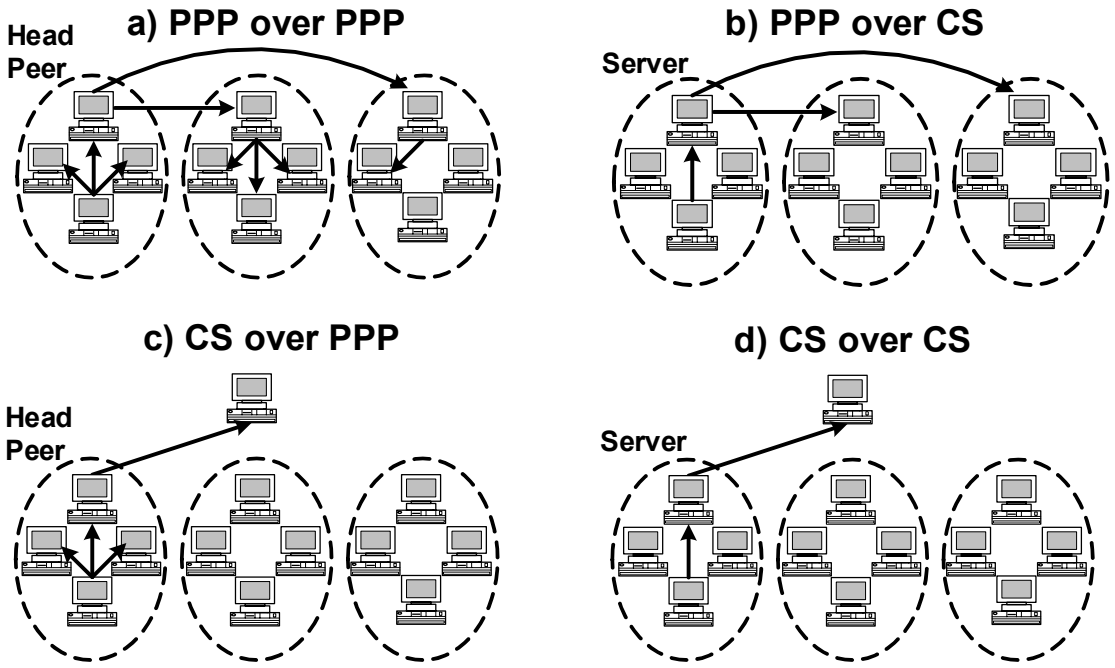

Figure 26. Four Layered Hierarchies Investigated.

The lower layer is divided into groups, where one node in each group is designated the head node. The server is the head node if the lower-layer scheme is CS and if the lower-layer scheme is PPP then one peer is designated as head node. The head node of each group then searches and transfers configuration files in the upper layer. In PPP-over-PPP shown in Figure 26a, a requesting node contacts the nodes in its group via the PPP scheme described. Normally a node can find the requested configuration file within its group, however when it cannot it contacts the head node of its group, which then retrieves the configuration file from the upper layer. This outcome is known as a configuration miss. On a configuration miss, the head node of the requesting group contacts the head nodes of other groups in search of the requested configuration file as dictated by the PPP scheme. When the requesting head node contacts other head nodes, each contacted head node searches for the configuration file within its group using the PPP scheme. In PPP-over-CS shown in Figure 26b, a requesting node contacts the server for its group. When the server does not have the requested configuration file, it contacts other servers using the PPP scheme. Figure 26c shows CS-over-PPP in which the nodes search for configuration files within the groups using PPP. When a configuration file is not found within the group the head node of the group contacts the server in the upper layer. Figure 26d shows CS-over-CS in which nodes contact the group's server, which then contacts the upper-layer server on a configuration miss. Using these hierarchical protocols, projection equations for completion latency can be derived and are shown in Table 1.

Table 1. Completion Latency Projection Equations for System Hierarchies.

Hierarchy	Completion Latency Projection Equations
PPP-over-PPP	$P \times (PPP(g)) + Q \times P \times (PPP(n \div g) + PPP(g))$
PPP-over-CS	$CS(g) + Q \times P \times (PPP(n \div g))$
CS-over-PPP	$P \times (PPP(g)) + Q \times (CS(n \div g))$
CS-over-CS	$CS(g) + Q \times (CS(n \div g))$

The functions *PPP(x)* and *CS(x)* represent the average completion latency experienced by one configuration request in a group of *x* nodes using the PPP and CS schemes, respectively. These functions are projected beyond 16 nodes and were presented earlier in this chapter. The variable *n* represents the number of nodes in the system, while *g* represents the group size. Groupings of 8, 16, 32, 64, and 128 are investigated, since these group sizes are reasonable in system sizes up to 4096 nodes. *P* represents the percentage of nodes contacted in the PPP scheme. Since we are using typical-case PPP, *P* has a value of 25%. *Q* is the configuration-miss rate and is assumed to occur 10% of the time based on measurement of distributed file systems in [42]. Although research into configuration caching is left for future work, the value of 10% is reasonable for this RC system. Using these equations and varying both group size and system size, a matrix of values is calculated. The optimal group size in each hierarchy as the system size increases is determined and shown in Figure 27.

Figure 27 shows PPP-over-CS has a gradual change of optimal group sizes ending with optimal group size of 8 nodes for systems sizes > 512, while CS-over-PPP has a rapid change of optimal group sizes over system sizes ending with optimal group size of 128 nodes for systems sizes > 2048. Another observation is that the upper-layer scheme determines the optimal group size. For instance, a 128-node group size minimizes the

high completion latency of CS in CS-over-PPP scheme, and an 8-node group size maximizes low completion latency of PPP in PPP-over-CS scheme. Figure 28 and 29 show the completion latency of the hierarchies using the optimal group size at the appropriate system sizes.

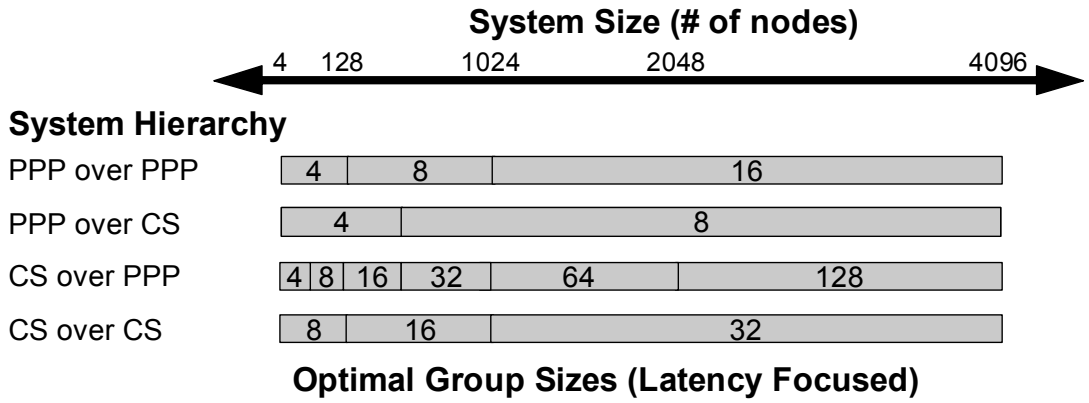

Figure 27. Optimal Group Sizes for each Hierarchy as System Size Increases.

The data in Figure 28 shows that for system sizes of up to 512, PPP-over-CS with groups of 4 has the lowest completion time. Furthermore, for system sizes of 512 to1024 nodes, PPP-over-PPP with groups of 8 has the lowest completion time, and for system sizes of 1024 to 4096, PPP-over-PPP with groups of 16 has the lowest completion time, shown in Figure 29.

Consumed Bandwidth Projections

While the PPP-over-PPP hierarchy is projected to achieve the lowest completion latency at large scale, the control-communication bandwidth between nodes could be weighty. This section presents analytical control-bandwidth calculations for previously presented layered hierarchies. The data-network bandwidth utilization for all the schemes is no worse than 7.7%, and remains constant regardless of system size and therefore is not investigated.

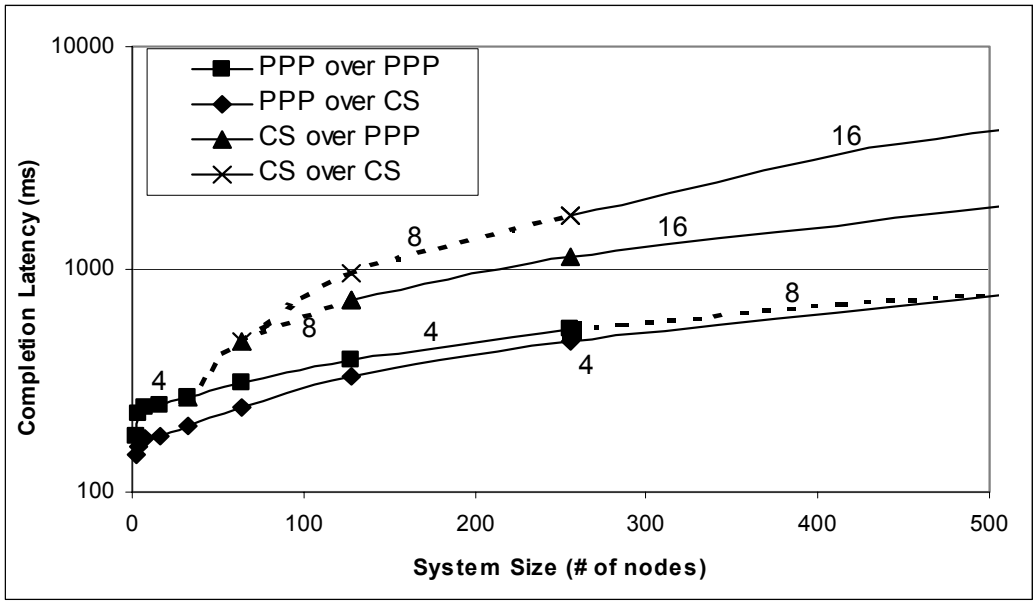

Figure 28. Completion Latency Projections with Optimal Group Sizes up to 500 Nodes.

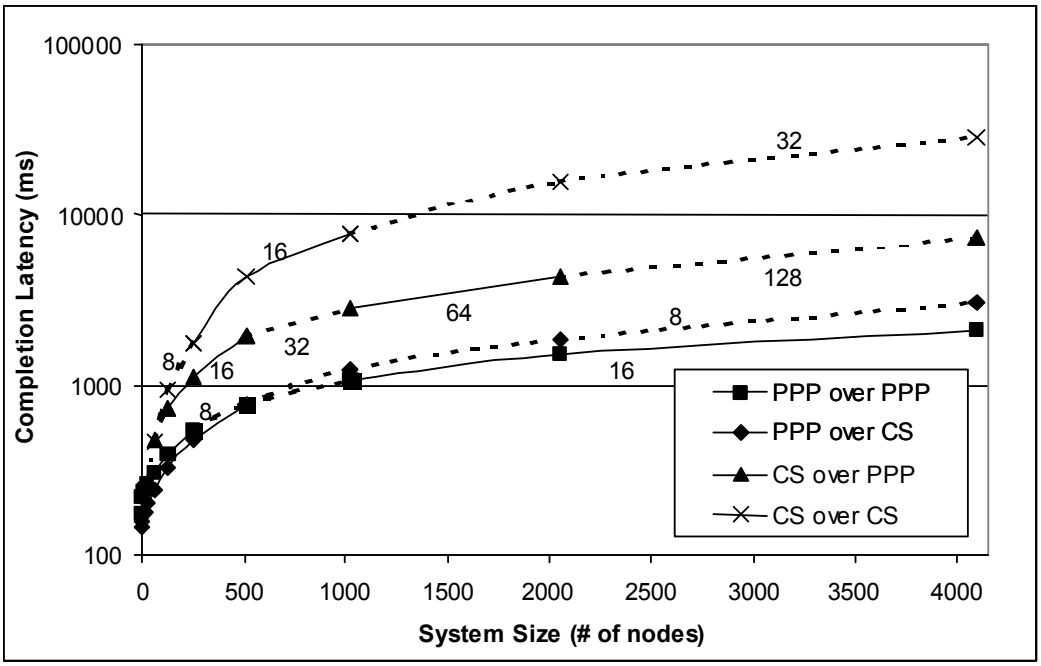

Figure 29. Completion Latency Projections with Optimal Group Sizes.

Using the bandwidth formula Equation (1) presented in [24] the control-bandwidth values can be determined. The parameters of Equation (1) are as follows: B is the bandwidth-per-node, λ is the number of layers in the system, L_i is the total number of

bytes transferred in the i^{th} layer, while f_i is the configuration-request frequency of the i^{th} layer. Since λ is set to 2 and L_i and f_i are constant for both layers, the formula is simplified. Furthermore, when adapting Equation (1) to the CM protocols, Equation (2) is produced. The additional parameters are as follows: Q is the configuration-miss rate and is again assumed to be 10%, and the functions $S_1(x)$ and $S_2(x)$ represent functions describing the average amount of data used in the lower and upper layers, respectively. As before, the variable n represents the number of nodes in the system and g represents the group size while f_i simplifies to the system's configuration-request frequency, f. As in the completion-latency projections, groupings of 8, 16, 32, 64, and 128 are investigated.

$$B = \sum_{i=1}^{\lambda} L_i \times f_i \tag{1}$$

$$B = \left(S_1(g) + Q \times (S_2(n \div g))\right) \times f \tag{2}$$

Table 2 shows the data-consumption equations for each management scheme based on its protocol. The parameter P represents the average percentage of nodes contacted in finding the configuration file. The parameter L_f is the total number of bytes transferred when the CM fails to find the configuration file, and L_s is the total number of bytes when the CM succeeds in finding the configuration file. The parameter L_r is the total number of bytes sent to the server to request a configuration file.

Table 2. Control Consumption Equations for each Management Scheme.

Management	Data Consumption Equation
PPP	$PPP_{bandwidth}(n) = P \times (n-2) L_f + L_s \quad n \geq 2$
CS	$CS_{bandwidth}(n) = L_r \quad n \geq 2$

Using the setup previously described in Chapter 6, Equation (2) is obtained. P is set to 25% of the nodes in the system. The L_f and L_s are equal in the PPP scheme since a node responds to a query by returning the query message to the requesting node. The message size is 74 bytes and breaks down as 14 bytes for Ethernet header, 20 bytes for TCP header, 20 bytes for IP, and 20 bytes of payload. The parameter L_r totals 74 bytes as well, with the same breakdown as L_f and L_s. Finally, f is set to correspond to the configuration request interval of 100 ms used in the completion latency calculations. Performing the appropriate substitutions of the data consumptions equations from Table 2 into the general bandwidth Equation (2) produces the bandwidth equations for each of the four hierarchies, shown in Table 3.

Table 3. Bandwidth Equations for System Hierarchies.

Hierarchy	Consumed Bandwidth Equations
PPP-over-PPP	$\left(P\times g\times L_f+Q\times((P\times(n\div g)\times g+P\times g)\times L_f)+L_s\right)\times f$
PPP-over-CS	$\left(L_r+Q\times(P\times(n\div g)\times L_f+L_s)\right)\times f$
CS-over-PPP	$\left(\left(P\times g\times L_f+L_s\right)+Q\times L_r\right)\times f$
CS-over-CS	$\left(L_r+(Q\times L_r)\right)\times f$

Using the equations in Table 3, the bandwidth consumed by each hierarchy over the entire network per request is calculated. Figure 30 shows the calculated results using the optimal group size found for completion latency at the appropriate system sizes. As seen in Figure 30, PPP-over-PPP incurs the most bandwidth and increases rapidly over larger system sizes. PPP-over-CS also increases with system sizes, however it consumes only 12% of the bandwidth of PPP-over-PPP. It can be observed that those hierarchies with CS in the upper layer have constant bandwidth, as can be seen by the bandwidth

consumption of CS-over-CS and CS-over-PPP. However, CS-over-CS consumes only roughly 3% of bandwidth consumed by CS-over-PPP.

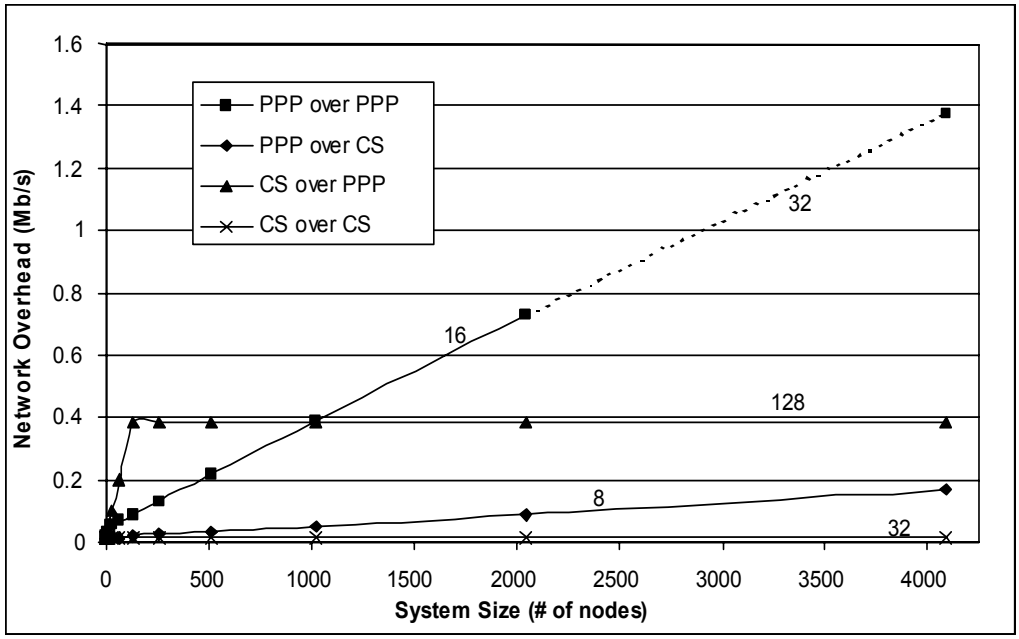

Figure 30. Network Bandwidth Consumed over Entire Network per Request.

From Figures 25, 28, 29 and 30 the optimal node configuration for an RC system is derived. Table 4 presents latency-bound, bandwidth-bound, and best overall system configurations. It should be noted that the bandwidth-bound constraint category excludes hierarchies with average completion latency values greater than 5 seconds. The low bandwidth benefits of these hierarchies cannot overcome their high completion latency.

Table 4 provides a summary of the projection results and can be used to make CM setup decisions in future systems. For small system sizes (< 32) the latency-bound and bandwidth-bound categories have the same configurations. However, as the number of nodes increases the latency-bound category prefers hierarchies containing the PPP scheme, while the bandwidth-bound category prefers the CS scheme. Furthermore, as the system size increases so do the group sizes. The best-overall category attempts to reduce

the bandwidth penalty of PPP schemes by choosing hierarchies that provide lower bandwidth requirements with minimal increase in latency. Thus, this category follows the latency-bound category until 512, since PPP-over-PPP consumes a significantly greater amount of bandwidth versus other hierarchies at system sizes greater than 512 (see Figure 30).

Table 4. System Configurations for Given Constraints over System Sizes

System Constraints	**System Size (number of nodes)**				
	< 8	8 to 32	32 to 512	512 to 1024	1024 to 4096
Latency bound	Flat CS	CS-over-CS group size 4	PPP-over-CS group size 4	PPP-over-PPP group size 8	PPP-over-PPP group size 16
Bandwidth bound*	Flat CS	CS-over-CS group size 4	CS-over-CS group size 8	PPP-over-CS group size 8	PPP-over-CS group size 8
Best Overall	Flat CS	CS-over-CS group size 4	PPP-over-CS group size 4	PPP-over-CS group size 8	PPP-over-CS group size 8

*Schemes with completion latency values greater than 5 seconds excluded.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

Traditional computing is inefficient at fine-grain data manipulation and highly parallel operations, thus RC arose to provide the flexibility and performance needed for today's HPC applications. First-generation RC systems typically entail FPGA boards coupled with general-purpose processors, thus merging the flexibility of general-purpose processors with the performance of an ASIC. RC achieves this flexibility with the use of configuration files and partial reconfiguration, which decreases configuration file sizes. RC has increased the performance of a variety of applications, such as cryptology, including encryption and decryption, and space-based processing, such as hyperspectral imaging.

Recent trends extend RC systems to clusters of machines interconnected with SANs. The HCS Lab at Florida has developed a 9-node RC cluster to examine middleware and service issues. The Air Force has a 48-node RC cluster while an earlier cluster exists at Virginia Tech. These clusters use High-Performance Networks (HPNs), which provide them with a high-throughput, low-latency network. These networks are ideally suited to transfer latency-critical configuration files among nodes. Following the COTS mentality of cluster-computing networks, COTS-based RC boards are typically used to reduce cost. Two such RC boards are Celoxica's RC1000 and Tarari's Content Processing Platform, among many others.

Such heterogeneous and distributed systems warrant an efficient method of deploying and managing configuration files that does not overshadow the performance

gains of RC computing. The Comprehensive Approach to Reconfigurable Management Architecture (CARMA) framework seeks to specifically address key issues in RC management. While some configuration-file management issues could relate to traditional HPC, other aspects may not. FPGA configuration is a critical processing component, which must be handled with care to ensure an RC speedup over a traditional system. Reusing the RC hardware during the process's execution, known as Run-Time Reconfiguration (RTR), increases system performance compared to static configuration. Furthermore, methods such as configuration compression, transformation, defragmentation, and caching can reduce configuration overhead further.

RC systems implement a Configuration Manager (CM) to handle the issues that arise from RTR and configuration overhead reduction. CARMA's configuration manager builds upon a few noteworthy designs, two of which are the RAGE from the University of Glasgow, and the reconfiguration manager from Imperial College, UK. In response to the recent trend toward COTS-based distributed RC clusters, the HCS lab presents a modular and distributed configuration manager middleware. CARMA's execution manager, analogous to the VHM in RAGE, coordinates the execution of configuration commands. The configuration manager module manages the configuration files using a layered architecture. The BIM, like the device driver of RAGE, handles low-level details of board configuration and communication. CARMA's configuration manager extends the configuration store of Imperial College's reconfiguration manager by maintaining a distributed store, and thus requires a communication module.

Since CARMA is fully distributed and targeted for thousands of nodes, various distributed files-management schemes are investigated. These include Master-Worker

(MW), which mimics the fully centralized job schedulers of today's RC designs, which farm jobs and data to workers, and Client-Server (CS), which is similar to FTP servers. Also included are "Pure" Peer-to-Peer (PPP), like the Gnutella file-sharing network, and "Hybrid" Peer-to-Peer (HPP), like the Napster file-sharing network.

In order to investigate completion latency and its components, experiments were performed in which the configuration request interval was varied. Additionally, the scalability to 16 nodes of the CM is investigated. The data gathered shows File Retrieval Time and HW Configuration Time are the dominant factors when request intervals are large. Furthermore, CM Queue Time is naturally the most dominant factor when request intervals are small. It is also observed that CARMA's configuration manager design imposes very little overhead on the system. In regard to scalability, the MW scheme is shown to not be scalable beyond 4 nodes. CS performs best for a small number of nodes (2-8), while PPP is found to be the most scalable to 16 nodes.

Analytical models were used to extrapolate latency for larger systems and to predict scalability for future testbeds. These calculations showed that in order to scale systems to thousands of nodes, hierarchical schemes are required. Since experimental results showed that CS and PPP are the only scalable schemes beyond 8 nodes, they are permutated into two-layered hierarchical schemes. These hierarchies include PPP-over-PPP, PPP-over-CS, CS-over-PPP, CS-over-CS, in which the lower-layer nodes are grouped. The data showed that in a heterogeneous hierarchy, the upper-layer scheme determines optimal group size. Moreover, for system sizes ranging from 32 to 512, PPP-over-CS with group size of 4 yields the best completion-latency performance. For larger system sizes (512 to 4096), PPP-over-PPP with a group size of 16 yields the best

completion-latency performance. In addition to completion latency, the bandwidth consumed by these hierarchies is investigated. The results showed that PPP-over-CS with a group size of 8 yields the best results, not only in terms of bandwidth but also in overall scalability, for large systems (> 512).

Directions for future work include improving CM schemes, such as the existing PPP scheme, and implementing and evaluating the HPP scheme. In addition, caching configuration files should be investigated, in particular caching algorithms, miss rates, and configuration file locality. Moreover, the analytical projections made in this thesis should be validated in a future work. Bandwidth and utilization experiments should be conducted on larger systems as well. Another direction of future work would be to extend the features of CARMA's configuration manager by supporting new SANs and RC boards, and incorporating advanced configuration file-management techniques (e.g. defragmentation, etc.)

LIST OF REFERENCES

[1] V. Ross, “Heterogeneous HPC Computing,” presented at 35th Government Microcircuit Applications and Critical Technology Conference, Tampa, FL, April 2003.

[2] A. Jacob, I. Troxel, and A. George, "Distributed Configuration Management for Reconfigurable Cluster Computing," presented at International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, NV, June 2004.

[3] I. Troxel, and A. George, “UF-HCS RC Group Q3 Progress Report” [online] 2004, http://www.hcs.ufl.edu/prj/rcgroup/teamHome.php (Accessed: May 25, 2004).

[4] I. Troxel, A. Jacob, A. George, R. Subramaniyan, and M. Radlinski, "CARMA: A Comprehensive Management Framework for High-Performance Reconfigurable Computing," to appear in Proc. of 7th International Conference on Military and Aerospace Programmable Logic Devices, Washington, DC, September 2004.

[5] Visual Numerics, Inc., “IMSL Mathematical & Statistical Libraries” [online] 2004, http://www.vni.com/products/imsl/ (Accessed: March 25, 2004).

[6] Xilinx, Inc., “Common License Consortium for Intellectual Property” [online] 2004, http://www.xilinx.com/ipcenter/ (Accessed: February 13, 2004).

[7] A. Derbyshire and W. Luk, “Compiling Run-Time Parameterisable Designs,” presented at 1st IEEE International Conference on Field-Programmable Technology, Hong Kong, China, December 2002.

[8] Chameleon Systems, Inc., “CS2000 Reconfigurable Communications Processor, Family Product Brief” [online] 2000, http://www.chameleonsystems.com (Accessed: May 12, 2003).

[9] K. Compton and S. Hauck, “Reconfigurable Computing: A Survey of Systems and Software,” ACM Computing Surveys, vol. 34, no. 2, June 2002, pp. 171-210.

[10] Xilinx, Inc., “Virtex Xilinx-II Series FPGAs” [online] 2004, http://www.support.xilinx.com/publications/matrix/virtex_color.pdf (Accessed: February 13, 2004).

[11] A. J. Elbrit and C. Paar, “An FPGA Implementation and Performance Evaluation the Serpent Block Cipher,” in Proc. of 8th International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 2000, pp. 33–40.

[12] J. R. Hauser and J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Coprocessor,” in Proc. of 5th IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, April 1997, pp. 12–21.

[13] K. H. Leung, K. W. Ma, W. K. Wong, and P. H. Leong, “FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor,” in Proc. of 8th IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, April 2000, pp. 68–76.

[14] G. Peterson and S. Drager, “Accelerating Defense Applications Using High Performance Reconfigurable Computing,” presented at 35th Government Microcircuit Applications and Critical Technology Conference, Tampa, FL, April 2003.

[15] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas and B. Schott, "Implementing an API for Distributed Adaptive Computing Systems," presented at 34th IEEE International Conference on Communications, Vancouver, Canada, April 1999.

[17] Tarari Inc., “High-Performance Computing Processors Product Brief” [online] 2004, http://www.tarari.com/PDF/HPC-BP.pdf (Accessed: February 12, 2004).

[16] Celoxica Inc., “RC1000 Development Platform Product Brief” [online] 2004, http://www.celoxica.com/techlib/files/CEL-W0307171KKP-51.pdf (Accessed: February 12, 2004).

[18] B. L. Hutchings, M.J. Wirthlin. "Implementation Approaches for Reconfigurable Logic Applications," in Proc. of 5th International Workshop on Field Programmable Logic and Applications, Oxford, England, August 1995, pp 419-428.

[19] A. Dandalis and V. Prasanna. “Configuration Compression for FPGA-based Embedded Systems,” in Proc. of 9th International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 2001, p.173-182.

[20] K. Compton, J. Cooley, S. Knol, and S. Hauck, "Configuration Relocation and Defragmentation for FPGAs," presented at 8th IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, April 2000.

[21] Z. Li, K. Compton, and S. Hauck, “Configuration Caching for FPGAs,” presented at 8th IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, April 2000.

[22] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Witt, “A Dynamic Reconfiguration Run-Time System,” presented at 5th Annual IEEE Symposium on Custom Computing Machines, Los Alamitos, CA, April 1997.

[23] N. Shiraz, W. Luk, and P. Cheung, “Run-Time Management of Dynamically Reconfigurable Designs,” in Proc. of 9th International Workshop Field-Programmable Logic and Applications, Tallinn, Estonia, September 1998, pp. 59-68.

[24] R. Subramaniyan, P. Raman, A. George, and M. Radlinski, “GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems” White Paper, currently in journal review [online] 2003, http://www.hcs.ufl.edu/pubs/GEMS2003.pdf (Accessed: April 14, 2004).

[25] K. Sistla, A. George, and R. Todd, "Experimental Analysis of a Gossip-based Service for Scalable, Distributed Failure Detection and Consensus," Cluster Computing, vol. 6, no. 3, July 2003, pp. 237-251 (in press).

[26] D. Collins, A. George, and R. Quander, "Achieving Scalable Cluster System Analysis and Management with a Gossip-based Network Service," presented at IEEE Conference on Local Computer Networks, Tampa, FL, November 2001.

[27] K. Sistla, A. George, R. Todd, and R. Tilak, "Performance Analysis of Flat and Layered Gossip Services for Failure Detection and Consensus in Scalable Heterogeneous Clusters," presented at IEEE Heterogeneous Computing Workshop at the Intnl. Parallel and Distributed Processing Symposium, San Francisco, CA, April, 2001.

[28] S. Ranganathan, A. George, R. Todd, and M. Chidester, "Gossip-Style Failure Detection and Distributed Consensus for Scalable Heterogeneous Clusters," Cluster Computing, vol. 4, no. 3, July 2001, pp. 197-209.

[29] M. Burns, A. George, and B. Wallace, "Simulative Performance Analysis of Gossip Failure Detection for Scalable Distributed Systems," Cluster Computing, vol. 2, no. 3, July 1999, pp. 207-217.

[30] Z. Li, S. Hauck, “Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation,” presented at 10th International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 2002.

[31] D. Gustavson, Q. Li, “The Scalable Coherent Interface (SCI),” IEEE Communications, vol. 34, no. 8, August 1996, pp. 52-63.

[33] S. Oral and A. George, "Multicast Performance Analysis for High-Speed Torus Networks," presented at 27th IEEE Conference on Local Computer Networks via the High-Speed Local Networks Workshop, Tampa, FL, November 2002.

[34] S. Oral and A. George, "A User-level Multicast Performance Comparison of Scalable Coherent Interface and Myrinet Interconnects," presented at 28th of IEEE Conference on Local Computer Networks via the High-Speed Local Networks Workshop, Bonn/Köswinter, Germany, October 2003.

[35] R. Brightwell , L. Fisk, “Scalable Parallel Application Launch on Cplant,” in Proc. of the ACM/IEEE conference on Supercomputing , November 2001, Denver, Colorado, p. 40.

[36] “The Earth Simulator Center” [online] 2004, http://www.es.jamstec.go.jp/esc/eng/ (Accessed: June 24, 2004).

[37] P. Kirk , “Gnutella – Stable – 0.4” [online] 2003 http://rfc-gnutella.sourceforge.net/developer/stable/index.html (Accessed: June 24, 2004).

[38] R. Schollmeier “A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications,” presented at the IEEE International Conference on Peer-to-Peer Computing, Linköping, Sweden, August 2001.

[39] Napster, LLC, “Napster.com,” [online] 2004, http://www.napster.com (Accessed: April 14, 2004).

[40] P. Ganesan, Q. Sun, and H. Garcia-Molina. “YAPPERS: A Peer-to-Peer Lookup Service over Arbitrary Topology,” presented at 22nd Annual Joint Conf. of the IEEE Computer and Communications Societies, San Francisco, April 2003.

[41] Dolphin Interconnect Solutions Inc.“Dolphin Interconnect Solutions” [online] 2004, http://www.dolphinics.com/ (Accessed: April 15, 2004).

[42] M. Baker and J. Ousterhout, "Availability in the Sprite Distributed File System," ACM Operating Systems Review, vol. 25, no. 2, April 1991, pp. 95-98.

BIOGRAPHICAL SKETCH

Aju Jacob received two bachelor's degrees, one in computer engineering and the other in electrical engineering, from the University of Florida. He is presently a graduate student in the Department of Electrical and Computer Engineering at the University of Florida. He is working on his Master of Science in electrical and computer engineering with an emphasis in computer systems and networks.

Currently, Aju is a research assistant in the High-performance Computing and Simulation Research Laboratory at the University of Florida. His research focuses on reconfigurable computing. Previously he spent a summer doing research in the Information Systems Laboratory at the University of South Florida. This research focused on design and implementation of an FPGA-based CICQ switch architecture.

Aju has gained valuable industry work experiences from internships and projects. As part of the IPPD program, he worked in a team environment for Texas Instruments. The project involved designing and implementing a high-speed test interface. He also gained experience during his internship in AOL Time Warner's Web Fulfillment Division. Here he designed and implemented web pages and reporting system for e-business. He was also exposed to the latest web design tools and technologies.