SHMEM+ AND SCF: SYSTEM-LEVEL PROGRAMMING MODELS FOR SCALABLE RECONFIGURABLE COMPUTING

By

VIKAS AGGARWAL

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
UNIVERSITY OF FLORIDA
2011
I dedicate this dissertation to my family (my parents and my caring sister), and my friends. I am truly grateful to you all.
ACKNOWLEDGMENTS

I would like to thank God and my fate; this dissertation could not have materialized without either. Mom, Dad, and my dear sister, I am truly grateful for your patience, prayers, and support (amongst many other things) – thank you for being so understanding. I would like to thank my committee, Dr. Herman Lam and Dr. Prabhat Mishra, for their time, advice and assistance as well as my advisors, Dr. Alan D. George and Dr. Greg Stitt, for their guidance, direction, and significant support through this work. I would also like to thank my friends, Swami, Kunal, Rick, and Dana for their support, care, and help in all the big and little ways – I can’t thank you all enough. Additionally, I would like to thank my former and current lab members, in particular Rafael Garcia, Changil yoon, and Kishore Yalamanchili for helping me in various parts of this research. And lastly, I would like to thank the activity of running! You really helped me keep a sane head on my shoulders, helped me resolve many problems on a fine sunny day by myself, helped me come to peace with things beyond my control and kept me from giving up on myself several times. I would like to thank all the guys with whom I have shared several runs. I am grateful to the group of “Team Asha” runners for all the memories that I will treasure. This work is supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422 and by equipment and/or tools provided by Altera and GiDEL.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>7</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>10</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1  INTRODUCTION</td>
<td>12</td>
</tr>
<tr>
<td>2  BACKGROUND AND RELATED RESEARCH</td>
<td>18</td>
</tr>
<tr>
<td>2.1 Reconfigurable Computing</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Parallel-Programming Models</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Performance Modeling</td>
<td>23</td>
</tr>
<tr>
<td>3  SHMEM+: A MULTILEVEL-PGAS PROGRAMMING MODEL FOR SCALABLE RC SYSTEMS</td>
<td>26</td>
</tr>
<tr>
<td>3.1 Multilevel PGAS</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Overview of SHMEM+</td>
<td>32</td>
</tr>
<tr>
<td>3.2.1 SHMEM+ Interface</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2 SHMEM+ Application Example</td>
<td>35</td>
</tr>
<tr>
<td>3.2.3 Design of SHMEM+</td>
<td>36</td>
</tr>
<tr>
<td>3.2.4 Design Methodology with SHMEM+</td>
<td>38</td>
</tr>
<tr>
<td>3.3 Experimental Results</td>
<td>39</td>
</tr>
<tr>
<td>3.3.1 Benchmarking Performance of Communication Routines</td>
<td>40</td>
</tr>
<tr>
<td>3.3.2 Case Study 1: Content-Based Image Retrieval (CBIR)</td>
<td>43</td>
</tr>
<tr>
<td>3.3.3 Case Study 2: Two-Dimensional FFT</td>
<td>50</td>
</tr>
<tr>
<td>3.4 Conclusions</td>
<td>53</td>
</tr>
<tr>
<td>4  PERFORMANCE MODELING FOR MULTILEVEL COMMUNICATION IN SHMEM+ (PHASE 2)</td>
<td>55</td>
</tr>
<tr>
<td>4.1 Overview of Multilevel Communication Model</td>
<td>56</td>
</tr>
<tr>
<td>4.2 Early DSE in Applications</td>
<td>60</td>
</tr>
<tr>
<td>4.2.1 Performance Estimation</td>
<td>62</td>
</tr>
<tr>
<td>4.2.2 Experimental Results</td>
<td>66</td>
</tr>
<tr>
<td>4.3 Optimizing SHMEM+ Functions</td>
<td>68</td>
</tr>
<tr>
<td>4.3.1 Performance Estimation</td>
<td>69</td>
</tr>
<tr>
<td>4.3.2 Experimental Results</td>
<td>71</td>
</tr>
<tr>
<td>4.4 Self-Tuning SHMEM+ Library</td>
<td>72</td>
</tr>
<tr>
<td>4.4.1 Experimental Results</td>
<td>74</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3-1</td>
<td>Baseline functions currently supported in SHMEM+ library.</td>
</tr>
<tr>
<td>3-2</td>
<td>End-to-end latency of transfers between various combinations of devices for traditional SHMEM on Mu cluster, and SHMEM+ on Mu cluster and Novo-G.</td>
</tr>
<tr>
<td>3-3</td>
<td>Major factors that contribute towards increased developer productivity when using SHMEM+.</td>
</tr>
<tr>
<td>3-4</td>
<td>Development hours spent in developing CBIR application. Time is reported in terms of 8-hour work days.</td>
</tr>
<tr>
<td>3-5</td>
<td>Development hours spent for developing Two-dimensional FFT application.</td>
</tr>
<tr>
<td>4-1</td>
<td>Observed time and estimated time to perform gather operation by single-FPGA designs using three different approaches on Novo-G.</td>
</tr>
<tr>
<td>4-2</td>
<td>Observed time and estimated time to perform gather operation for quad-FPGA designs on Novo-G using Approaches 1 and 2.</td>
</tr>
<tr>
<td>4-3</td>
<td>Observed time and estimated time to perform gather operation for quad-FPGA designs on Novo-G using Approach 3.</td>
</tr>
<tr>
<td>4-4</td>
<td>Estimating performance of packetized transfers between two remote FPGAs on Novo-G for $P = 2$MB and 512KB.</td>
</tr>
<tr>
<td>4-5</td>
<td>Relative error between estimated bandwidth and observed bandwidth of transfers between remote FPGAs.</td>
</tr>
<tr>
<td>4-6</td>
<td>Performance improvement for transfers between remote FPGAs obtained by self-tuned SHMEM+ library vs. baseline.</td>
</tr>
<tr>
<td>5-1</td>
<td>SCF library primitives.</td>
</tr>
<tr>
<td>5-2</td>
<td>Execution time and speedup of target-tracking application under different mapping scenarios for three systems.</td>
</tr>
<tr>
<td>5-3</td>
<td>Productivity improvement for target-tracking application.</td>
</tr>
<tr>
<td>5-4</td>
<td>Overhead measurements for target-tracking application.</td>
</tr>
<tr>
<td>5-5</td>
<td>Major factors that contribute to increased developer productivity using SCF.</td>
</tr>
<tr>
<td>5-6</td>
<td>Computation and communication performance of backprojection task on a CPU or FPGA.</td>
</tr>
<tr>
<td>5-7</td>
<td>Performance of backprojection application under different mapping scenarios.</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>13</td>
</tr>
<tr>
<td>2-1</td>
<td>20</td>
</tr>
<tr>
<td>3-1</td>
<td>27</td>
</tr>
<tr>
<td>3-2</td>
<td>28</td>
</tr>
<tr>
<td>3-3</td>
<td>29</td>
</tr>
<tr>
<td>3-4</td>
<td>30</td>
</tr>
<tr>
<td>3-5</td>
<td>32</td>
</tr>
<tr>
<td>3-6</td>
<td>35</td>
</tr>
<tr>
<td>3-7</td>
<td>37</td>
</tr>
<tr>
<td>3-8</td>
<td>37</td>
</tr>
<tr>
<td>3-9</td>
<td>38</td>
</tr>
<tr>
<td>3-10</td>
<td>41</td>
</tr>
<tr>
<td>3-11</td>
<td>42</td>
</tr>
<tr>
<td>3-12</td>
<td>43</td>
</tr>
<tr>
<td>3-13</td>
<td>45</td>
</tr>
<tr>
<td>3-14</td>
<td>49</td>
</tr>
<tr>
<td>3-15</td>
<td>50</td>
</tr>
<tr>
<td>3-16</td>
<td>51</td>
</tr>
<tr>
<td>3-17</td>
<td>53</td>
</tr>
</tbody>
</table>
Heterogeneous computing systems comprised of FPGAs coupled with standard microprocessors are becoming an increasingly popular solution for building future high-performance computing (HPC) and high-performance embedded computing (HPEC) systems due to their higher performance and energy efficiency versus their CPU-only counterparts. Unfortunately, the time and difficulty associated with developing scalable, parallel applications for reconfigurable computing (RC) platforms is often prohibitive, making it difficult to exploit the potential gains in performance and energy savings. Design and implementation of applications for these systems involve many system-wide considerations such as algorithm decomposition and architecture mappings to exploit multiple levels of parallelism, inter-device communication and control, and system-level debug and verification. Thus, system-level languages with constructs for expressing multiple levels and forms of parallelism are vital for productive implementation of designs.

In order to improve a developer’s productivity as well as the portability of scalable RC applications, we propose and analyze two different approaches for establishing communication in parallel RC applications. The first approach extends the traditional partitioned, global-address-space (PGAS) model to a multilevel abstraction by integrating a hierarchy of multiple memory components present in reconfigurable HPC.
systems into a single virtual memory layer. Based on this model, we adapt the SHMEM communication library to become what we call SHMEM+, the first known SHMEM library enabling coordination between FPGAs and CPUs in a reconfigurable HPC system. The second approach provides a system coordination framework (SCF) based on a message-passing programming model. SCF enables transparent communication and synchronization between tasks running on heterogeneous processing devices in a system by hiding low-level communication details while presenting a uniform communication interface across heterogeneous devices. Besides improving developer productivity, both of these approaches enhance the portability of applications (by hiding the platform-specific details from applications).

Further improvements in productivity are obtained by combining techniques for modeling performance with the aforementioned approaches for application design. By allowing developers to estimate the performance of their application, modeling enables the developers to make critical design decisions before undertaking an expensive implementation. Case studies illustrate the merits and effectiveness of the proposed approaches through increased performance and portability of applications along with improvement in developer productivity. The resulting communication libraries and coordination framework are projected to provide large productivity improvements, thus expanding the use of RC technology in the fields of HPC and HPEC.
The power bottleneck created by high clock frequencies has forced computer architects to consider alternative methods for increasing system performance, focusing on parallel [1] and often heterogeneous architectures [2]. Systems from domains ranging from embedded systems [3, 4] to high-performance computing [5–9] now increasingly combine microprocessors with fixed-logic, reconfigurable-logic, and/or heterogeneous multicore and many-core devices [2, 10–12]. These technologies are driving systems to become ever more powerful and efficient, but unfortunately also more complex to program, with multiple types and levels of hardware parallelism to be understood and exploited.

A special class of such systems featuring reconfigurable computing (RC), based on closely coupled microprocessors and FPGAs, offers an attractive solution for both high-performance computing (HPC) and high-performance embedded computing (HPEC) systems. Numerous studies have demonstrated that RC systems can achieve performance improvements ranging from $10 \times$ [13] to more than $1000 \times$ [9, 14] over their microprocessor-based counterparts while concomitantly reducing energy consumption. Despite their superior performance, RC systems have yet to make a significant impact on the HPC and HPEC market, largely because of increased complexity of application development.

Application development for RC systems requires specification of computation, communication, and control for FPGAs and other components such as microprocessors and serial/parallel bus interfaces. Design and implementation of these systems involve many system-wide considerations such as algorithm decomposition and architecture mappings to exploit multiple levels of parallelism, inter-device communication and control, and system-level debug and verification. Although some of these issues can be explored through modeling, many issues are left unresolved until an actual
implementation. Thus, system-level languages with constructs for expressing multiple levels and forms of parallelism are vital for efficient design implementation.

Figure 1-1 shows an overview of the programming languages and tools employed by developers for implementing an RC application. Although programming languages for device-level design continue are evolving and have raised the level of implementation abstraction above HDL to a level associated with HLL-based design, system-level design issues have received limited attention. Similar to MPI [15], SHMEM [16, 17], UPC [18, 19], etc. in the conventional parallel-computing community, coordination tools for RC systems are needed to harness the best of the device-level design languages and yet operate at a higher level of abstraction, expressing and managing computations and data across multiple devices. However, there are characteristic differences between RC systems and traditional parallel systems, such as additional levels of memory and communication in the system as well as different execution models of heterogeneous
devices present in the system which warrant a programming model that can address these differences. Currently, application developers employ ad-hoc methods and multiple libraries (and APIs) to develop RC applications, which require significant coding modifications for porting an application to a different system. As a result, the development productivity for scalable RC applications has suffered and applications have lacked portability. In addition, most existing formulation tools that are employed for modeling and early design-space exploration (DSE) lack a seamless transition to design tools which is essential for utilizing the knowledge gained by DSE.

This research focuses on defining and developing novel tools for establishing communication in scalable RC applications. To realize this goal, we explore and analyze two complementary approaches to raise the level of abstraction of communication for RC applications, SHMEM+ and System Coordination Framework (SCF). SHMEM+ is an extended and adapted version of the SHMEM communication library [17] for heterogeneous systems, based on Partitioned, Global Address Space (PGAS) [20]. By contrast, SCF provides an approach based on a message-passing programming model for exchanging data between heterogeneous devices in a system. An application designer will typically choose between these two approaches based upon the requirements of a specific application and preference for a programming model, which is similar to the choice between programming models based on message-passing and PGAS in the traditional parallel-programming community. Because these approaches provide a higher level of abstraction, thus distancing a developer from details of the actual implementation and attainable performance, modeling and DSE tools become important to provide high-performance communication as well as to allow the developers to accurately predict performance (to avoid wasted effort). Both of the aforementioned approaches provide necessary bridges to formulation tools (existing or new tools proposed in this research) for performing early DSE, while facilitating a smoother transition from the formulation stage to the design stage.
The research presented in this document is sub-divided into three phases. In the first phase, we explore techniques for adapting a programming model based on PGAS for scalable RC systems. We extend the traditional PGAS model to a multilevel-PGAS programming model, which abstracts various levels of memory hierarchy available in such systems and presents the designer with a flattened, unified view of system memory. Based on this model, we extend and adapt the SHMEM communication library to become what we call SHMEM+, the first known SHMEM library that enables coordination between FGPMs and CPUs in a reconfigurable system. We present the design of SHMEM+ and describe the methodology for developing applications using SHMEM+. Our design of the SHMEM+ library is highly portable and provides peak communication bandwidth comparable to a vendor-proprietary version of SHMEM. In addition, applications designed with SHMEM+ yield improved developer productivity compared to current methods of multi-device RC design and exhibit a high degree of portability. We analyze the performance and benefits of our implementation of SHMEM+ through two case studies on two different platforms.

One important challenge when using libraries such as SHMEM+ is choosing an appropriate communication strategy. Variances include, for example, who initiates the transfer, what functions are employed, what intermediate steps are involved, etc. Such factors can have a significant impact on the overall performance of an application. In order to evaluate the impact of communication on application performance and obtain optimal performance, a concrete understanding of the underlying communication infrastructure is often imperative. In the second phase of this research, we introduce a new performance model, the multilevel communication model, for estimating the performance of various data transfers encountered in SHMEM+. Such a model can lead to improvements in application performance and productivity. We illustrate the merits of our model using three use cases. In the first use case, the model enables application developers to perform early design-space exploration of communication
patterns in their applications before undertaking the laborious and expensive process of implementation, yielding improved performance and productivity. Second, the model enables system developers to quickly optimize the performance of data-transfer routines within tools such as SHMEM+ when these are being ported to a new platform. Third, the model augments the SHMEM+ communication library to automatically improve the performance of data transfers by self-tuning its internal parameters to match platform capabilities. Results from our experiments with these use cases suggest marked improvement in performance, productivity, and portability.

In the third phase of this research, we explore an alternate approach for establishing communication between various devices in a heterogeneous system. To complement the research in the first two phases, the research in this phase extends and adapts concepts based on a message-passing programming model, commonly employed in traditional HPC and HPEC systems, for heterogeneous systems. We investigate a novel framework called System Coordination Framework (SCF), to simplify application design for heterogeneous systems by enabling transparent communication and synchronization between tasks running on different devices. Our framework consists of a library of message-passing coordination primitives suitable for potentially any language or device, a framework that allows an application to be expressed as a static task-graph and each task of an application to be defined in potentially any language, and a set of tools that can create customized communication methods for a given system architecture based on the mapping of tasks to devices. With SCF, many low-level architectural details are hidden from application developers by allowing them to specify coordination between tasks using message-passing primitives when defining each task in a language (or tool) of their choice. By hiding low-level architectural details from an application designer, SCF can improve developer productivity, provide higher levels of application portability, and offer rapid design-space exploration of different task/device mappings. We present the technical details of the framework and an implementation of required tools which
we developed to demonstrate its effectiveness. To further improve productivity, we incorporate support for existing formulation tools such as RCML [21] and provide automated translation of models specified using RCML into a format compliant with SCF (i.e., task graphs).

This document consists of background, technical details, and results from each phase of research and is organized as follows. Background and existing research pertaining to the research in this document are summarized in Chapter 2. Chapter 3 provides a detailed description and analysis of our multilevel PGAS model and the SHMEM+ communication library. Next, Chapter 4 augments the research in Phase 1 by developing a performance model called the multilevel communication model for representing various data transfers encountered in SHMEM+ and for estimating performance. In Chapter 5, a detailed description and evaluation of SCF is presented. Chapter 6 compares and contrasts SHMEM+ and SCF based on certain characteristics. Finally, the conclusions from this research are provided in Chapter 7. Detailed information regarding the RC platforms and the applications used throughout this research are provided in Appendix A and Appendix B respectively.
CHAPTER 2
BACKGROUND AND RELATED RESEARCH

The background and related research presented in this chapter are split into three sections. Section 2.1 provides a brief and relevant background from the field of reconfigurable computing. Section 2.2 discusses the existing parallel-programming models that are commonly employed for developing parallel applications, emphasizing heterogeneous systems. Finally, Section 2.3 discusses the existing methods for modeling communication and application performance, with additional focus on RC systems.

2.1 Reconfigurable Computing

In this section we present a brief, related overview of the field of RC, focusing mainly on the aspects of application development on RC systems. For a comprehensive literature survey from the field of RC, see Compton and Hauck’s work in [22]. Although RC employs a variety of devices such as Programmable Logic Arrays (PLAs) or Complex Programmable Logic Devices (CPLDs), Field-Programmable Gate Arrays (FPGAs) have become the predominant enabling technology in RC systems. Unprecedented levels of computational density, increased internal memory bandwidth, and improved support for double-precision floating-point arithmetic in modern FPGAs have made them an extremely attractive technology in the fields of HPC and HPEC.

Reconfigurable HPC and HPEC systems are usually comprised of one or more FPGAs working alongside a host microprocessor, the former of which acts as an accelerator used to speed up key computations running on the system. FPGAs are employed in such systems in multiple ways ranging from a peripheral device on a PCIe-based card to an accelerator residing in a CPU socket on the server’s motherboard. Each form of RC-system architecture offers a varying degree of coupling between the FPGA and host CPU, which can significantly affect the complexity and efficiency of communication between the two devices. This research focuses
on alleviating problems and issues with establishing communication between heterogeneous devices in scalable RC systems.

RC-application developers traditionally programmed RC systems using Hardware Description Languages (HDLs) such as VHDL and Verilog to describe the hardware portion of an application. The software portion of the application that interacts with the FPGA is typically specified in a programming language such as C/C++, using an API to access the FPGA (e.g., writing and reading data from memory, accessing FPGA registers). More recently, High-Level Languages (HLLs), offering a level of abstraction closer to software, have been employed for describing hardware circuits for an FPGA. Although easier to use than VHDL, most of these languages are still relatively immature and can suffer from some limitations. For example, HLLs such as SystemC [23] and Impulse C\textsuperscript{TM}[24] offer improved user productivity, but trade off execution and resource efficiency for limited portability and reusability. The Carte programming environment may achieve higher efficiency but at the cost of portability [25]. Additionally, while enabling faster hardware development for an FPGA, HLLs typically only address a subset of the system-level issues involved with programming on scalable RC systems. The communication models and libraries presented in this research focus on resolving system-level issues encountered in the development of scalable RC applications while permitting flexibility in the tools and languages used for describing hardware circuits for FPGAs.

2.2 Parallel-Programming Models

Traditionally, developers of parallel programs have performed coordination between tasks using either message-passing libraries such as MPI [15] or shared-memory libraries such as OpenMP [26]. Recently, languages and libraries that present a partitioned global address space (PGAS) to the programmer, such as UPC [18, 19] and SHMEM [17], have become more visible and popular. These languages provide a simple interface for developers of parallel applications through implicit or explicit one-sided
Figure 2-1. Source code for a SHMEM application that transfers an array of data from PE1 to PE0.

```
#include <stdio.h>
#include <shmem.h>
#include <intrinsics.h>

int me, npes, i;
int *source, *dest;
main()
{
   shmem_init();
   me = my_pe(); /* Get PE information */
   source = shmalloc(4*8); /* Allocate data in shared memory */
   dest = shmalloc(4*8);
   if(me == 1) { /* Perform send on PE 1 */
      for(i=0; i<8; i++)
         source[i] = i+1;
      /* put source data at PE1 to dest at PE0*/
      shmem_putmem(dest, source, 8*sizeof(dest[0]), 0);
   }
   /* Make sure the transfer is complete */
   shmem_barrier_all();
   /* PE 0 now contains array from PE 1 */
}
```

data-transfer functions while providing comparable performance to message-passing libraries [27]. In particular, the SHMEM communication library is currently experiencing a growth in interest in the HPC community due to its innate simplicity, low overhead, and emphasis upon explicit, high-bandwidth, one-sided communications. The SHMEM communication library consists of a set of routines that allow exchange of data between cooperating parallel processes (called processing elements or PEs). Programs developed using SHMEM follow the single-program, multiple-data model (SPMD) [28], and are similar in style to programs based on MPI. SHMEM routines support remote data transfers through put (or get) operations, which transfer data to (or from) a different PE using remote pointers which allow direct references to data objects owned by the remote PE. Several other operations are also supported such as broadcast, collective reduction, synchronization operations, and atomic memory operations. Figure 2-1 shows the source code of an example application which uses the SHMEM library to transfer an array of data from “source” variable on PE1 to “dest” variable on PE0.
Since most of such languages and libraries were developed for traditional HPC systems, they have been typically limited to homogeneous systems of microprocessors connected via commodity interconnect technology (e.g., Ethernet, InfiniBand). Although researchers have extended some of these libraries to a heterogeneous mix of microprocessors connected via different network technologies [29–31], the heterogeneity supported by these libraries has been limited to different types of microprocessors.

Owing to the emergence of a plethora of devices that are used for application acceleration and coupled with microprocessors in HPC, there has been a quest for exploring parallel-programming models that are better suited for heterogeneous systems. Some researchers have attempted to build hybrid models using multiple models for a system [32]. System-level libraries and languages such as MPI and UPC were used for coordination between tasks executing on different nodes of a cluster, and libraries such as OpenMP for coordination between tasks within each node. Hybrid models require the developer to partition their design into multiple levels and acquire expertise with multiple programming models, languages, libraries, and tools. By contrast, this research attempts to abstract these details from an application developer presenting them with an integrated programming model and a uniform communication interface.

Other research groups have shown interest in asynchronous execution in the PGAS model, leading to Asynchronous PGAS (APGAS) [33], which lays the foundation for active-message programming and fine-grained concurrency. However, APGAS is largely tailored towards spawning massively parallel, multi-threaded kernel computations at run-time on accelerators such as GPUs and is not well-suited for FPGA devices.

The work in [34] extends the UPC programming model to abstract a system of microprocessors and accelerators through a two-level hierarchy of parallelism. While their work relates to SHMEM+ in that both seek to provide a unified programming model,
their approach is quite different than ours in that it relies on identifying and extracting sections of code in a UPC program that are amenable to hardware acceleration, re-directing them through a source-to-source translator and a high-level synthesis tool to generate hardware designs. Instead of providing a means for creating hardware designs, SHMEM+ provides a parallel programming model, amenable to HPC systems that have a hierarchy of computational devices and memory resources, and deferring to and leveraging the efficiency of existing and emerging high-level synthesis tools to raise the abstraction for device-level design and generate the appropriate hardware.

TMD-MPI [35] extends the MPI library to support message-passing between heterogeneous devices, such as a mix of FPGAs and microprocessors. Although conceptually similar, SCF, the coordination framework presented in Phase 3 of this research, has several novel aspects. For instance, TMD-MPI uses a dynamic, task-graph representation of an application and a communication-architecture based on packet-switched, Network-On-Chip (NoC) design. In contrast, applications are defined as static task graphs in SCF, which, because of a known mapping, can yield designs which provide improved performance.

Auto-Pipe and the X language presented in [36, 37] provide a framework for developing pipelined applications distributed across the resources of a heterogeneous system. Although SCF adopts a similar approach, it extends this concept to allow applications with arbitrary task-graphs. To provide a high-level abstraction to developers, Auto-Pipe presents an abstraction of FIFOs for communication between any two tasks of an application. While simplifying the job of developers, this abstraction requires Auto-Pipe to re-organize and packetize the data for efficient transfers between any two tasks. By contrast, SCF employs message-passing semantics for communication, which allow developers to specify an appropriate message size for obtaining high performance for data transfers. Recently, much effort has been channeled towards standardizing the interface between microprocessors and enhanced devices such as FPGAs [38] and
GPUs [39]. Focus of these efforts has been on low-level interaction between accelerator devices and microprocessors, not making any assumptions on higher-level forms of communication and synchronization. The scope of SCF and SHMEM+ is larger and complements such efforts, as they can overlay a communication framework on top of such existing APIs/languages.

2.3 Performance Modeling

There are various communication models prevalent in the field of HPC that aim to provide developers with a better understanding of the underlying communication infrastructure. Such models provide valuable ideas and useful insight towards our performance model for multilevel communication proposed in Phase 2 of this research for estimating the performance of data transfers in SHMEM+.

Communication models have been prevalent in the field of HPC for estimating the performance of communication in parallel applications. Models like Parallel Random Access Machine (PRAM) [40], Bulk Synchronous Parallel (BSP) [41], LogP [42], LogGP [43], PlogP [44], etc. aim to be fairly generic and architecture-independent, and provide high-level estimates of communication performance for parallel programs. However, these models were developed for traditional HPC systems based on a homogeneous set of microprocessors and cannot be directly employed to represent the multilevel transfers in reconfigurable HPC systems. Other researchers have attempted to build system-level models for heterogeneous systems. Heterogeneous LogGP (HLogGP) [45] considers extensions of LogGP for multiple processor speeds and communication networks within a cluster. In [46], system-level modeling concepts form the basis for a proposed model for heterogeneous clusters. However, the primary emphasis of these models was to target systems based on heterogeneous microprocessors and thus these approaches are ill-suited for multilevel systems based on accelerators.

In contrast, comparatively few efforts have focused on system-level performance prediction of RC systems. In [47], RC Amenability Test (RAT) defines an analytical
model for performance estimation of an RC algorithm for a given RC platform prior to any implementation. Although it provides a fairly accurate representation of an algorithm targeting a single device, it was not intended to model the effects of scalable, multi-device applications. By contrast, our proposed model focuses on modeling and estimation of the communication performance in large-scale RC systems while leveraging models such as RAT for estimating the performance of computational parts of the application. The work in [48] proposes a set of application-specific analytical equations for performance prediction of iterative synchronous programs running on heterogeneous clusters with RC devices. However, [48] does not delve into the details of communication modeling and therefore does not consider the nuances of multilevel communication enabled by tools such as SHMEM+.

This research also shares and leverages concepts from other academic projects that aim to improve developer productivity for heterogeneous systems through modeling and simulation. Ptolemy [49] studies modeling, simulation, and design of concurrent real-time embedded systems based on different models of computation. Other researchers have proposed techniques for abstract modeling and estimation to allow users to quickly and more abstractly build model representations of their proposed applications. These models can be easily analyzed and modified in order to efficiently perform design-space exploration before implementation. We leverage one such formulation tool in our work to enable abstract modeling, the RC Modeling Language (RCML). RCML [21] provides hierarchical models for the algorithm, system architecture, and total application mapping with specialized constructs to express parallelism, communication patterns, and other common aspects in RC. RCML is intended to allow users to quickly model systems before lengthy coding of an implementation, using abstract constructs and quantitative attributes to define behavior. SCF, described in Phase 3 of this research, provides automated translation of RCML models to task-graph
specifications required by SCF to further improve productivity by allowing users to quickly transition from formulation to design.
CHAPTER 3
SHMEM+: A MULTILEVEL-PGAS PROGRAMMING MODEL FOR SCALABLE RC SYSTEMS (PHASE 1)

This chapter presents a multilevel-PGAS programming model for RC systems, which abstracts the memory hierarchy available in the system, and presents the designer with a flattened, unified view of the system memory. Furthermore, we employ the model to develop SHMEM+ (i.e. an extended SHMEM library), the first known implementation of SHMEM that enables communication and synchronization between FPGAs and CPUs in scalable RC systems. Using SHMEM+, designers can create scalable, parallel applications that execute over a mix of microprocessors and FPGAs. The high-level abstraction provided by SHMEM+ can yield significant improvement in developer productivity. Concomitantly, for the decomposed tasks of a parallel application, developers of FPGA cores can employ high-level synthesis tools and languages (e.g. Impulse C™, Carte™, Handel-C) for creating hardware designs for FPGAs to further improve productivity. This study analyzes the performance of our implementation of SHMEM+ and investigates its inherent strengths through two case studies on two different platforms. Although the work in this chapter focuses on HPC systems and applications, the proposed multilevel PGAS model and SHMEM+ library can also be extended to systems based on other types of accelerators such as GPUs, many-core processors, etc.

The remainder of this chapter is organized as follows. A detailed description of the multilevel-PGAS programming model is presented in Section 3.1. Section 3.2 gives an overview of the design of SHMEM+. In Section 3.3, the performance of data-transfer routines available in SHMEM+ is benchmarked. The section also presents two case studies to illustrate the design methodology and evaluate the advantages of application design using SHMEM+. Finally, conclusions from this phase of research are summarized in Section 3.4.
3.1 Multilevel PGAS

Next-generation RC systems will be targeting FPGA devices in their system architectures in exotic ways to extract performance, ranging from closely coupled, in-socket accelerators to PCIe-based accelerator cards. Figure 3-1 depicts an example RC system, where every node contains a set of processing units (PUs), each a microprocessor or FPGA. With FPGA devices and multicore CPUs, each with one or more associated memory modules, all within a single node, becoming pervasive in high-end computing systems, the existence of multiple levels of memory hierarchy and different permutations of communication is becoming increasingly difficult to ignore. As a result, application developers are presented with a daunting task of orchestrating data amongst heterogeneous devices and several memory components by employing multiple APIs.

There is a need for a parallel-programming model that provides application developers with a high level of abstraction and presents a simplified view of the system, somewhat akin to that provided by the global memory layer in PGAS. However, there are various challenges involved in applying an existing parallel-programming model such as
Figure 3-2. High-level abstractions for programming heterogeneous systems. A) An ideal programming abstraction for application developers. B) A more practical and realizable approach.

PGAS to reconfigurable HPC systems. Some of the concepts and semantics associated with PGAS-based programming model on traditional systems are not directly applicable to such hybrid systems. For example, a majority of parallel programs are described using the SPMD model, where each node in the computational system executes the same program while working on a different part of input data. Heterogeneous systems comprised of devices with different programming paradigms often require an application developer to create separate programs, one for each type of device in the system, and necessitate a re-definition of SPMD for such systems. Similarly, the multi-tier memory hierarchy that exists in reconfigurable HPC systems warrants a re-examination of the distribution of the virtual, global memory layer of PGAS programming model over the physical memory resources of a system.

From application developers’ point of view, an ideal programming model should provide an abstraction where the heterogeneous devices present in the system are treated as logically equivalent PUs (Figure 3-2A). Using such a model, each PU will execute a program instance of a SPMD application, obtained by translating the source
code into logically equivalent operations in different programming paradigms. The PGAS interface on each PU would be responsible for presenting a logically homogeneous system view to the application developers. While it may provide a simplified view of the system, such an abstraction would be difficult to implement and may lead to inefficient utilization of system resources. For example, FPGA devices yield exceptional performance for computations which have a high degree of parallelism, but can lead to inefficiencies when implementing complete functionality of a SPMD program.

A more practical solution would raise the level of abstraction for application developers while making efficient use of the specialized resources present in a system. Figure 3-2B shows such an approach, where each individual task of a SPMD application is further partitioned across, and collectively executed by all the PUs on a node. Such a solution can also extend the concept of partitioned, global address space to a multilevel abstraction, which integrates a hierarchy of multiple memory components into a single, virtual memory layer. We call this model multilevel PGAS.

Figure 3-3 shows the physical distribution of memory components that form the global address space in multilevel PGAS. Memory blocks associated with all PUs in the system, irrespective of their physical location and hierarchy in the system architecture, can form a part of the virtual memory layer and have globally unique memory addresses in the system. Both CPUs and FPGAs provide interfaces required for the global memory abstraction for their corresponding memory blocks. Note that, all physical memory
blocks do not have to be a part of the PGAS. The memory blocks that do not form a part of the virtual, global memory layer can be used by their PUs for storing local variables. It should be noted that memory blocks shown in Figure 3-3 correspond only to off-chip memory resources for the focus of our work. On-chip memory structures of an FPGA such as block RAMs and register files are treated as local storage and not exposed as a part of PGAS. Such modeling of local storage is similar to that of microprocessor cache and registers, which are hidden from the PGAS layer in traditional HPC systems. Such resources were not included in the global address space in our design because the memory consistency required by parallel applications may preclude the usage of BRAMs as shared resources in most cases. However, our framework does not prevent the usage of such resources in the global address space if it can be supported by the target FPGA platform.

Figure 3-4 depicts a detailed view of the physical distribution of resources within each node and its equivalent logical abstraction provided by the multilevel-PGAS model (used by SHMEM+). Although the figure depicts two processing units per node, one CPU and one FPGA, it can be generalized to include any number and variety. As shown in Figure 3-4A, the global address space, partitioned across multiple nodes in the system, is composed of memory blocks which are physically distributed across different processing units within a node. However, the logical abstraction presented to a
designer (shown in Figure 3-4B) is a flattened view of the node’s shared memory. Thus, application designers do not have to understand the distribution of data over the physical memory resources when accessing a remote node.

The PGAS interface on each node is responsible for providing application designers with an abstraction of a single, integrated memory block. Similar to the case for memory resources, the logical view of the PGAS interface presented to the developer is different from its physical implementation. The physical implementation of the interface itself is system-dependent and can be realized in different ways by system architects. While each node provides the entire functionality required by the PGAS interface, each PU within a node may implement only a subset of this functionality. The distribution of these responsibilities amongst the PUs within a node is dictated by their capabilities in the system. For example, in our current design, the CPUs provide a majority of the SHMEM functionality and the FPGAs only provide assistance for transfers to and from the FPGA’s memory using the vendor-specific memory controllers. As future work, we intend to investigate the feasibility of FPGA-initiated transfers, which will require more extensive support from FPGAs and may lead to some resource utilization on the FPGAs by SHMEM+, unlike our current design.

The multilevel-PGAS model supports two additional features which help in attaining high performance for applications. First, it allows application developers to specify affinity of various application data to specific memory components within a node during memory allocation. Therefore, the data can be placed in a memory block closer to the processing unit that operates on it most frequently. Second, multilevel-PGAS model requires explicit transfers between local memory components. In systems equipped with multiple non-coherent memory blocks within a node, DMA operations are often employed for data transfer between different memory blocks, which are expensive operations and can significantly hamper application performance. Having explicit calls for data transfers within a local node eliminates the possibility of inefficiencies caused
3.2 Overview of SHMEM+

Using the multilevel-PGAS programming model, we extend conventional SHMEM to become what we call SHMEM+, a communication library which enables additional communication capabilities between heterogeneous devices. Using SHMEM+, designers can create highly scalable applications that execute over a mix of microprocessors and FPGAs. Previous implementations of the SHMEM API have targeted specific systems [16] and often lacked portability. SHMEM+ is built over services provided by Global Address Space NETworking (GASNet from UC Berkeley) [50] which is a language-independent, communications middleware that provides network-independent, high-performance primitives tailored for implementing parallel GAS languages. As a result, SHMEM+ can be easily ported to other systems that are supported by GASNet by simply modifying the FPGA interfaces that employ vendor-specific APIs.

SHMEM+ provides developers with a high-productivity environment for establishing communication in an RC application, by providing developers with several choices for...
Table 3-1. Baseline functions currently supported in SHMEM+ library.

<table>
<thead>
<tr>
<th>Function</th>
<th>SHMEM+ call</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>shmem_init</td>
<td>Setup</td>
<td>Initializes SHMEM library and other resources</td>
</tr>
<tr>
<td>Comm. Id</td>
<td>my_pe</td>
<td>Setup</td>
<td>Provides a unique ID for each process</td>
</tr>
<tr>
<td>Comm. size</td>
<td>num_pes</td>
<td>Setup</td>
<td>Provides number of PEs in the system</td>
</tr>
<tr>
<td>Finalize</td>
<td>shmem_finalize</td>
<td>Setup</td>
<td>De-allocates resources and gracefully terminates</td>
</tr>
<tr>
<td>Malloc</td>
<td>shmalloc</td>
<td>Setup</td>
<td>Allocates memory for shared variables</td>
</tr>
<tr>
<td>Get</td>
<td>shmem_int_g</td>
<td>P2P</td>
<td>Reads single element from a remote node</td>
</tr>
<tr>
<td>Put</td>
<td>shmem_int_p</td>
<td>P2P</td>
<td>Writes single element to a remote node</td>
</tr>
<tr>
<td>Get</td>
<td>shmem_getmem</td>
<td>P2P</td>
<td>Bulk read from a remote node</td>
</tr>
<tr>
<td>Put</td>
<td>shmem_putmem</td>
<td>P2P</td>
<td>Bulk write to a remote node</td>
</tr>
<tr>
<td>Quiet</td>
<td>shmem_quiet</td>
<td>Synch.</td>
<td>Waits for completion of outstanding puts</td>
</tr>
<tr>
<td>Barrier Sync.</td>
<td>shmem_barrier_all</td>
<td>Synch.</td>
<td>Synchronizes all the nodes</td>
</tr>
</tbody>
</table>

Data transfers between devices of a heterogeneous RC system, some of which did not exist in conventional SHMEM library. Figure 3-5 illustrates the different options for data transfers provided by SHMEM+ in a system with a CPU and an FPGA on each node. The existing transfer capabilities are marked by labels ‘a’ (CPU-only SHMEM) and ‘b’ (platform-specific APIs) in the figure, and the ones introduced by SHMEM+ are labeled as ‘x’ and ‘y’. While these additional data-transfer options simplify the process of developing parallel applications and improve productivity, the developers should understand the tradeoffs associated with such transfers. For example, direct transfer between two remote FPGAs eliminates the need for a developer to carefully orchestrate the data through the local CPUs on the source and destination nodes. However, it might occasionally reduce the opportunities of overlapping intermediate steps of communication that exist in the application. SHMEM+ does not force developers to work at a particular level of abstraction. Instead it provides transfer functions which improve productivity along with functions which allow more detailed control over data transfers, for achieving higher performance. The choice of the data-transfers employed will often depend on the characteristics and structure of target application.
3.2.1 SHMEM+ Interface

Our design of SHMEM+ as described in this chapter focuses on a subset of baseline functions selected from the entire API function set of SHMEM. In this chapter, we discuss 11 baseline functions shown in Table 3-1, which include five setup functions, four point-to-point messaging calls, and two synchronization routines. Some of these functions can be easily extended to support other SHMEM functions; such is the case for single-element and contiguous data-transfer routines. In this version of SHMEM+, we focus primarily on blocking communication. However, we also provide limited support for non-blocking communication as in the case for transfers between a CPU and its local FPGAs. More extensive support for non-blocking transfers is the focus of our ongoing research and future work. In addition, all of the various transfers are currently initiated by CPU devices which invoke SHMEM+ functions to transfer data between any two locations.

It is our objective to keep the interface of SHMEM+ consistent with previous SHMEM implementations. However, the functionality provided by SHMEM+ has been extended in various ways to incorporate support for FPGAs and provide multilevel-PGAS abstraction. Thus, SHMEM+ functions perform these extra tasks in addition to the ones performed by traditional SHMEM routines. For example, the `shmem_init` routine performs FPGA initialization (i.e. configuration of FPGA with the required hardware design) and FPGA memory-management operations, concomitant to initialization and management of CPU memory segments as performed by the traditional `shmem_init` function. The routines for data transfer (variations of `shmem_get` and `shmem_put`) perform exchanges between any two devices, such as two CPUs or between a CPU and an FPGA, etc. Based on the target memory address specified in the function, SHMEM+ identifies whether the requested data resides in CPU or FPGA memory and employs appropriate means of transferring the data. In addition, transfers to both remote and local FPGAs can be performed using the same interface, eliminating the
need for multiple APIs. Without SHMEM+, application developers must decompose the algorithm in multiple stages, using the conventional SHMEM library for system-level decomposition and lower-level vendor APIs for distributing the functions across various PUs within a node, and then carefully orchestrate the communication through multiple libraries based on the location and type of the target device. The memory allocation routine (shmalloc), which allocates memory for data variables from the shared address space, has been modified to allow users to specify the affinity of any data to a particular memory block in the system. For example, a set of data that is operated upon by an FPGA can be specified to be allocated on FPGA memory which, as explained in Section 3.1, can be beneficial for application performance. The application developer conveys this information by specifying the “type” parameter (type = 0 for CPU memory, 1 for FPGA memory) in the shmalloc function call.

3.2.2 SHMEM+ Application Example

In order to develop a better understanding and appreciation for SHMEM+, we highlight the differences introduced by SHMEM+ in an application through an example. Figure 3-6A shows the task-graph of a multi-FPGA “add-one” application along with
the desired mapping of each task onto a device (labeled alongside in the task graph). The task “Send data” which is mapped on the CPU of Node 0 sends input data to the “Add one” tasks (mapped on the first FPGA of each node) and collects the output data on completion of processing. The architecture of the target RC system for the application is presented in Figure 3-6B. Figure 3-7 lists the code snippets of the add-one application designed using (a) SHMEM+ and (b) a combination of CPU-only SHMEM and vendor-specific APIs. With SHMEM+ an application developer has the capability of transferring the input data to various FPGAs (both local and remote) using the same interface of SHMEM+. Traditionally, this transfer would have been achieved by distributing the data from CPU on Node 0 to the CPUs on other nodes. All the nodes would then have to perform a synchronization operation to ensure the receipt of data before proceeding with a transfer to their local FPGAs. Although Figure 3-7B summarizes the transfer to the local FPGA using a single function call, the process is often less than trivial and non-uniform across different FPGA platforms. Even for this simplified example, it is easy to see the benefits of employing SHMEM+ over traditional methods of application design. More complex applications with more intricate communication patterns will benefit from larger reduction in program complexity and developer effort.

3.2.3 Design of SHMEM+

Figure 3-8A illustrates the software architecture of SHMEM+. It makes use of GASNet’s Core API, Extended API, and Active Message (AM) services. The setup functions, which perform memory allocation and other initialization tasks, employ the “Core API” services of GASNet. The data transfers to/from the CPU memory were built using the “Extended API,” which provides direct support for high-level operations such as remote memory access. As a result, SHMEM+ functions that perform transfers between two CPUs can be implemented by simply providing wrappers around the underlying GASNet functions. Since transfers to/from FPGA memory are not directly supported by
Figure 3-7. Code snippets for a multi-FPGA add-one application. A) Designed using SHMEM+. B) Designed using SHMEM.

Figure 3-8. Design of the SHMEM+ library. A) Software architecture of SHMEM+. B) Data transfer example using Active Messages.

underlying GASNet functions, they were developed using the AM service in conjunction with FPGA interfaces that we created for our FPGA-platform (more details about our platform are provided in Section 3.3).
Figure 3-9. Design methodology for application development using multilevel PGAS and SHMEM+.

Figure 3-8B shows the sequence of steps involved in a transfer using Active Messages when a CPU requests data from a remote FPGA. The CPU on Node 1 initiates the transfer by calling the `shmem_getmem` function, which sends an AM request to the CPU on remote node (Node 2). Upon receiving the AM request message, Node 2 invokes an AM request handler which reads the requested data from the local FPGA in a temporary buffer and sends an AM reply message containing the requested data to the initiating node. When Node 1 receives the reply message, it invokes a corresponding reply handler to copy the incoming data into the user-specified location. The message handlers shown in the figure employ `FPGA_read` and `FPGA_write` functions, which we developed using the FPGA-board vendor’s API to communicate with FPGA memory. Due to overhead incurred by AM services and data access to/from the FPGA board, communication with an FPGA can result in slightly higher latency and lower bandwidth when compared to CPU transfers.

### 3.2.4 Design Methodology with SHMEM+

One of the important goals of multilevel PGAS and SHMEM+ is to provide developers with a framework for building large-scale RC applications using familiar techniques of parallel programming. The design methodology for building applications using SHMEM+ is described in Figure 3-9. A developer begins with a baseline
algorithm of the application (step a). A parallel algorithm is obtained by system-level decomposition (step b) of the baseline into multiple tasks, each of which is assigned to a node in the target RC system. Various conventional techniques of decomposition can be employed during this stage, such as SPMD, pipelining, etc. Each task of the parallel algorithm is further decomposed into constituent functions which are distributed amongst the processing units in each node (step c). In the following step (step d), the developer describes the functions mapped on FPGAs as hardware engines using a hardware description language (HDL) or HLLs. Finally, the remainder of the functions are described in software to be mapped on CPUs (step e). The software also provides the FPGA with control signals required by the hardware engines developed in the previous step. The functions that are mapped on the CPUs employ SHMEM+ routines to access the PGAS in the system and perform synchronization operations. Although the current version of SHMEM+ only allows CPU-initiated transfers, the capability of FPGA-initiated transfers in future can provide numerous opportunities for innovative application design. The design flow described here is further exemplified through multiple case studies in Section 3.3.

3.3 Experimental Results

In this section, we present the performance obtained for various memory transfers with SHMEM+ and compare it against the performance obtained with the vendor-proprietary, CPU-only version of SHMEM provided by Quadrics for QsNet systems. We then present two case studies to illustrate the design methodology, and evaluate various advantages of application-design using SHMEM+. To evaluate portability and scalability of applications designed using SHMEM+, we conducted our experiments on two different systems, Mu cluster and the Novo-G RC supercomputer, which are detailed in Appendix A.

The SHMEM+ library was initially developed on the Mu cluster and later ported on Novo-G. Porting the SHMEM+ library to the Novo-G system was a straightforward
process, just requiring an installation of GASNet on the new system. Ideally, any system that is supported by GASNet can be supported by SHMEM+ with ease, as in the case of Novo-G. However, in general, the process of porting SHMEM+ onto a new platform requires solving: (a) system-level issues and (b) FPGA platform-level issues. For supporting a new FPGA platform, the system architects are responsible for creating functions to interact with the FPGA board. Although the time to port SHMEM+ is largely dependent on the skills of a system developer and the complexity of the vendor APIs for the FPGA board, it should be in the order of a couple of weeks for an experienced system developer. Our current design of SHMEM+ library allows each of the four FPGAs on each node to support up to 2GB of shared memory which forms a part of the PGAS layer. The remainder of the memory is available to the FPGAs for storing local data.

### 3.3.1 Benchmarking Performance of Communication Routines

Figure 3-10 depicts performance of point-to-point communication in SHMEM+ for transfers between two CPUs on our two systems and compares it with the performance obtained by the SHMEM library from Quadrics on the Mu cluster. Bulk communication routines such as `shmem_getmem` and `shmem_putmem` attain a peak throughput of about 850MB/s on the Mu cluster. The bandwidth obtained with SHMEM+ calls, for transfers between two CPUs, is comparable to the proprietary version of SHMEM available from Quadrics for our Mu cluster. The SHMEM+ routines for these transfers benefit from direct support provided by GASNet and thus incur minimal overheads. Novo-G offers higher peak bandwidth (over 1400 MB/s, approx. 75% of the max. capacity of the network) for point-to-point transfers between two CPUs when compared to the Mu cluster due to the faster interconnect. Although the peak bandwidth obtained on Novo-G is higher than Mu cluster, the performance of GASNet for smaller message sizes is better on Mu cluster. Since there was no known implementation of SHMEM available for InfiniBand, we compared the performance of SHMEM+ with the performance of synchronous data-transfer functions present in MVAPICH [51] (an
Figure 3-10. Bandwidth of point-to-point routines using SHMEM+ and Quadrics SHMEM for transfers between two CPUs. Note that in the performance results for Mu cluster, the line graphs of SHMEM GET and SHMEM+ GET overlap each other. A) On Mu cluster. B) On Novo-G.

implementation of MPI over InfiniBand) which is a commonly used communication library. The graphs indicate that SHMEM+ outperforms MVAPICH for large data transfers and offers a higher peak bandwidth.

Figure 3-11A shows performance of data transfers between a CPU and an FPGA using SHMEM+ routines on Novo-G. The “Local PUT” and “Local GET” labels represent the bandwidth of data transfers between a host CPU and its local FPGA on the same node. The bandwidth of such local transfers is specific to the particular FPGA board and depends upon a variety of factors associated with interconnect(s) between CPU and FPGA, efficiency of the communication controller on the board, etc. Many RC systems offer a higher bandwidth for read operation from an FPGA (FPGA to CPU) when compared to write operation (CPU to FPGA). Similarly, our system yields a peak bandwidth of approximately 275MB/s for local put operations (CPU to FPGA) and approximately 1000MB/s for local get operations (FPGA to CPU). The “Remote PUT” and “Remote GET” labels represent the bandwidth of data transfers between a CPU and an FPGA on a different node. As expected, the bandwidth for such transfers is observed to be lower than the bandwidth attained for local transfers. Figure 3-11B
Figure 3-11. Bandwidth of point-to-point routines on Novo-G using SHMEM+ for transfers between different devices. A) CPU and FPGA. B) Two FPGAs. The label ‘Local’ in the graphs represents transfers between devices which are on the same node, whereas the label ‘Remote’ represents the transfers between devices on separate nodes. Note that for transfers between two FPGAS, the line graphs of Local GET and Local PUT overlap each other.

Table 3-2. End-to-end latency of transfers between various combinations of devices for traditional SHMEM on Mu cluster, and SHMEM+ on Mu cluster and Novo-G. The times are reported in microseconds for data transfer of 32 bytes.

<table>
<thead>
<tr>
<th>Transfers between</th>
<th>SHMEM (Mu)</th>
<th>SHMEM+ (Mu)</th>
<th>SHMEM+ (Novo-G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two remote CPUs</td>
<td>3.87 μs</td>
<td>3.76 μs</td>
<td>7.96 μs</td>
</tr>
<tr>
<td>CPU &amp; local FPGA</td>
<td>205.49 μs</td>
<td>213.60 μs</td>
<td>644.92 μs</td>
</tr>
<tr>
<td>CPU &amp; remote FPGA</td>
<td>211.67 μs</td>
<td>219.49 μs</td>
<td>664.70 μs</td>
</tr>
<tr>
<td>Two remote FPGAs</td>
<td>427.09 μs</td>
<td>425.27 μs</td>
<td>1292.22 μs</td>
</tr>
</tbody>
</table>

Table 3-2 reports the end-to-end latency (EEL) observed for transfers between various combinations of devices. The smallest data size for transfers in our experiments was restricted to 32 bytes by the requirements of the FPGA board. The second and third column in the table compare the latencies observed for traditional SHMEM with those observed for SHMEM+ on Mu cluster. The differences between the two are less than 5% for all the cases. The table also lists the EEL observed for transfers on
Figure 3-12. Processing steps involved in parallel algorithm for CBIR using SHMEM+.

Novo-G. The latency for data transfers using GASNet on Novo-G is higher than on Mu cluster, which concurs with the performance graphs presented earlier. From results presented in this section, it can be observed that performance of SHMEM+ compares well with conventional SHMEM for transfers between two CPUs, and SHMEM+ performs reasonably well for communication with an FPGA.

3.3.2 Case Study 1: Content-Based Image Retrieval (CBIR)

In this section, we use CBIR application (Appendix B) to illustrate the design methodology associated with SHMEM+, and evaluate various advantages of application-design using SHMEM+. The processing steps involved in the parallel algorithm employed in our experiments are shown in Figure 3-12. We employed the design-flow described in Section 3.2 to derive this algorithm as follows:

Step a The serial algorithm iterates over the set of images in the database to calculate their feature vector and determine their similarity with the query image. Once
all the images in the database have been processed, the results are sorted in decreasing order of their similarity.

**Step b** Parallel algorithm is obtained by distributing the set of images in the database over the processing nodes in the system.

**Step c** The set of images is further partitioned amongst the processing units within each node. The number of images to be processed on each FPGA and CPU was determined based on their processing capability. By exploiting fine-grain parallelism available in algorithm, FPGAs are able to process images at a faster rate than the CPUs and are assigned a larger subset of images.

**Step d** Using VHDL, we developed a hardware engine for FPGAs, which iterates over its assigned set of input images to compute their feature vectors and evaluate their similarity to the specified query image.

**Step e** We developed software code for processing a subset of images on the CPU, providing each FPGA with control signals to initiate processing and wait for completion, and transferring results from all processing units in the system to root node (node 0) at completion.

In addition to software parallelism described in the algorithm above, our hardware design for each FPGA instantiates multiple computational kernels that operate on five images in parallel. Figure 3-13 compares the execution time and speedup versus a serial software baseline for different implementations of a CBIR algorithm on the Mu cluster. Our experiments were conducted for an image size of $128 \times 128$, with the search database consisting of approximately 2800 images. Advantages of using FPGA devices are evident through faster execution times for RC-based implementations over software-only solutions. The FPGAs were able to process images at a much faster rate than the CPUs leading to over $30 \times$ speedup with four nodes when employing FPGA devices. Figure 3-13 also compares the performance of the algorithm implemented using SHMEM+ with solution implemented using a combination of Quadrics SHMEM
Figure 3-13. Performance comparison of different implementations of parallel CBIR application on Mu cluster. Software designs involve only CPU devices whereas RC designs involve both CPU and FPGAs on each node. Experiments were conducted for a search database consisting of 2800 images, each of size $128 \times 128$. A) Execution time of different designs. B) Speedup obtained by different designs when compared to a serial software baseline running on a single processor.

(cpu-only) library and platform-specific APIs for interaction with FPGAs. It is evident that the application developed using SHMEM+ incurs minimal overhead compared to traditional techniques of development where expert developers have access to vendor APIs.

More importantly, SHMEM+ provides application developers with a parallel-programming model that enables productive and portable design of scalable RC applications. There are a variety of factors that contribute towards improvement in developer productivity that are listed in Table 3-3. Applications developed without using SHMEM+ exhibit higher conceptual complexity. Application developers are often forced to employ multiple libraries with varying APIs to incorporate communication amongst a cluster of host CPUs and to facilitate coordination between a host CPU and its local FPGAs. In addition, any communication with an FPGA on a remote node will have to be explicitly routed by the developer, through the host CPU on that node. The processing on the host CPU of the remote node will have to be interrupted to service this communication request, which further increases the complexity of developing the
parallel program. With SHMEM+, a developer is oblivious to such details and exposed to a higher level of abstraction. Similarly, high-level SHMEM+ functions eliminate the need to explicitly perform various intermediate steps of communication, leading to a reduction in code size. Portability and scalability increase the application lifespan and reduce the recurring cost that would have been involved without the use of a library like SHMEM+.

A combination of the factors shown in Table 3-3 (and more) have a collective influence on the development time for an application. Although a comprehensive analysis of the impact of each of the factors listed here is beyond the scope of this research, to understand the productivity gains of SHMEM+, we present a brief discussion about the total development hours spent in application development by our team.

Table 3-3. Major factors that contribute towards increased developer productivity when using SHMEM+.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program complexity</td>
<td>High-level abstraction provided by SHMEM+ functions shields the application developer from various underlying details</td>
</tr>
<tr>
<td>Learning curve</td>
<td>Familiar APIs and programming model lead to reduced learning period when migrating to a new system</td>
</tr>
<tr>
<td>Source lines of code</td>
<td>Each SHMEM+ function can perform several intermediate steps of communication which eliminates the need for extra code and function calls</td>
</tr>
<tr>
<td>Application portability</td>
<td>Applications have longer life cycle as they can be executed on a variety of platforms</td>
</tr>
<tr>
<td>Scalability</td>
<td>Reduction in recurring developer effort to execute an application on systems of different sizes</td>
</tr>
</tbody>
</table>

Table 3-4 compares the development hours spent by our team during various stages of application design employing (a) traditional techniques of implementation and (b) SHMEM+ separately. Although the numbers cited are specific to our team personnel, we believe they are a fair estimate of improvements expected from SHMEM+. For the numbers cited in Table 3-4, we assume the developer has experience in parallel programming and in creating FPGA designs using VHDL. In addition, we assume
the developer is new to the RC system and hence has to undergo a learning process to familiarize with the platform, which is often the case when porting applications to a new system. The rows in Table 3-4 report the time spent (in terms of 8-hour work days) in various phases of application development which required significant amount of the time and effort. The time spent in each activity also includes the hours spent for debugging in that phase, wherever applicable. Since SHMEM+ does not modify the process of developing hardware cores for FPGAs, the time required for FPGA-core development (first two rows of the table) remains unaffected for both techniques, but has been included here for completeness. It should be noted that we employed HDL for developing our hardware-cores and further reductions in effort can be obtained by employing HLLs, if they are supported by the target platform. The time spent in parallel-software development includes the amount of time a developer spends in familiarizing with the platform-specific API, learning the SHMEM API, and finally designing the parallel application using these APIs. When using SHMEM+, a developer employs the SHMEM+ interface for interacting with FPGA memory and hence has to spend less time understanding only a subset of platform-specific APIs which are required for sending (or receiving) control signals (third row in the table). By contrast, the learning period involved for the SHMEM API remains unaffected as both the techniques expose the developer to a similar interface. Due to a higher level of abstraction provided by SHMEM+, a reduction in the time for designing the parallel application was observed as indicated by the fifth row in the table. As shown in the sixth row, an overall reduction in time and effort of about 30% was obtained for the total time spent in various phases of parallel-software development. Such an improvement could translate to significant savings in the development hours and money spent on the design of a complex application. For example, an application that required 10 weeks of development time could now be completed in just 7 weeks. Applications with more complex communication patterns are expected to have higher gains in productivity.
Table 3-4. Development hours spent in developing CBIR application. Time is reported in terms of 8-hour work days.

<table>
<thead>
<tr>
<th>Development phase</th>
<th>Traditional SHMEM</th>
<th>SHMEM+</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA-core development</td>
<td>Learning platform-specific wrappers 10 days</td>
<td>10 days</td>
</tr>
<tr>
<td></td>
<td>App-core design</td>
<td>15 days</td>
</tr>
<tr>
<td>Parallel-software development</td>
<td>Platform-specific API learning period 5 days</td>
<td>2 days</td>
</tr>
<tr>
<td></td>
<td>SHMEM API learning period 5 days</td>
<td>5 days</td>
</tr>
<tr>
<td></td>
<td>Parallel-application design 10 days</td>
<td>7 days</td>
</tr>
<tr>
<td>Total software development time</td>
<td>20 days</td>
<td>14 days</td>
</tr>
</tbody>
</table>

Since SHMEM+ applications do not employ any vendor-specific APIs for interaction with FPGAs, applications developed using SHMEM+ are highly portable. As long as the SHMEM+ library can be supported on a target RC system, any application designed with SHMEM+ can execute on it without requiring changes to the application source code. We evaluated the portability by migrating the CBIR application from our Mu cluster to Novo-G. This process did not require any modification to software and hardware source code. A simple re-compilation of the software was required to obtain an executable for the new system. It should be noted that for a system with a different FPGA board, some modifications will be needed for the hardware designs. Figure 3-14 shows the performance of the CBIR application scaling up to 16 nodes on Novo-G. We expanded our search database to include approximately 22,000 images on this larger system. The graphs in Figure 3-14 compare the performance of a design developed using SHMEM+ with a design based on traditional SHMEM. It is evident that designs developed using SHMEM+ continue to offer comparable performance to designs based on traditional SHMEM for larger system sizes. The minor variation in performance of the design created using SHMEM+ when compared to the one created with SHMEM is due to the difference in the communication mechanism used to gather the results on the root node in the last processing step. As with any form of high-level abstraction, a tradeoff exists between productivity and performance. For our application...
Figure 3-14. Performance comparison of different implementations of parallel CBIR application on Novo-G. Software designs involve only CPU devices whereas RC designs involve both CPU and FPGAs on each node. Experiments were conducted for a search database consisting of 22,000 images, each of size $128 \times 128$. A) Execution time of different designs. B) Speedup obtained by different designs when compared to a serial software baseline running on a single processor.

design using SHMEM+, the ability to use direct transfer to the remote FPGAs eliminates the opportunity to overlap the intermediate steps of communication for this design. However, SHMEM+ does not force developers to work at a particular level of abstraction. Instead it provides application developers with multiple options to meet the demands of the application. Although the performance penalty incurred by the designs created using SHMEM+ is minimal, we made a minor modification to the implementation of the quad-FPGA designs to eliminate this penalty as discussed in the following paragraphs.

Parallel algorithms employing multiple FPGAs on each node exhibit more complex communication patterns and often require increased developer effort to obtain an efficient implementation. SHMEM+ has the ability to support multiple FPGAs on each FPGA board using the same interface, which further simplifies the development process and yields additional improvement in productivity. Figure 3-15 compares the performance of different designs employing all of the four FPGAs on each node of Novo-G. The algorithm designed using SHMEM+ was modified slightly to optimize the collection of results at the end on root node (Node 0). Instead of the root node
Figure 3-15. Performance comparison of different implementations of parallel CBIR application on Novo-G. Software designs involve only CPU devices whereas RC designs involve both CPU and four FPGAs on each node. Experiments were conducted for a search database consisting of 22000 images each of size $128 \times 128$. Designs based on SHMEM+ continue to offer minimal overheads when compared to traditional techniques of application development. A) Execution time of different designs. B) Speedup obtained by different designs when compared to a serial software baseline running on a single processor.

using a “GET” routine to receive results from all the FPGAs, each processing node uses a “PUT” function to transfer the results from each of its local FPGA to the root node. The optimization allows the designs created with SHMEM+ to exhibit excellent scaling behavior and minimal overheads when compared to the designs created using a combination of CPU-only SHMEM and vendor APIs.

### 3.3.3 Case Study 2: Two-Dimensional FFT

As our next case study of parallel application, a two-dimensional FFT was chosen because of its emphasis on a more complex communication pattern and its relevance in a variety of application domains such as medical imaging systems, Synthetic Aperture RADAR (SAR) systems, and image processing [52–54]. The heavy computation demands of Fourier transform [55] pose tremendous pressure on the capabilities of computation platforms in most real-world applications, as a result of which several researchers have explored FPGA implementations for the same [56, 57]. A 2-D FFT
operation on an image is performed by decomposing it into a series of 1-D FFT over the rows of the image, followed by a series of Fourier transforms over the columns.

Our parallel implementation of 2-D FFT algorithm distributes rows of the input image across the computational nodes which perform a 1-D FFT over their assigned subset of rows as shown in Figure 3-16. A corner-turn (distributed transpose), which involves all-to-all communication between the processing nodes, is required to re-distribute the data across all the nodes. The nodes then compute 1-D FFT over the columns of the image. Another corner turn is required to re-organize the data and recover the transformed output image. Following the design flow described earlier we derive our implementation as follows:

**Step a** The serial algorithm computes the 2-D FFT of the image by performing a series of 1-D FFT over the rows followed by 1-D FFT over the columns of the image.

**Step b** Our parallel algorithm is obtained by using block decomposition to distribute a subset of rows and columns to be transformed on each node. An all-to-all communication is required to re-distribute the data between the two stages of 1-D FFTs.

**Step c** The FPGA on each node is able to perform 1-D FFT operations faster than the CPU and is hence assigned to transform the assigned set of rows and columns.
Since the CPUs are more efficient in re-organizing the data in their local memory than FPGAs, they are assigned to perform the corner turn.

**Step d** Using VHDL, we developed a hardware engine for FPGAs, to perform a series of 1-D FFT over its assigned set of input data. The FPGA waits for control signal from the CPU to begin processing and indicates completion of transforms through another control signal.

**Step e** Software code on the CPU is responsible for transferring the input data to the FPGAs and reading the transformed output once FPGA completes processing. The software code also performs an all-to-all communication to complete the corner-turn. In addition, it also provides the control signal required by the FPGAs.

Figure 3-17 compares the execution time and speedup versus a serial software baseline for different implementations of a parallel 2-D FFT algorithm on Novo-G. Our experiments were conducted for an 8k×8k image. Similar to our first case study, designs for 2-D FFT implemented using SHMEM+ yield performance which is comparable to designs implemented using a combination of traditional, CPU-only SHMEM library and platform-specific APIs. The minor difference in the performance of both of these designs is within reasonable limits of experimental error. A comparison of the hours spent in developing 2-D FFT algorithm using both of these techniques is presented in Table 3-5. For this case study, we assume a parallel application developer is familiar with the SHMEM API and does not have to spend any effort/time learning it. However, since platform-specific learning is a common occurrence each time the application is ported to a new platform, we retain the learning period for the platform-specific API. Our estimates indicate an improvement of approximately 25% in developer productivity. Portability experiments were also conducted for the second case study. Since most of the results and inferences from these experiments were consistent with the first case study, they are not repeated here.
Figure 3-17. Performance comparison of different implementations of parallel 2-D FFT algorithm on Novo-G. Software designs involve only CPU devices whereas RC designs involve both CPU and FPGAs on each node. Experiments were conducted for an 8k×8k image. A) Execution time of different designs. B) Speedup obtained by different designs when compared to serial software baseline running on a single processor.

Table 3-5. Development hours spent for developing Two-dimensional FFT application.

<table>
<thead>
<tr>
<th>Development phase</th>
<th>Traditional SHMEM</th>
<th>SHMEM+ SHMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA-core development</td>
<td>Learning platform-specific wrappers</td>
<td>15 days</td>
</tr>
<tr>
<td></td>
<td>App-core design</td>
<td>15 days</td>
</tr>
<tr>
<td>Parallel-software</td>
<td>Platform-specific API learning period</td>
<td>5 days</td>
</tr>
<tr>
<td>development</td>
<td>Parallel-application design</td>
<td>12 days</td>
</tr>
<tr>
<td></td>
<td>Total software development time</td>
<td>17 days</td>
</tr>
</tbody>
</table>

3.4 Conclusions

The lack of integrated, system-wide, parallel programming models has limited current RC applications to small system sizes. To realize the full potential of reconfigurable HPC systems, parallel-programming models and languages that are suited to such systems are critical yet lacking. In this phase of research, we presented a parallel-programming model and a communication library for scalable, heterogeneous, reconfigurable systems. The multilevel-PGAS model proposed in this chapter is able to capture key characteristics of RC systems, such as different levels of memory hierarchy and differences in the execution model of heterogeneous devices present in the system.
The existence of such a programming model will enable productive development of scalable, parallel applications for reconfigurable HPC systems.

Using the multilevel-PGAS programming model, we extend the existing SHMEM library to SHMEM+, the first known version of the library that enables designers to create scalable applications that execute over a mix of microprocessors and FPGAs. SHMEM+ offers developers of RC applications a high-level of abstraction that allows them to facilitate complex communication in application between heterogeneous devices, while providing high productivity and performance. Results from our experiments and case studies demonstrate that performance offered by SHMEM+ is comparable to an existing vendor-proprietary version of SHMEM. Our case studies showcase the simplified design process involved with SHMEM+ for developing scalable RC applications, which is very similar to traditional methods for development of parallel applications. More importantly, the higher level of abstraction provided by SHMEM+ leads to significant improvement in productivity without sacrificing performance significantly. Although it is difficult to quantify the productivity gains, our case studies demonstrate an average improvement in productivity of about 30%. In addition, by hiding the details of vendor-specific FPGA communication from developers, SHMEM+ creates highly portable applications.
CHAPTER 4
PERFORMANCE MODELING FOR MULTILEVEL COMMUNICATION
IN SHMEM+ (PHASE 2)

This chapter focuses on the Phase 2 of this research. Programming models and libraries for heterogeneous, parallel, and reconfigurable computing such as SHMEM+ are useful for supporting communication involving a diverse mix of processor devices. However, to evaluate the impact of communication on application performance and obtain optimal performance, a concrete understanding of the underlying communication infrastructure is often imperative. One important challenge when using libraries such as SHMEM+ is choosing an appropriate communication strategy. Variances include, who initiates the transfer, what functions are employed, what intermediate steps are involved, etc. Such factors can have a significant impact on the overall performance of an application. Therefore, it is critical for an application designer to understand the underlying communication infrastructure. Traditional HPC systems and applications assist designers in understanding and optimizing communication infrastructure by employing communication models that use a set of parameters to provide a high-level representation of communication, while hiding the unnecessary details. Although a few general-purpose communication models have been prevalent for traditional HPC systems, these models were not developed to target multilevel systems such as reconfigurable HPC systems. As a result, these models lack sufficient details for representing multilevel systems completely and accurately.

In this phase of research, we introduce a new multilevel communication model for representing various data transfers provided by the SHMEM+ library and for predicting their performance. The proposed model provides a system-level representation which integrates the effect of intermediate steps of communication present in a multilevel transfer. Additionally, the model allows for the capability of overlapping these intermediate steps which is critical for obtaining high-performance. The potential merits of the proposed model are showcased in this chapter through several examples.
Besides attaining higher performance from communication infrastructure, such a model can lead to considerable improvement in the productivity of application and system developers.

The remainder of this chapter is formatted as follows. An overview of the multilevel communication model is presented in Section 4.1. In Section 4.2, we showcase the effectiveness of the model in enabling early DSE for an application. Section 4.3 illustrates the potential use of the model to optimize the performance of the SHMEM+ library for a target RC system. In Section 4.4, we discuss a mechanism to augment SHMEM+ with the self-tuning capability using the model. Finally, Section 4.5 provides conclusions from this phase of research.

4.1 Overview of Multilevel Communication Model

The SHMEM+ library described in Chapter 3, provides a high-productivity environment for establishing communication in RC systems by abstracting the details of the underlying data transfers with a uniform, high-level interface. Figure 4-1 (reproduced from Figure 3-5) illustrates the different options for data transfers provided by the SHMEM+ library in a system with a CPU and an FPGA on each node (although the library can support any number and variety of devices). Some of the direct-transfer capabilities shown in the figure such as the ones labeled ‘x’ and ‘y’ were enabled by SHMEM+ and did not exist previously.

While having various options to transfer data from one device to another simplifies the development of parallel applications, users should understand the underlying mechanism and tradeoffs associated with such transfers in order to obtain optimal performance. Since only the CPU on each node can physically perform inter-node communication, some of the data transfers are internally achieved through multiple intermediate transfer steps. Consider a function that provides direct transfer between two remote FPGAs (labeled as ‘y’ in Figure 4-1). This function is internally performed by (a) transferring data to internal buffers on local CPU, (b) followed by a transfer to the...
Figure 4-1. Transfer capabilities supported by various communication routines in SHMEM+.

CPU on the remote node, and (c) finally writing data to the memory of the destination FPGA. However, such a function may serialize the aforementioned transfer steps which could have been explicitly parallelized by an application developer. For example, in an application where data is collected from several FPGA devices on a remote CPU, an application developer can explicitly parallelize the transfer of data from FPGAs to their local CPUs on all nodes and then transfer the data from the CPUs on each node to the remote CPU where data is collected. To enable better understanding of the underlying details of communication present in SHMEM+, we derive a multilevel communication model for the same.

The communication time for a point-to-point data transfer between any two devices of the system shown in Figure 4-1 can be represented by a generic model described as:

$$T_{comm} = f(T_{cpu\leftrightarrow cpu}) + f(T_{cpu\leftrightarrow fpga})$$  \hspace{1cm} (4–1)
Where,

\[ T_{\text{comm}} : \text{Total time for communication}, \]

\[ T_{\text{cpu}+\text{cpu}} : \text{Transfer time for two CPUs on remote nodes}, \]

\[ T_{\text{cpu}+\text{fpga}} : \text{Total time for various transfers between a CPU and its local FPGA}, \]

\[ f(T) = \text{serial time } T \text{ for communication} - \text{part of time } T \text{ hidden by overlap with other communication} \]

\[ = \text{non-overlapped part of time } T \]

The goal of the model in Equation 4–1 is to estimate the performance of a data-transfer routine by incorporating each intermediate step of the transfer separately. Depending upon the source and destination devices, a communication routine may require two transfers between a CPU and an FPGA (such as the case for transfer labeled as ‘y’ in Figure 4-1). \( T_{\text{cpu}+\text{fpga}} \) in Equation 4–1 represents the cumulative time of both transfers in such cases. The multilevel model does not estimate the performance of individual transfer steps or impose the use of a specific model for the same. Instead, it allows the use of different communication models for representing each component of the complete transfer. Such modeling is important because it is often difficult to describe the various intermediate steps of transfer using a single communication model. In our experiments we perform microbenchmarking to determine the time for intermediate steps of communication accurately. Many RC systems exhibit asymmetric performance for read (FPGA to CPU) and write (CPU to FPGA) operations. To account for this asymmetry in speed of read and write operations for an FPGA, the model can be rewritten as:

\[ T_{\text{comm}} = f(T_{\text{cpu}+\text{cpu}}) + f(T_{\text{cpu}+\text{fpga}}) + f(T_{\text{cpu}+\text{fpga}}) \quad (4–2) \]
Where,

\[ T_{\text{cpu} \leftarrow \text{fpga}} \]: Transfer time for read operation from FPGA,

\[ T_{\text{cpu} \rightarrow \text{fpga}} \]: Transfer time for write operation to FPGA

By overlapping various steps of communication, it is possible to reduce the overall time taken by the transfer. It should be noted that the model in Equation 4–2 (and Equation 4–1) only considers the effective (non-overlapped) time of each level of communication that influences the overall performance of communication. The provision for “effective” time is an important feature of the model because communication with the FPGA is invariably an expensive operation and is often overlapped with other steps of communication to improve overall performance. This feature of the model will be employed to optimize transfers between two FPGAs on remote nodes in Section 4.3. In addition to estimating communication time, the multilevel communication model can also be combined with existing approaches such as RAT [47] to estimate the execution time of the complete application.

In the following three sections, we discuss several benefits of the multilevel communication model through three use cases. First, the model can enable application developers to perform early design-space exploration of communication patterns in their applications before undertaking the laborious and expensive process of implementation, yielding improved performance and productivity. Second, the model can be employed by system developers to quickly optimize performance of data-transfer routines within SHMEM+ when being ported to a new platform. Third, the model can be used to augment SHMEM+ to automatically improve performance of data transfers by self-tuning its internal parameters to match platform capabilities which improves the portability of SHMEM+. The experiments for this phase of research will employ two different systems, Mu Cluster and Novo-G, which are described in Appendix A.
4.2 Early DSE in Applications

One advantage of the multilevel communication model is the enabling of early design-space exploration (DSE) that can improve performance and reduce development time for an application. By providing performance estimates, the model allows developers to make design decisions about the communication infrastructure before undertaking expensive implementation. We illustrate this capability using an example of a content-based image retrieval (CBIR) application, detailed in Appendix B.

The parallel algorithm employed in our experiments distributes the set of images to be searched over a set of nodes and allows multiple processing devices to evaluate these images simultaneously (more details about implementation of the algorithm can be found in Section 3.3.2. The steps involved in our parallel implementation based on SHMEM+ are as follows:

1. All nodes perform initialization using *shm_init*, which also configures local FPGAs on each node with the desired bitfile.

2. CPUs on all nodes read their subset of input images from a storage device (such as a local hard disk or a network storage device) along with the feature vector of the query image.

3. CPUs transfer the subset of images to their local FPGAs for hardware acceleration using the *shm_putmem* function.

4. CPUs initiate the execution on their local FPGAs through a “GO” signal. FPGAs on all nodes compute feature vectors and similarity measures for their subset of images in parallel.

5. FPGAs signal the completion of execution to local CPUs through a “DONE” signal. Once computation on CPUs and FPGAs on each node is complete, all nodes synchronize using *shm_barrier_all*.

6. Finally, similarity values from all of the FPGAs are gathered on the root node using one of the several approaches (determined by our analysis presented in Section 4.2.1). Results are then sorted in decreasing order of similarity.

By employing multiple FPGAs to accelerate the application, the computation time of the algorithm can be reduced significantly as shown by the performance of a typical
implementation in Figure 4-2A. The single-FPGA designs refer to the designs which employ multiple nodes with each node using a single FPGA. Similarly, quad-FPGA designs execute over multiple nodes with each node using four local FPGAs. As the application is scaled across more nodes, communication starts becoming a significant proportion of the total time. Figure 4-2B shows that the amount of time required to collect the results on the root node increases substantially with increasing number of processing nodes. The effect is more pronounced for the quad-FPGA design as the results need to be collected from more FPGAs (64 FPGAs for a system size of 16 nodes). As a result, the relative performance gain from employing more nodes (and FPGAs) starts decreasing.

To resolve this bottleneck, we need to understand the mechanism by which the application collects results on the root node. The gather operation in the application can be implemented in multiple ways such as:

**Approach 1:** By using a `shmem_getmem` function on the root node to receive the output data from all of the FPGAs involved in the application individually.
**Approach 2:** By using a `shmemb.putmem` function on every node to send the results computed by each FPGA to the root node individually.

**Approach 3:** By first collecting the results from the local FPGAs on every node’s CPU and then sending them to the root node collectively using `shmemb.putmem` on each node.

### 4.2.1 Performance Estimation

While performing the same gather operation, the aforementioned approaches can offer different performance and require different levels of developer effort. To evaluate the impact of these three approaches on the overall performance of the CBIR application, we estimate the performance of gather operation using each approach.

The difference in the performance becomes apparent by applying our multilevel communication model to the three approaches. The estimated time for performing the gather operation can be expressed using the model as:

\[
T_{\text{gather}} = \sum_{i=1}^{n} T_{\text{comm}_i} \tag{4-3}
\]

\[
T_{\text{comm}_i} = f(T_{\text{cpu} \rightarrow \text{fpga}}) + f(T_{\text{cpu} \leftarrow \text{cpu}}) + f(T_{\text{cpu} \leftarrow \text{fpga}}) = \text{non-overlapped part of } T_{\text{cpu} \rightarrow \text{fpga}} + \text{non-overlapped part of } T_{\text{cpu} \leftarrow \text{cpu}} + 0 \tag{4-4}
\]

Where,

- \(n\) : Number of nodes involved in the gather operation,
- \(T_{\text{gather}}\) : Total time for gather operation,
- \(T_{\text{comm}_i}\) : Communication time corresponding to \(i^{th}\) node

Since there are no write operations to an FPGA, the corresponding term becomes zero in Equation 4-4. While Approach 1 seems the most intuitive way to perform a gather, the process of invoking a get operation to receive data from all devices
individually serializes all of the transfers. As a result, none of the communication time can be overlapped. Therefore, the time to perform the gather operation for Approach 1 can be described as follows:

**Approach 1, Single-FPGA Design:**

\[ T_{\text{gather}} = (n - 1) \times (T_{\text{cpu-fpga}} + T_{\text{cpu-cpu}}) + T_{\text{cpu-fpga}} \quad (4-5) \]

**Approach 1, Quad-FPGA Design:**

\[ T_{\text{gather}} = 4 \times [(n - 1) \times (T_{\text{cpu-fpga}} + T_{\text{cpu-cpu}}) + T_{\text{cpu-fpga}}] \quad (4-6) \]

Note that the root node only requires a read operation from its local FPGA, which does not involve a network transaction (last term in Equations 4–5 and 4–6).

By contrast, Approach 3 requires more effort from a developer as it first collects the data from the FPGAs on the local CPU before sending the data across to the root node. However, this approach allows each node to overlap the read operation from its local FPGAs (which is usually an expensive operation) with the same operation on the other nodes. Therefore, the time to perform the gather operation for Approach 3 can be represented as follows:

**Approach 3, Single-FPGA Design:**

\[ T_{\text{gather}} = T_{\text{cpu-fpga}} + (n - 1) \times T_{\text{cpu-cpu}} \quad (4-7) \]

**Approach 3, Quad-FPGA Design:**

\[ T_{\text{gather}} = 4 \times T_{\text{cpu-fpga}} + (n - 1) \times T_{\text{cpu-cpu}} \quad (4-8) \]

Approach 2, which appears very similar to Approach 1 on the surface, behaves much like Approach 3. Allowing each node to put the data directly from the local FPGAs to the root node essentially overlaps the part of the transfer which reads the data from the local FPGAs (into a temporary buffer in SHMEM+) with the same operation on the
other nodes. As a result, Approach 2 yields performance that is comparable to the third approach while requiring comparatively lower developer effort. The performance of Approach 2 is identical to Approach 3 for single-FPGA designs. For quad-FPGA designs, Approach 2 requires four CPU-to-CPU transfers from each node to the root node (of a smaller data size) as opposed to a single transfer required by Approach 3.

The estimated time can be represented using the equations as follows:

**Approach 2, Single-FPGA Design:**

\[
T_{gather} = T_{cpu\rightarrow fpga} + (n - 1) \times T_{cpu\rightarrow cpu} \tag{4-9}
\]

**Approach 2, Quad-FPGA Design:**

\[
T_{gather} = 4 \times [T_{cpu\rightarrow fpga} + (n - 1) \times T_{cpu\rightarrow cpu}] \tag{4-10}
\]

In order to compute the estimates for the time required by the gather operation, we performed microbenchmarking to determine \(T_{cpu\rightarrow fpga}\) and \(T_{cpu\rightarrow cpu}\) for different data sizes. Table 4-1 lists the input parameters \((n, T_{cpu\rightarrow fpga}, T_{cpu\rightarrow cpu})\), the estimated gather times, and the corresponding times observed experimentally for performing the gather operation using single-FPGA designs on Novo-G. The results reported in the table correspond to a gather operation collecting 8MB data on the root node (to allow for longer gather times in our analysis, a data size much larger than required by our CBIR implementation was chosen). The estimates listed in the table for Approaches 1, 2 and 3 are computed using Equations 4–5, 4–9, and 4–7 respectively. For example when \(n = 16\) (last row of Table 4-1), the estimate for Approach 3 can be computed from Equation 4–7 as:

\[
T_{gather} = T_{cpu\rightarrow fpga} + (n - 1) \times T_{cpu\rightarrow cpu} = 1.51 + (16 - 1) \times 0.37 = 7.06 \text{ ms}
\]
Table 4-1. Observed time and estimated time to perform gather operation by single-FPGA designs using three different approaches on Novo-G. All times are reported in milliseconds. Total amount of data collected on the root node is 8MB in all cases.

<table>
<thead>
<tr>
<th>n</th>
<th>$T_{cpu\to fpga}$</th>
<th>$T_{cpu\to cpu}$</th>
<th>Approach 1</th>
<th>Approach 2</th>
<th>Approach 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.14</td>
<td>5.72</td>
<td>8.14</td>
<td>8.33</td>
<td>8.14</td>
</tr>
<tr>
<td>2</td>
<td>4.93</td>
<td>2.88</td>
<td>12.74</td>
<td>12.41</td>
<td>7.81</td>
</tr>
<tr>
<td>4</td>
<td>2.93</td>
<td>1.45</td>
<td>16.07</td>
<td>16.79</td>
<td>7.28</td>
</tr>
<tr>
<td>8</td>
<td>1.91</td>
<td>0.73</td>
<td>20.39</td>
<td>22.28</td>
<td>7.02</td>
</tr>
<tr>
<td>16</td>
<td>1.51</td>
<td>0.37</td>
<td>29.71</td>
<td>35.12</td>
<td>7.06</td>
</tr>
</tbody>
</table>

Table 4-2. Observed time and estimated time to perform gather operation for quad-FPGA designs on Novo-G using (a) Approaches 1 and 2. All times are reported in milliseconds. Total amount of data collected on the root node is 8MB in all cases.

<table>
<thead>
<tr>
<th>n</th>
<th>$T_{cpu\to fpga}$</th>
<th>$T_{cpu\to cpu}$</th>
<th>Approach 1</th>
<th>Approach 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Est.</td>
<td>Obs.</td>
</tr>
<tr>
<td>1</td>
<td>2.93</td>
<td>1.45</td>
<td>11.72</td>
<td>12.00</td>
</tr>
<tr>
<td>2</td>
<td>1.91</td>
<td>0.73</td>
<td>18.20</td>
<td>20.85</td>
</tr>
<tr>
<td>4</td>
<td>1.51</td>
<td>0.37</td>
<td>28.60</td>
<td>31.72</td>
</tr>
<tr>
<td>8</td>
<td>1.24</td>
<td>0.20</td>
<td>45.28</td>
<td>52.50</td>
</tr>
<tr>
<td>16</td>
<td>1.24</td>
<td>0.10</td>
<td>85.36</td>
<td>81.37</td>
</tr>
</tbody>
</table>

The estimated and observed time for the quad-FPGA designs on Novo-G are listed in Tables 4-2 and 4-3 along with the corresponding input parameters required for computing the estimates. In Approach 3, CPUs collect data from their local FPGAs before sending the collected data to the root node. As a result, $T_{cpu\to cpu}$ for Approach 3 is different than for Approaches 1 and 2. The results are tabulated in separate tables.

Table 4-3. Observed time and estimated time to perform gather operation for quad-FPGA designs on Novo-G using Approach 3. All times are reported in milliseconds. Total amount of data collected on the root node is 8MB in all cases.

<table>
<thead>
<tr>
<th>n</th>
<th>$T_{cpu\to fpga}$</th>
<th>$T_{cpu\to cpu}$</th>
<th>Approach 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Est.</td>
</tr>
<tr>
<td>1</td>
<td>2.93</td>
<td>5.72</td>
<td>11.72</td>
</tr>
<tr>
<td>2</td>
<td>1.91</td>
<td>2.88</td>
<td>10.52</td>
</tr>
<tr>
<td>4</td>
<td>1.51</td>
<td>1.45</td>
<td>10.39</td>
</tr>
<tr>
<td>8</td>
<td>1.24</td>
<td>0.73</td>
<td>10.07</td>
</tr>
<tr>
<td>16</td>
<td>1.24</td>
<td>0.37</td>
<td>10.51</td>
</tr>
</tbody>
</table>
(Tables 4-2 and 4-3). The estimates listed in the tables for Approaches 1, 2 and 3 are computed using Equations 4–6, 4–10 and 4–8 respectively. For example when \( n = 8 \) (in Table 4-2), the estimate for Approach 2 can be computed from Equation 4–10 as:

\[
T_{\text{gather}} = 4 \times [T_{\text{cpu} \leftrightarrow \text{fpga}} + (n - 1) \times T_{\text{cpu} \leftrightarrow \text{cpu}}]
\]

\[
= 4 \times [1.24 + (8 - 1) \times 0.20] = 10.56 \text{ ms}
\]

Figure 4-3 shows the estimated performance of the gather operation using the different approaches for (a) single-FPGA designs and (b) quad-FPGA designs. While the time required for gathering the data increases significantly for the first approach, it stays relatively constant for the second and third approaches. The effect is more pronounced for the quad-FPGA designs because there are more processing devices involved in the gather operation.

Figure 4-3. Estimated performance of gather operation on Novo-G using three different approaches. A) For single-FPGA designs. B) For quad-FPGA designs. Total amount of data collected on the root node is 8MB in all cases.

### 4.2.2 Experimental Results

The estimates computed by use of the proposed model were verified by comparing them with experimental performance observed for the three approaches (reported in Tables 4-1, 4-2 and 4-3). The average error between the estimates and experimental results was 9.4% (which is considered reasonably accurate given the focus on early DSE prior to any implementation). A few cases experienced higher errors (between
20-25\%), such as the quad-FPGA design over 16 nodes using Approach 2. With more processing devices, the time for intermediate steps of transfer became considerably small such that large variations were observed in experimental data and in the results of microbenchmarking (which form the inputs for the estimation model). As a result, even a minor deviation in measurements manifested as a larger relative error. Although not ideal, the absolute error had little impact on our design decisions as the trend observed between the experimental data and the estimates based on our model were consistent.

From the behavior of the three approaches, it appears that a CBIR application employing Approach 1 would lead to significant performance degradation as the system size increases, whereas with Approaches 2 and 3, the application would continue to offer satisfactory performance. The choice between using either of the two approaches (2 or 3) would be based on the level of programming complexity and the effort required from the developer. Approach 2 appears promising as it offers performance comparable to Approach 3 while requiring lower developer effort. To observe the effect of the three approaches on overall application performance, we developed a full CBIR implementation using the three approaches. Figure 4-4 shows the performance obtained for quad-FPGA designs of CBIR application on Novo-G using the three approaches. The results agree with the behavior predicted based
on the estimates from our model. Approaches 2 and 3 offer similar performance, better than Approach 1. By using the proposed model to quickly perform early DSE of communication patterns, we were able to improve performance of the application by approximately 42%. This exercise showcased the use of our multilevel communication model as a tool for enabling DSE in the early phase of application development. Such a tool can eliminate several iterations of expensive design cycle and help in improving developer productivity.

### 4.3 Optimizing SHMEM+ Functions

The multilevel communication model enables system developers to quickly optimize the performance of a communication library such as SHMEM+ when porting it to a new system. Consider the case of data transfers between two remote FPGAs. As shown in Figure 4-5, such a transfer can be performed by (1) reading the data from the source FPGA to its local CPU; (2) transferring the data from the local CPU to the CPU on the remote node; (3) and finally, writing the data from the CPU on the remote node to the destination FPGA.

Instead of performing Steps 1, 2 and 3 sequentially, an efficient implementation for the transfer shown in Figure 4-5 may overlap the intermediate steps of such a transfer. For example, Step 3 can be overlapped with Steps 1 and 2 collectively. In order to overlap various steps of a transfer, the data needs to be divided into smaller packets.
The size of the data packet can have a significant impact on overall performance of the transfer. A small packet size would lead to low performance for intermediate steps of the transfer, while a large packet size would limit the amount of communication that can be efficiently overlapped. Determining an appropriate packet size can be a laborious task and may occasionally require development of a testbench and numerous executions of the testbench with various packet sizes until satisfactory performance is obtained.

4.3.1 Performance Estimation

The proposed model can assist system developers in determining an appropriate packet size without requiring any test code. To estimate the performance of the transfer for a particular packet size, our model can be employed as follows:

Let,

\[ N : \text{Number of packets} \]
\[ D : \text{Size of data being transferred in bytes} \]
\[ P : \text{Size of data packets in bytes} \]
\[ T(L) : \text{Time } T \text{ for transferring } L \text{ bytes} \]

then,

\[ N = \left\lfloor \frac{D}{P} \right\rfloor \quad (4-11) \]

\[ T_{\text{step1}} = \begin{cases} 
T_{\text{cpu-fpga}}(P), & \text{when } D > P \\
T_{\text{cpu-fpga}}(D), & \text{when } D < P 
\end{cases} \quad (4-12) \]

\[ T_{\text{step2}} = \begin{cases} 
T_{\text{cpu-cpu}}(P), & \text{when } D > P \\
T_{\text{cpu-cpu}}(D), & \text{when } D < P 
\end{cases} \quad (4-13) \]

\[ T_{\text{step3}} = \begin{cases} 
T_{\text{cpu-fpga}}(P), & \text{when } D > P \\
T_{\text{cpu-fpga}}(D), & \text{when } D < P 
\end{cases} \quad (4-14) \]

Here, \( T_{\text{step1}}, T_{\text{step2}} \) and \( T_{\text{step3}} \) represent the time to transfer a single packet for the corresponding steps in Figure 4-5. When \( D < P \), the entire data is sent in a single
packet. Whereas for $D > P$, the transfer is broken into packets of size $P$. By overlapping Step 3 with Steps 1 and 2 collectively, the overall time of transfer can be described as:

$$T_{comm} = f(T_{cpu \leftarrow fpga} + T_{cpu \leftrightarrow cpu}) + f(T_{cpu \rightarrow fpga})$$

$$= \text{non-overlapped part of } (T_{cpu \leftarrow fpga} + T_{cpu \leftrightarrow cpu})$$

$$+ \text{non-overlapped part of } T_{cpu \rightarrow fpga}$$

$$= (T_{step1} + T_{step2}) + (N - 1) \times$$

$$\max((T_{step1} + T_{step2}), T_{step3}) + T_{step3} \quad (4-15)$$

Since Steps 1 and 2 are collectively overlapped with Step 3, only the time for the greater of the two components affects the overall time of transfer. In addition, Steps 1 and 2 for the first packet and Step 3 for the last packet cannot be overlapped and are included separately. By applying Equation 4–15 for different packet-sizes, overall time of the transfer can be estimated for various packet sizes. Table 4-4 presents the estimates computed for transfer time and bandwidth when $P = 2$MB and $P = 512$KB. The input parameters ($D$, $N$, $T_{step1}$, $T_{step2}$ and $T_{step3}$) required to compute the estimates are also listed in the table. The values reported in the table for $T_{step1}$, $T_{step2}$ and $T_{step3}$ were determined empirically using microbenchmarks on Novo-G. For example, estimates for $P = 2$MB and $D = 16$MB can be computed using Equation 4–15 as:

$$T_{comm} = (T_{step1} + T_{step2}) + (N - 1) \times$$

$$\max((T_{step1} + T_{step2}), T_{step3}) + T_{step3}$$

$$= (3 + 1.45) + (8 - 1) \times \max((3 + 1.45), 8) + 8$$

$$= 68.45 \text{ ms}$$

$$\frac{D}{T_{comm}} = \frac{16 \times 1024 \times 1024}{68.45 \times 10^{-3}} \times 10^{-6} = 245.1 \text{ MB/s}$$
Table 4-4. Estimating performance of packetized transfers between two remote FPGAs on Novo-G for $P = 2\text{MB}$ and $512\text{KB}$.

<table>
<thead>
<tr>
<th>$D$ (Bytes)</th>
<th>$N$</th>
<th>$T_{\text{step1}}$ (ms)</th>
<th>$T_{\text{step2}}$ (ms)</th>
<th>$T_{\text{step3}}$ (ms)</th>
<th>$T_{\text{comm}}$ (ms)</th>
<th>$B/W$ (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512K</td>
<td>1</td>
<td>1.17</td>
<td>0.37</td>
<td>3.01</td>
<td>4.55</td>
<td>115.2</td>
</tr>
<tr>
<td>1M</td>
<td>1</td>
<td>2.00</td>
<td>0.73</td>
<td>4.99</td>
<td>7.72</td>
<td>135.8</td>
</tr>
<tr>
<td>2M</td>
<td>1</td>
<td>3.00</td>
<td>1.45</td>
<td>8.00</td>
<td>12.45</td>
<td>168.4</td>
</tr>
<tr>
<td>4M</td>
<td>2</td>
<td>3.00</td>
<td>1.45</td>
<td>8.00</td>
<td>20.45</td>
<td>205.1</td>
</tr>
<tr>
<td>8M</td>
<td>4</td>
<td>3.00</td>
<td>1.45</td>
<td>8.00</td>
<td>36.45</td>
<td>230.1</td>
</tr>
<tr>
<td>16M</td>
<td>8</td>
<td>3.00</td>
<td>1.45</td>
<td>8.00</td>
<td>68.45</td>
<td>245.1</td>
</tr>
<tr>
<td>32M</td>
<td>16</td>
<td>3.00</td>
<td>1.45</td>
<td>8.00</td>
<td>132.45</td>
<td>253.3</td>
</tr>
</tbody>
</table>

For $P = 512\text{KB}$

<table>
<thead>
<tr>
<th>$D$ (Bytes)</th>
<th>$N$</th>
<th>$T_{\text{step1}}$ (ms)</th>
<th>$T_{\text{step2}}$ (ms)</th>
<th>$T_{\text{step3}}$ (ms)</th>
<th>$T_{\text{comm}}$ (ms)</th>
<th>$B/W$ (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512K</td>
<td>1</td>
<td>1.17</td>
<td>0.37</td>
<td>3.01</td>
<td>4.55</td>
<td>115.2</td>
</tr>
<tr>
<td>1M</td>
<td>2</td>
<td>1.17</td>
<td>0.37</td>
<td>3.01</td>
<td>7.56</td>
<td>138.7</td>
</tr>
<tr>
<td>2M</td>
<td>4</td>
<td>1.17</td>
<td>0.37</td>
<td>3.01</td>
<td>13.58</td>
<td>154.4</td>
</tr>
<tr>
<td>4M</td>
<td>8</td>
<td>1.17</td>
<td>0.37</td>
<td>3.01</td>
<td>25.62</td>
<td>163.7</td>
</tr>
<tr>
<td>8M</td>
<td>16</td>
<td>1.17</td>
<td>0.37</td>
<td>3.01</td>
<td>49.70</td>
<td>168.8</td>
</tr>
<tr>
<td>16M</td>
<td>32</td>
<td>1.17</td>
<td>0.37</td>
<td>3.01</td>
<td>97.86</td>
<td>171.4</td>
</tr>
<tr>
<td>32M</td>
<td>64</td>
<td>1.17</td>
<td>0.37</td>
<td>3.01</td>
<td>194.18</td>
<td>172.8</td>
</tr>
</tbody>
</table>

Following a similar approach, performance of the packetized transfers can be estimated for other packet sizes. Figure 4-6A shows the estimated bandwidth for a variety of packet sizes on Novo-G. Based on the estimates, the packet size which offers the best performance can be determined. The results indicate that for $P = 512\text{KB}$ the overall bandwidth is lower than the non-packetized baseline. The packet size of 2MB was found to offer satisfactory performance over a large range of data sizes under consideration. Depending on the size of the data transfer involved, the improvement obtained ranged from 11% to 24% (for the range of data sizes under consideration).

4.3.2 Experimental Results

Figure 4-6B shows the bandwidth of packetized transfers between two remote FPGAs for various packet sizes, recorded experimentally using a testbench on Novo-G. The trend observed from the experimental data concurs with the estimates generated using our model. While it took our team two to three hours to run microbenchmarks and
compute the estimates (from the input parameters, as shown in Table 4-4) to determine the best packet size, the process of developing a testbench and conducting several trials to determine the best packet size experimentally took in the order of two days. The system developers can benefit greatly from reduction in their time and effort using modeling and estimation to perform such optimizations on a target system.

The errors observed between the estimated and observed bandwidth are reported in Table 4-5. The average of errors reported in the table is under 6%. A few cases (especially transfers involving small data sizes) experienced higher errors. For smaller data sizes, the time for intermediate steps of transfers (which forms the input to our model) became considerably small. As a result, even a minor variation in microbenchmarking results manifested as a larger relative error in the estimates computed using the model. Nevertheless, the behavior of estimated performance concurs with the observed performance, which helped us in determining the best packet size. A similar methodology can also be employed to optimize other data transfers in the SHMEM+ library.

### 4.4 Self-Tuning SHMEM+ Library

Depending on the capabilities of the interconnect technology and the I/O bus, a communication library such as SHMEM+ may have different ranges for optimal operation on different systems. For example, a certain packet size for transfers discussed in
Table 4-5. Relative error between estimated bandwidth and observed bandwidth of transfers between remote FPGAs.

<table>
<thead>
<tr>
<th>Data size (Bytes)</th>
<th>Non-packetized</th>
<th>512KB packet</th>
<th>2MB packet</th>
<th>8MB packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>512K</td>
<td>15.5%</td>
<td>18.3%</td>
<td>14.7%</td>
<td>18.8%</td>
</tr>
<tr>
<td>1M</td>
<td>3.3%</td>
<td>5.6%</td>
<td>3.5%</td>
<td>4.1%</td>
</tr>
<tr>
<td>2M</td>
<td>9.3%</td>
<td>8.4%</td>
<td>8.9%</td>
<td>7.9%</td>
</tr>
<tr>
<td>4M</td>
<td>5.3%</td>
<td>2.5%</td>
<td>7.1%</td>
<td>1.5%</td>
</tr>
<tr>
<td>8M</td>
<td>2.2%</td>
<td>1.1%</td>
<td>3.6%</td>
<td>2.5%</td>
</tr>
<tr>
<td>16M</td>
<td>1.4%</td>
<td>0.4%</td>
<td>2.2%</td>
<td>2.5%</td>
</tr>
<tr>
<td>32M</td>
<td>0.5%</td>
<td>-1.0%</td>
<td>1.4%</td>
<td>1.0%</td>
</tr>
</tbody>
</table>

Section 4.3 may lead to satisfactory performance on some systems while yielding sub-optimal performance on others. In addition, the variation in the system load can also change the operational characteristics of a system. Allowing a communication library such as SHMEM+ to automatically tune itself to the capabilities of a target system can increase its usefulness in achieving portable performance.

The methodology described in Section 4.3 can be extended to automate the optimization process. The multilevel communication model can be embedded in the SHMEM+ library to allow it to automatically tune its performance on a target system, hence preserving performance while improving portability. Such a capability also enables applications to dynamically tune the SHMEM+ library according to the system load at several points during an application’s execution.

To incorporate this feature in SHMEM+, we will modify the initialization function (shmem_init) in the SHMEM+ library to invoke a “self-optimization” function. This function will execute a series of microbenchmarks to determine the input parameters for estimation. The parameters can be the transmission times for different data sizes over the network and the I/O bus, or a set of model parameters (e.g. LogGP, PlogP) which can then be used to estimate the transmission times over these interconnects. The self-optimization function will then use the multilevel communication model to determine the packet sizes required to obtain optimal performance for various SHMEM+ routines.
The information generated by the self-optimization function will also be stored to a file, from which it can be later retrieved to eliminate the need for performing these tests for every execution of an application. However, if the system performance varies over a period of time, an application may choose to invoke the optimization function to re-tune the performance of the library.

Figure 4-7. Bandwidth of transfers between remote FPGAs obtained by baseline SHMEM+ library and self-tuned version of SHMEM+. A) On Novo-G employing slower DMA engine. B) On Novo-G employing faster DMA engine. C) On Mu Cluster.

4.4.1 Experimental Results

We augmented the SHMEM+ library with self-tuning capability and evaluated its effectiveness on several different system configurations. Figure 4-7 compares the bandwidth observed for transfers between remote FPGAs using the baseline SHMEM+ (without self-tuning) with the bandwidth observed for those transfers by a self-tuned SHMEM+. The Novo-G and Mu cluster differ in the interconnect technology used by the two systems. We also added diversity in the capabilities of the I/O bus by employing two different versions of the DMA engine (with varying transfer speeds) provided by the FPGA-board vendor on the Novo-G system. Figure 4-7 shows that the self-tuned SHMEM+ library was able to automatically determine the appropriate packet size for transfers on different systems and offer significantly improved performance. Figure 4-8 lists the packet size that were determined to offer best performance by the self-optimization function for transfers between remote FPGAs. Table 4-6 highlights the
improvement in bandwidth obtained by the routines in the self-tuned SHMEM+ library over the baseline version. More importantly, the self-tuning capability improves the portability of SHMEM+.

![Table Image]

Table 4-6. Performance improvement for transfers between remote FPGAs obtained by self-tuned SHMEM+ library vs. baseline.

<table>
<thead>
<tr>
<th>Data size (Bytes)</th>
<th>Novo-G (slower DMA engine)</th>
<th>Novo-G (faster DMA engine)</th>
<th>Mu Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>4M</td>
<td>6.9%</td>
<td>5.5%</td>
<td>104.3%</td>
</tr>
<tr>
<td>8M</td>
<td>14.5%</td>
<td>22.0%</td>
<td>74.6%</td>
</tr>
<tr>
<td>16M</td>
<td>18.9%</td>
<td>23.7%</td>
<td>59.3%</td>
</tr>
<tr>
<td>32M</td>
<td>25.1%</td>
<td>27.7%</td>
<td>52.6%</td>
</tr>
</tbody>
</table>

4.5 Conclusions

Programming models and libraries for heterogeneous, parallel, and reconfigurable computing such as SHMEM+ provide a useful mechanism for establishing communication in such systems. However, a communication model is an important tool for optimizing the performance of an application and developing a concrete understanding of the underlying communication infrastructure. The new multilevel communication model proposed in this phase of research provides a system-level representation to integrate the effect of multiple levels of communication that are routinely encountered in scalable RC systems. The model has provisions for accurately representing the opportunities for overlapping intermediate steps of communication, which is critical for obtaining high performance.
In this chapter, we demonstrated the benefits of our model as a tool for performing early DSE to optimize the communication infrastructure yielding improved performance and productivity. An improvement of 42% was observed in the overall performance of our CBIR application. The communication model also enabled us to simplify the process of optimizing the transfer functions in SHMEM+ for a target system. Furthermore, the model allowed us to augment the SHMEM+ library to automatically tune its performance based on the capabilities of a system, hence making SHMEM+ more portable. Improvement in performance of up to 100% was obtained in certain cases by self-tuned routines in SHMEM+. 
In this chapter, a framework for establishing communication between various devices in a heterogeneous system is presented, the System Coordination Framework (SCF). SCF consists of a library of message-passing coordination primitives suitable for potentially any language or device, a framework that allows an application to be expressed as a static task-graph and each task of an application to be defined in potentially any language, and a set of tools that can create customized communication methods for a given system architecture based on the mapping of tasks to devices. One key advantage of SCF is that many low-level architectural details are hidden from the application developers, allowing them to simply define each task in a language of their choice, while specifying coordination between tasks using message-passing primitives. By hiding low-level architectural details from the application developer, SCF can improve application development productivity and offer rapid exploration of different task-to-device mappings without modifications to task definition code which often yields improved system performance. Furthermore, since applications employ a uniform, high-level interface to define communication, SCF helps in improving application portability. Our preliminary results show marked improvement in productivity and portability while incurring minimal performance overheads.

The remainder of this chapter is formatted as follows. A detailed overview of the design of SCF is presented in Section 5.1. In Section 5.2, we describe the tools present in the SCF ecosystem. Section 5.3 evaluates our implementation of SCF on two different platforms and presents results from two case studies. Finally, Section 5.4 provides a summary of conclusions.

5.1 Overview of SCF

Figure 5-1 presents an overview of the application design methodology with SCF. This description illustrates a bottom-up approach where an application is built by
connecting constituent tasks. However, it can easily be adapted to a top-down design flow by switching the order of the first two stages involved in the process.

The designer begins by defining individual tasks as shown in Figure 5-1A, using potentially any language, compiler, or synthesis tool per task, thus enabling tool-chain interoperability by allowing different tasks to be written in different languages. Such behavior is critical for heterogeneous systems, where each device may require a different, specialized language [58, 59]. Existing IP cores could also be used as task definitions, with only minor extensions to integrate with the SCF send and receive mechanisms.

![Figure 5-1. Application design philosophy using SCF. A) Designer defines tasks independently from task graph and device mapping. B) Creates a task graph by simply interconnecting tasks without changing individual task definitions. C) And then, maps tasks onto devices. D) Using automated communication synthesis to create efficient coordination mechanisms based on the mapping.](image)

An important part of each task definition is defining the input and output to other tasks, which can have a large effect on designer productivity. Without SCF, defining task interactions is dependent on the source device and the receiver device, often requiring
different device/platform-specific APIs for different mappings. In many cases, changing a mapping requires time-consuming modifications to task definition code. With SCF, a designer specifies all task interactions using a message-passing library called the SCF library. When defining task interactions with this library, a designer can be completely unaware of the source or destination device—a key advantage that enables task portability across multiple devices, task reuse in different applications, and transparency of low-level, device-specific communication details. Furthermore, because SCF is aware of the source and destination device for all inter-task data transfers, SCF can potentially allow automatic conversion between data formats (for example, conversion of endianness of data), which further improves designer productivity.

After defining individual tasks, the designer then builds the complete application by simply connecting inputs and outputs of the various tasks to form a task graph, as shown in Figure 5-1B. Note that this process does not require any changes to the definition of the individual tasks. Currently, SCF can map a task to any device, provided that the code for the task can be implemented on that device. In the worst case, a designer would have to provide task code for each device under consideration. However, improvements in high-level synthesis tools (e.g. C to VHDL) can provide the capability of converting task-definition code of one device to another. SCF does not attempt to automate this conversion of code specified using a particular language or vendor tools to another.

Once the task graph is defined, the designer maps individual tasks to specific devices in the system architecture (as shown in Figure 5-1C), again without making any modifications to the task definition code or the task graph of the application. SCF can be used with systems from domains ranging from HPC to embedded computing. Figure 5-1C shows several examples of system architectures where SCF can be employed such as a cluster of CPU nodes (HPC system) optionally equipped with accelerators, a
combination of FPGAs and embedded processor (high-performance embedded system), or a system based on a stand-alone FPGA (embedded system).

As shown in Figure 5-1D, SCF uses the specified mapping to automatically implement all data transfers in the task graph using the specific communication capabilities of the system. For example, in a platform comprised of a host microprocessor and an accelerator board with multiple FPGAs (such as our experimental system in Section 5.3.1), SCF could implement communication between any FPGA and the host CPU using on-board memory, whereas communication between different FPGAs could be implemented using physical wires on the board, while hiding all implementation details from the designer. Furthermore, knowledge of task interaction and their specific mapping onto system resources prior to compilation allows SCF to perform optimizations for reconfigurable devices, such as FPGAs, that are not possible in approaches that use dynamic routing to enable arbitrary communication between tasks, such as packet-switched, NoC design [35]. Such optimizations can improve application performance and reduce area requirements.

The transparency provided by SCF enables a designer to rapidly explore mappings of tasks to different devices, which is often critical for meeting design constraints. Although much previous work has focused on automatic design-space exploration [60, 61], such exploration is still largely a manual process for heterogeneous systems. It should be noted that SCF itself does not provide any automated design-space exploration. Rather, this work abstracts away from the designer the details of coordination between various tasks mapped over heterogeneous resources. Designers may still perform optimization and design-space exploration by means of external tools, and SCF provides an easy-to-use entry point for implementing those designs. The following subsections explain SCF in more detail and discuss the programming and communication model adopted by this framework.
5.1.1 Programming Model

There are four terms that define the SCF programming model. A *task* in SCF is the finest, indivisible unit of computation that can be mapped onto a device. Each SCF task communicates data with other tasks using message-passing primitives. A *task graph* is a directed graph formed by connecting the *communication interfaces* (i.e., inputs and outputs) of various tasks together using *edges*. The edges in SCF task graphs do not specify any precedence or order of execution, but represent the interaction between two tasks. The SCF programming model places no restriction on where or when the communication primitives are used inside tasks, which allows representation of communication between tasks using different underlying models such as dataflow graphs (synchronous and asynchronous), communicating sequential processes, and generic message-passing models. *Task-definition code* implements the computational portion of a task using code in potentially any language such as C++, VHDL, Impulse C™, CUDA™, OpenCL™, etc., while using SCF library primitives for specifying communication. As long as message-passing constructs of the library can be specified in a language, SCF can support task definitions using that language. Finally, *mapping* in SCF defines how tasks are mapped onto specific devices and system resources.

5.1.2 Architectural Model

To enable coordination between heterogeneous devices on as many systems as possible, SCF uses a hierarchical architecture model that captures structures common to heterogeneous systems, while abstracting away details that designers may not require. The SCF architecture model consists of three levels of abstraction: devices, platforms, and systems. All SCF tasks execute on SCF devices, which are the finest-grained computational resources of a given system. Every SCF device is part of an SCF platform, which is a subsystem containing a set of SCF devices interconnected by a specific communication topology. Each SCF platform is part of
Figure 5-2. Architectural model of an example system.

an SCF system, at the top of the hierarchy, which connects a set of platforms using a specific communication topology.

Figure 5-2 illustrates how SCF architecture models can be used to represent common systems. The figure shows a representation of a cluster of nodes connected over Ethernet (or any network technology), where some nodes consist of a CPU (D1 devices in Figure 5-2), or a CPU and an accelerator board, each having multiple FPGAs (D2 and D3 devices). This architecture is represented as an SCF model in the following way. The FPGAs and CPUs are SCF devices, each node collectively acts as an SCF platform, and the cluster of all nodes forms an SCF system. The different levels of abstraction are not necessarily mutually exclusive; a single physical device could be a SCF device, platform, and system. For example, platform P1, which is comprised of a single device, is both an SCF device and platform. Such flexibility allows the SCF architectural model to represent diverse systems.

5.1.3 Communication Model

One key advantage to the multiple levels of abstraction in the architecture model is that communication can be made transparent to the designer by distributing communication responsibilities throughout the system. Furthermore, such transparency eases conversion of existing devices and platforms into SCF systems. SCF-compliant devices are capable of handling all device-level communication, which we define to
be communication between tasks mapped onto the same SCF device. For example, a microprocessor is SCF-compliant if it is capable of supporting communication between multiple tasks mapped onto it. The physical implementation does not affect SCF compliance and could vary for different devices; it could be achieved via a message-passing library or message queues supported by operating systems. Similarly, SCF-compliant platforms provide communication routines that are responsible for all data transfers when the receiver is implemented on a different device in the same platform. SCF platforms are capable of transferring messages to the appropriate SCF device in target platform. Again, the physical implementation of such communication could vary for different platforms. For example, in an SCF platform containing multiple processor cores, messages could be passed through FIFOs in shared memory whereas, for an SCF platform comprised of multiple FPGAs, streaming data transfer could be achieved through physical wires. Alternatively, SCF resorts to the system-level communication routines if the receiver task is mapped on a different platform.

Table 5-1. SCF library primitives.

<table>
<thead>
<tr>
<th>Function</th>
<th>API</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>SCF_Init</td>
<td>Setup</td>
</tr>
<tr>
<td>Termination</td>
<td>SCF_Finalize</td>
<td>Setup</td>
</tr>
<tr>
<td>Send</td>
<td>SCF_Send</td>
<td>Point-to-Point</td>
</tr>
<tr>
<td>Recv</td>
<td>SCF_Recv</td>
<td>Point-to-Point</td>
</tr>
</tbody>
</table>

Despite the existence of distinct communication levels, SCF hides these levels from designers, who are instead exposed to a single communication API. Without SCF, a designer would have to go through an extensive process of using ad-hoc methods of establishing communication between any two devices on which the communicating tasks are mapped. Furthermore, the designer would have to re-establish the communication mechanism following any changes in the resource mapping. With SCF, a designer simply specifies the input and output of each task while relying on the tools in our framework (discussed in Section 5.2) to implement the communication.
Communication between SCF tasks uses the message-passing model, which is commonly employed in the parallel computing community [62]. With message passing, all communication in a task-definition code is specified explicitly as function calls that send or receive data and require both participating tasks to execute their respective functions to complete a data transfer. Such a communication model is generic enough to interface with potentially any programming model associated with any device. Table 5-1 presents the coordination primitives currently supported by the SCF library. The setup routine, `SCF_Init`, performs initialization operations and allocation of resources for all levels of communication and underlying libraries. `SCF_Finalize` performs complementary termination functions such as de-allocation of the resources which were setup during initialization. `SCF_Send` and `SCF_Recv` are communication calls (which could be synchronous or asynchronous) that provide data transfer between tasks. In Section 5.3, we further describe how this simple set of communication routines is adapted to the programming models of the devices present in our experimental system.

Note that this API is intentionally much simpler than other message-passing libraries (e.g., MPI), which typically contain constructs for scatter, gather, broadcast, etc. With SCF, a designer defining a task is not required to know if the inputs and outputs of the task are used for point-to-point communication or collective communication. The designer simply defines the inputs and outputs of the task. Then, at the task-graph level, collective communication operations can be specified via specialized edges between tasks. By separating specialized communication from task definitions, SCF increases portability of task-definition codes to different applications. The simple language that we developed (described in Section 5.2.1) for describing task graphs currently supports unicast edges (single source, single receiver) for point-to-point communication and multicast edges (single source and multiple receivers, or multiple sources and a single receiver) for broadcast, scatter, and gather operations.
5.2 SCF Ecosystem

The SCF ecosystem refers to the set of tools (commercial tools as well as tools developed as a part of this phase of research) which an application developer will typically employ for developing applications with SCF. In the following subsections, we describe the tools that we developed to support SCF on our experimental systems, and their interaction with commercial tools for developing applications in SCF.

5.2.1 SCF Task-Graph Language and Mapping

To evaluate SCF, we created a set of tools in the Eclipse environment [63, 64]. These tools include an SCF task-graph editor that allows developers to specify applications as task graphs and specify the mapping of tasks onto system resources, as well as a communication synthesizer that uses the information specified to generate appropriate communication routines.

We developed a simple language for specification of application task graphs in the task-graph editor. The primary constructs in this language are task, edge, and loop. Every task of an application that implements a part of its computation is represented in a graph using a task construct. The task graph specification language requires that tasks have unique names and well-defined communication interfaces for any communication going to (and from) a task. The communication interfaces are specified using the prefix input or output before the interface name of the respective tasks. Figure 5-3 shows an example of a task graph (of target tracking application detailed in Section 5.3.3) specified using the SCF task-graph language. There are four tasks in the example graph (T1, T2, T3, and T4). Task T1 has three output interfaces (out2, out3, and out4) and three input interfaces (in2, in3, and in4).

Each communication interface of a task is connected to an interface of another task using an edge. The SCF task-graph language currently supports unicast edges for point-to-point communication (defined using type edge) and multicast edges for collective communication (defined using type bedge for broadcast, gedge for gather,
/* Task graph description for target tracking */
edge i2, o2, i3, o3, i4, o4;

task T1 ( output out2, output out3, output out4, input in2, input in3, input in4) {
  o2 = out2 ;
o3 = out3 ;
o4 = out4 ;
in2 = i2 ;
in3 = i3 ;
in4 = i4 ;
}

loop ( i = 2 ; i < 5 ; i++ ) {
  task T<i> ( output out1, input in1) {
    i<i> = out1 ;
in1 = o<i> ;
  }
}

Figure 5-3. An example application graph description in SCF task-graph language.

*edge* for scatter). Depending on the type of an edge, it could connect one or more input and output interfaces. In Figure 5-3, the interfaces for task T1 are connected to the interfaces of other tasks using unicast edges (i2, o2, etc.). The edge assignments are based on the following convention: the output interfaces are assigned to an edge (for example, out2 of T1 is assigned to o2), and the input interfaces receive their values from an edge (for example, in2 receives its value from i2 in T1).

To simplify specification of task graphs of scalable applications which typically include several instantiations of the same task, the language provides a *loop* construct. A *loop* construct creates a loop iterator which is only defined inside the body of the loop. The range of the loop iterator controls the number of tasks instantiated by that loop. The values of the loop iterator are employed in the body of the *loop* (enclosed between <>) to specify unique names (or IDs) for the instantiated tasks and to make any edge assignments. Figure 5-3 employs the *loop* construct to define and perform edge assignment for tasks T2, T3 and T4.
The mapping information, which defines how the tasks of an application are mapped on to the resources of a target system, is specified in a separate file using the task-graph editor. The basic mapping in SCF requires that each task of an application is assigned to a device on a target system. The assignment of a task to a device is performed by specifying the unique address of that device in the system. In addition, few other attributes are also provided to direct SCF tools in synthesizing appropriate communication routines such as the target IDE, communication capabilities available at different levels of hierarchy (device, platform, and system), etc. The next section will provide an example of a mapping file along with the task graph of corresponding application, while describing the steps involved in developing an application using SCF.

### 5.2.2 Tool Flow

The SCF tool flow shown in Figure 5-4 begins with a task definition step. Designers define the computations of tasks, or use appropriate cores, using any language, compiler, or synthesis tool, while defining all task interactions using primitives from the SCF library. Figure 5-5A shows the task graph of an example application consisting of two tasks, which will be mapped to a CPU and FPGA, respectively. As shown in Figure
one task, written in C++, generates random numbers and outputs the random
tables through an interface called “in1” and accumulates
them. Note that these two tasks are defined independently of each other, using different
programming languages. The communication is specified using the SCF library, which is
adapted to meet the requirements of the programming environment associated with the
target device.

After defining tasks, a designer specifies the task graph of application in SCF’s
task-graph editor. Figure 5-5C shows the corresponding task graph (“.scf” file) for the
application under consideration. As shown in Figure 5-5C, the output of task “random” is
connected to the input of task “accumu” through the “edge1” edge. Note that the names
of the interfaces used in the task graph are same as the labels supplied (as one of the
parameters) to the library routines in the task-definition code in Figure 5-5B.

Next, an application designer specifies the mapping information and provides the
architectural model for a target system. An example mapping (“.map” file) is shown in
Figure 5-5D. Although this mapping process is currently performed manually, existing
design-space exploration techniques could be integrated into the SCF tool flow to
provide optimal mapping suggestions to application designers.

Communication synthesis (Step 4 in Figure 5-4) analyzes the mapping and
the architectural model to automatically create efficient implementations for all
edges in the task graph. At each level of hierarchy in the SCF architecture model,
communication synthesis determines what mechanisms to use, and based on this
information it generates definitions of the communication functions for different
devices and platforms. In the simplest case, communication synthesis determines if
an edge of the task graph corresponds to device-level, platform-level, or system-level
communication and translates SCF library functions to an underlying library specified
in the mapping file. For example, on a platform consisting of a CPU and PCI-X FPGA
Communication synthesis would define an SCF_Send from a host CPU to an FPGA using vendor-specific API to transfer data over PCI-X. Although communication synthesis currently implements all communication as a mapping onto underlying vendor API calls, there are numerous possibilities for future work.

The SCF tools extract information from the “.scf” and “.map” files, and further invoke separate plug-ins (one for each IDE) to automatically generate the communication routines for different programming languages. For the example showed in Figure 5-5, there will be a separate plug-in for C++ and Handel-C, each of which will generate the behavior of the SCF_Send and SCF_Recv routines for their respective tasks.

Such a structure allows for new computational devices along with their programming languages and tools to be easily integrated into our implementation, by simply creating a plug-in for the new tool (or programming language). We hope such a framework will
be amenable to vendors of future technology, and provide an easy mechanism for using their technology with other devices in the system. After communication synthesis, the user combines the definitions for the SCF library functions with their corresponding task-definition code and compiles them collectively using the native compiler of the associated programming language to form an SCF executable (which represents a set of multiple executables, one for each participating device in a target system) that can run on the target system.

5.2.3 Interfacing with External Modeling Tools

The format of SCF task graphs allows easy interfacing with external tools that offer the capability of early design-space exploration through algorithm modeling. Such tools also provide application developers with a mechanism for describing applications as abstract models (in certain cases a graphical) which can then be automatically translated into SCF task graphs. To provide developers with a graphical interface for describing task graphs of applications and performing early design-space exploration, we integrated our framework with RCML which is a modeling language for abstract modeling of RC applications. While RCML has several interesting features for modeling and analysis of an application prior to any implementation, this section discusses the equivalence between RCML models and SCF task graphs and the capability of automatically translating the former into the latter through an example. We refer interested readers to [21] for further details about RCML.

Figure 5-6 shows an RCML model of a target-tracking application (application presented as a case study in Section 5.3.3) and its equivalent task graph generated automatically by our framework. An application in RCML is modeled as a collection of blocks and RC-specialized constructs, in which each parallel task is represented by a function block. Each function block of an RCML model can directly correspond to a task in SCF task graphs, or multiple function blocks can be combined into a single SCF task graph. We achieve the combination of multiple RCML blocks into a single
SCF task by defining an attribute called TaskID for each function block. Function blocks with same value for TaskID are combined into a single task when translating an RCML model into its equivalent SCF task graph. Connections that link the blocks in an RCML model represent communication and dependencies, similar to the edges in SCF task graphs. RCML supports all of the various communication patterns currently provided in our implementation of SCF such as broadcast, scatter, gather, and point-to-point communication.

5.3 Results and Analysis

In this section, we present experiments illustrating the productivity and performance advantages of SCF. The two experimental systems employed in this phase of research are detailed in Appendix A (Heterogeneous testbed and Novo-G). We first describe the coordination library which we developed to make these systems SCF compliant. We then demonstrate the advantages of custom communication in SCF by comparing it with communication based on a packet-switched, NoC communication architecture. Finally,
Figure 5-7. SCF architectural model for experimental system.

we illustrate the capabilities of SCF and benefits of rapid-design space exploration enabled by SCF through two case studies.

5.3.1 Experimental Setup

To evaluate the strengths and weaknesses of SCF, we conducted our experiments on two different systems. The first system is the heterogeneous testbed detailed in Appendix A. This system can be represented in the SCF architectural model as shown in Figure 5-7. The combination of the four FPGAs and the host CPU form an SCF platform (P1 in the figure) and the second CPU forms another SCF platform (P2 in the figure). These two platforms collectively form our first experimental system. Each FPGA and CPU is an SCF device. The second system is the Novo-G RC supercomputer detailed in Appendix A.

We developed the SCF library to support various levels of communication on both the systems. System-level communication between the processors on different servers is established using MPI as the underlying communication mechanism. Platform-level communication for the host processor on each server, which allows the host processor to interact with the FPGAs, was supported by using API calls provided by GiDEL. Note that an application designer is not exposed to MPI or GiDEL APIs. Instead, a designer
simply specifies all the communication using the coordination library calls, which SCF tools automatically map onto appropriate underlying communication mechanism (e.g., MPI and the GiDEL API in our case). Platform-level communication on the FPGAs is supported through send and recv entities which we developed in VHDL and are analogous to send (or recv) function calls on a CPU. The entities connect to the user’s VHDL applications through data and control ports (such as ‘go’ and ‘done’ signals), and perform handshaking operations through registers (instantiated on FPGA) to exchange notifications and locations of outstanding messages with the host CPU and other FPGAs.

For these experiments, we did not implement full SCF compliance on each system, and instead implemented only the types of communication necessary for the targeted applications. For example, our implementation of the SCF library does not currently support direct communication between devices such as a CPU and an FPGA on different nodes, or between two FPGAs on different nodes. Note that these limitations are only related to our implementation of the library on these machines, and are not limitations of SCF. Such communication can be implemented through a two-level hierarchy, where the data from any device is first transferred to the host CPU on the originating SCF platform. Then, the data is transferred to the host CPU on the destination SCF platform, from which it can be transferred to the destination device.

5.3.2 Custom Communication Synthesis

One of the advantages of SCF is its ability to synthesize custom communication well suited to the requirements of the application and the capabilities of the system. We demonstrate the advantages of such a scheme versus a generic, packet-switched solution using a simple example, which involves sending data from two tasks to a third task. The task graph of the application is shown in Figure 5-8A, in which tasks T2 and T3 send data to T1. The figure also shows two possible designs for this example, both of which were implemented on a single FPGA.
Figure 5-8. Results for custom communication synthesis. A) Task graph of an application. B) NoC-based design. C) Customized communication. D) Speedup obtained.

The first design (Figure 5-8B) employs a simple router that has a connection to each task with a FIFO to buffer outgoing data on its output ports. While this design is generic and can support a variety of permutations for data communication between connected tasks, it fails to exploit information specified by data dependencies in the task graph. The second design (Figure 5-8C), alternatively, adopts a customized design based on details of data communication extracted from the task graph. It instantiates T1 with two ports and connects them to T2 and T3 directly. As a result of this optimization, the latter design offers superior performance over the generic solution, as indicated by the application execution times for two different data sizes in Figure 5-8D.

In a similar manner, custom communication, one of the important components of SCF, can lead to better application designs in other situations. Although communication synthesis is currently defined by implementing all communication as a mapping onto underlying features (including vendor-provided features as well as the functionality that we developed as a part of this phase of research) for establishing communication, there
are numerous automatic synthesis possibilities enabled by SCF which will be explored in our future research.

5.3.3 Case Study: Target Tracking

In this section, we analyze the overall performance of target-tracking application (Appendix B) which tracks three objects using three Kalman filters using SCF to rapidly explore different resource mappings. Figure 5-9 shows the task graph of the application. Task T1, the sensor process, creates the inputs for all three filters and is implemented in C++. The three Kalman filters (tasks T2, T3, T4) can be mapped on a CPU (with its design implemented in C++) or on an FPGA (as a VHDL design).

Table 5-2 presents execution times of the application under various mapping scenarios for three different systems. System I is our first experimental system (represented in Figure 5-7). Systems II and III represent notional systems, emulating the characteristics of a system that has lower bandwidth and higher latency of communication between the CPU and FPGAs. When power consumption is a major consideration, systems often employ FPGAs running at a lower frequency, perhaps attached to the system over a lower-speed bus. We implemented these emulations by adding extra delays on the FPGA in our experimental system to reduce communication bandwidth. The rows of the table represent the number of the Kalman-filter tasks (amongst T2 to T4) mapped to CPUs and FPGAs. T1 is always mapped on the CPU.
device of platform P1 (in Figure 5-7). Each of the other three tasks either time-shares the CPU device on platform P2 with other tasks mapped onto it or executes on an FPGA on platform P1.

Table 5-2. Execution time of target-tracking application under different mapping scenarios for three systems. Speedup shows performance gain of optimal mapping (highlighted in bold) compared to all-CPU baseline (i.e. 3-CPU, 0-FPGA mapping).

<table>
<thead>
<tr>
<th>Mappings</th>
<th>System I</th>
<th>System II</th>
<th>System III</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-CPU, 0-FPGA</td>
<td>243</td>
<td>243</td>
<td>243</td>
</tr>
<tr>
<td>2-CPU, 1-FPGA</td>
<td>152</td>
<td>152</td>
<td>167</td>
</tr>
<tr>
<td>1-CPU, 2-FPGA</td>
<td>67</td>
<td>86</td>
<td>167</td>
</tr>
<tr>
<td>0-CPU, 3-FPGA</td>
<td>8</td>
<td>86</td>
<td>167</td>
</tr>
<tr>
<td>Speedup</td>
<td>28.3</td>
<td>2.8</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Table 5-2 shows that the three systems, although similar, yield different optimal mappings (where optimal mapping is defined as the mapping which achieves best performance and, in case of tie, with least number of FPGAs). With SCF, exploring these different mappings only required simple modifications to the resource mapping file, and none to source code of the task definitions. The SCF communication synthesis tool adapted the communication infrastructure based on information in the mapping file. In contrast, any such changes traditionally would require modifications to the application source code and designer intervention to create communication infrastructure to match the new resource mapping. Moreover, the process would have to be repeated multiple times until a suitable level of performance is obtained. With SCF, we were able to perform design-space exploration rapidly, which led to speedups ranging from 1.4 times faster to more than 28.

In order to understand productivity gains obtained by employing SCF, we recorded development hours spent by our team during our experiments, in addition to source lines of code (SLOC) involved in certain parts of the application code. Table 5-3 shows the increase in source lines of code involved for establishing communication from the
Table 5-3. Productivity improvement for target-tracking application.

<table>
<thead>
<tr>
<th>Productivity improvement (for FPGA comm.)</th>
<th>Without SCF</th>
<th>With SCF</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOC</td>
<td>357</td>
<td>112</td>
<td>3.18 ×</td>
</tr>
<tr>
<td>Development hours</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conservative</td>
<td>40 hrs</td>
<td>16 hrs</td>
<td>2.5 ×</td>
</tr>
<tr>
<td></td>
<td>(1 week)</td>
<td>(2 days)</td>
<td></td>
</tr>
<tr>
<td>Optimistic</td>
<td>80 hrs</td>
<td>16 hrs</td>
<td>5 ×</td>
</tr>
<tr>
<td></td>
<td>(2 weeks)</td>
<td>(2 days)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5-4. Overhead measurements for target-tracking application.

<table>
<thead>
<tr>
<th>Overhead (on FPGA)</th>
<th>Without SCF</th>
<th>With SCF</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>76 MHz</td>
<td>75 MHz</td>
<td>1.3%</td>
</tr>
<tr>
<td>ALUTs used</td>
<td>2095/143520</td>
<td>2152/143520</td>
<td>2.6%</td>
</tr>
<tr>
<td>Interconnect resources used</td>
<td>11%</td>
<td>12%</td>
<td>8.3%</td>
</tr>
</tbody>
</table>

CPU to each FPGA without SCF. A large part of this improvement comes from hiding
details of communication from the designer while presenting a simpler interface through
SCF library that does not change depending on the source and destination of the
communication.

Table 5-3 also presents productivity estimates, which are explained as follows.
There are a variety of factors that contribute towards improvement in developer
productivity, which are listed in Table 5-5. Applications developed without using SCF
exhibit higher conceptual complexity. Developers are often forced to employ (and learn)
multiple libraries with varying APIs to incorporate communication between different
combinations of devices. Alternatively, with SCF, application developers do not have to
go through a learning curve to familiarize themselves with a vendor-specific API when
migrating to a new system. Similarly, high-level SCF functions eliminate the need to
explicitly perform various intermediate steps of communication, leading to a reduction
in code size. For example, SCF can enable data transfers from a CPU to an FPGA
on a remote platform through a single send routine. Without SCF, any communication
with an FPGA on a remote platform will have to be explicitly routed by a developer,
through the host CPU on that platform. The processing on the host CPU of the remote

97
platform will have to be interrupted to service this communication request, which further increases the complexity of developing parallel programs. Portability increases the application lifespan, promotes code reuse, and reduces the recurring cost that would have been involved without the use of a framework like SCF. Additionally, a developer does not have to make complex modifications to the source code several times to evaluate different mapping configurations. A combination of the factors shown in Table 5-5 (and more) had a collective influence on the development time for the target-tracking application and led to a reduction in the application development time as reported in Table 5-3.

Table 5-5. Major factors that contribute to increased developer productivity using SCF.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program complexity</td>
<td>High-level abstraction provided by SCF functions hides various underlying details from application developers</td>
</tr>
<tr>
<td>Learning curve</td>
<td>Familiar APIs and programming model lead to reduced learning period when migrating to a new system</td>
</tr>
<tr>
<td>Source lines of code</td>
<td>Each SCF function can perform several intermediate steps of communication which eliminates the need for extra code and function calls</td>
</tr>
<tr>
<td>Application portability</td>
<td>Allows parts of an application to be re-used by other applications. Additionally, applications can be executed on a variety of platforms which reduces recurring costs</td>
</tr>
<tr>
<td>Design-space exploration</td>
<td>No modifications are required in the source code to evaluate different resource mappings using SCF</td>
</tr>
</tbody>
</table>

Based on these contributing factors, we estimate SCF can reduce development time and improve productivity by a factor ranging from approximately 2.5 to 5, as shown in Table 5-3. The optimistic case represents a case where a designer is unfamiliar with the system, and thus has to undergo a steep learning process for the APIs for each device. The conservative case represents an experienced designer who is familiar with the tools and vendor-APIs for that particular system. Although these numbers are specific to
our experimental system and team personnel, we believe them to be a fair estimate of improvements we expect to obtain with SCF.

Table 5-4 lists the overhead incurred by our VHDL design employing SCF for communication, in comparison to optimized, handwritten design developed for the same application. These results show that the communication routines employed by the SCF tools result in modest overheads in terms of both resources and performance.

5.3.4 Case Study: Backprojection

Figure 5-10. Task graph for backprojection application.

In this section, we present a case study on backprojection (detailed in Appendix B) application running on Novo-G. Due to the heavy computation demands of the application, most implementations decompose the backprojection algorithm in an embarrassingly parallel manner to exploit its inherent parallelism. In this case study, we first write a parallel program to accelerate the backprojection algorithm on a cluster of microprocessors. We then seamlessly migrate parts of the application onto FPGAs to further accelerate the application and determine the best mapping of the algorithm onto system resources. For our evaluation, we employ the backprojection algorithm to generate a $512 \times 512$ image. We decompose the computational load of the algorithm over four processing devices (CPUs or FPGAs), each of which receives its input data and sends its result to a task mapped on a CPU as shown by the task graph.
in Figure 5-10. Table 5-6 lists the computation time and communication time for the backprojection task measured on each of the two device types on Novo-G. The times for transferring both the input and output to a task are listed in the row labeled as Data transfer. The data-transfer time for the CPU corresponds to the time for transferring data between two CPUs on different SCF platforms, while for the FPGA it represents the transfer time between a CPU and an FPGA on the same platform.

Table 5-6. Computation and communication performance of backprojection task on a CPU or FPGA.

<table>
<thead>
<tr>
<th>Function</th>
<th>Resource</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backprojection computation</td>
<td>CPU</td>
<td>2145.75</td>
</tr>
<tr>
<td>Backprojection computation</td>
<td>FPGA</td>
<td>3.28</td>
</tr>
<tr>
<td>Data transfer (input/output data)</td>
<td>CPU</td>
<td>0.33/0.85</td>
</tr>
<tr>
<td>Data transfer (input/output data)</td>
<td>FPGA</td>
<td>1.50/2.16</td>
</tr>
</tbody>
</table>

Table 5-7 lists the execution time of the application under various mapping scenarios on Novo-G. The mapping configuration in the table denotes the ordered mapping of tasks T2-T5 (in Figure 5-10) onto the system resources of Novo-G. In the simplest case (case 1), all of the tasks sharing the computational load of backprojection algorithm are mapped onto four different cores of a quad-core processor on a single platform. For case 2, the tasks are mapped on different CPUs that reside in different SCF platforms. The next three cases (cases 3-5) achieve similar performance. The performance in these cases is limited by the slowest performing task (i.e. tasks mapped onto CPUs). It is interesting to observe the slight variation in performance between cases 4 and 5 due to a change in the order of task mappings. The last case (case 6), implements each of the tasks sharing the computational load on a different FPGA of a single SCF platform. Due to the superior computational capability of FPGAs, this mapping offers the best performance.

While the optimal mapping is reasonably self-explanatory for this case study, it may not always be obvious. For example, one may not expect case 3 (with two FPGAs) to be slower than case 1 with only CPUs. The different mapping possibilities are listed...
here to illustrate the ease of evaluating various mapping configurations of application
tasks using SCF, which may be critical for finding optimal mappings in other applications.
Determining the performance of various mapping configurations listed in Table 5-7
did not require any change in the source code by application developers. Instead, a
developer just made modifications to the mapping file in the SCF editor and followed
the steps 4-7 from Figure 5-4 to generate a particular executable. Since most of the
inferences on the productivity analysis from this case study were consistent with the first
case study, they are not repeated here.

Table 5-7. Performance of backprojection application under different mapping scenarios.
Mapping configuration represents the ordered mapping for the four tasks
sharing the computational load of the application. Speedup compares
performance of a particular mapping configuration with a serial software
baseline which requires 8.4s on a single CPU of Novo-G. For the device
mappings, all FPGAs are always on the same platform as the host CPU to
which Task T1 is mapped. CPU devices are on different platforms except

<table>
<thead>
<tr>
<th>Case</th>
<th>Mapping configuration</th>
<th>Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU, CPU, CPU, CPU</td>
<td>2213.8</td>
<td>3.80</td>
</tr>
<tr>
<td></td>
<td>(all CPUs on same platform)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CPU, CPU, CPU, CPU</td>
<td>2323.4</td>
<td>3.62</td>
</tr>
<tr>
<td></td>
<td>(all CPUs on different platforms)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CPU, CPU, FPGA, FPGA</td>
<td>2317.2</td>
<td>3.63</td>
</tr>
<tr>
<td>4</td>
<td>CPU, FPGA, FPGA, FPGA</td>
<td>2276.9</td>
<td>3.70</td>
</tr>
<tr>
<td>5</td>
<td>FPGA, FPGA, FPGA, CPU</td>
<td>2285.4</td>
<td>3.68</td>
</tr>
<tr>
<td>6</td>
<td>FPGA, FPGA, FPGA, FPGA</td>
<td>12.5</td>
<td>672.96</td>
</tr>
</tbody>
</table>

5.4 Conclusions

To complement the work in Phases 1 and 2 of this research and to address
challenges involving task coordination in future reconfigurable, heterogeneous
systems, we have presented a novel framework for system-wide coordination that
enables communication and synchronization between tasks running on heterogeneous
processing devices in a system. SCF hides the low-level communication details from the
application designer, resulting in improved productivity. By allowing designers to define
communication independently of the devices in a system, SCF improves application
portability. In addition, SCF allows designers to define tasks using potentially any language, which enhances the inter-operability between different vendor tools.

In this chapter, we evaluated an implementation of SCF and its associated tools and libraries through experiments on two different systems. Our experiments indicate that custom communication, one of the important components of SCF, creates designs that offer superior performance over generic solutions. We also illustrated the simplified design process involved with SCF for developing RC applications, which enabled rapid design-space exploration of various mappings to obtain higher performance from applications in our case studies. More importantly, the higher level of abstraction offered by this framework led to substantially improved designer productivity. Although productivity gains are often difficult to quantify, our case studies demonstrate an improvement ranging between $2.5 \times$ to $5 \times$. 
CHAPTER 6
COMPARISON OF SHMEM+ AND SCF

In this chapter, we compare and contrast the capabilities and characteristics of SHMEM+ and SCF. Both of the approaches presented in this research, SHMEM+ and SCF, provide alternative techniques for developing scalable, parallel applications for RC systems. The choice of the approach adopted for a particular application is driven largely by the requirements of that application and developer’s preference for a programming model. While SHMEM+ is based on the PGAS programming model, SCF uses the message-passing model for establishing communication. As a result, these approaches have several contrasting features such as the model of decomposition adopted for the application, the programming complexity offered to an application developer, the performance of the communication routines in each, as well as their data-transfer capabilities. In the following sections, we evaluate both of the approaches based on the aforementioned characteristics and compare the performance obtained by data transfer routines in each.

6.1 Programming Model and Complexity

Applications developed using SHMEM+ typically follow the SPMD model for algorithm decomposition, i.e., every participating node in a system executes an instance of the same program. Each program instance is partitioned across, and collectively executed by all the heterogeneous devices in the system. By contrast, SCF allows any arbitrary decomposition for a parallel application. Task graphs in SCF can represent applications described using SPMD decompositions, as well as functional decompositions to create pipelines. In addition, SCF also allows any assignment of application tasks on to the heterogeneous resources in a system.

Most RC applications are composed of two portions, the hardware core written in an HDL, and the software code that is specified in a programming language such as C/C++. Both SHMEM+ and SCF aim to reduce the program complexity of a parallel, RC
application including both software development, as well as hardware-core development. The two-sided communication routines in SCF require an application developer to provide matching send and receive operations to complete a data transfer. By contrast, SHMEM+ accomplishes the same data transfer using one-sided get and put routines. As a result, an application developer does not have to program matching send and receive pair when using SHMEM+. Consequently, programs written using SHMEM+ provide lower software program complexity.

For description of hardware cores, SCF provides support for data transfers through VHDL entities which operate in a manner similar to send and receive functions in software. In contrast, the hardware cores developed for SHMEM+ applications currently employ vendor-specific functionality to fetch data from the on-board memory. As a result, SCF offers lower programming complexity for hardware cores compared to SHMEM+.

6.2 Performance of Data Transfers

One-sided communication routines in SHMEM+ do not require coordination with the other task involved in a data transfer. Consequently, communication routines in SHMEM+ have the potential to offer higher performance than the ones in SCF, which require the communicating tasks to synchronize prior to completing a transfer. However, for certain types of communications (such as, communication between two FPGAs), the custom communication synthesis in SCF can optimize the communication infrastructure to offer superior performance.

Figure 6-1 shows the average bandwidth of transfers for a ping-pong benchmark transferring data between a CPU and an FPGA (when both devices are on the same node) using communication routines from SHMEM+ and SCF. The performance of transfers using SHMEM+ routines is superior to that of routines in SCF. The lower bandwidth for SCF routines can be attributed to the overheads of a data transfer encountered on an FPGA when performing a ping-pong test. The *SCF_Send* and
Figure 6-1. The average round-trip bandwidth for a ping-pong microbenchmark using communication routines in SHMEM+ and SCF.

$SCF_{Recv}$ entities on FPGAs, write (or read) one data element in every clock cycle. As a result, the time required to complete a data transfer to (or from) the on-board memory from (or to) an application core on an FPGA, becomes substantial for large data transfers. Fortunately, most applications can overlap this overhead with useful computation and therefore do not experience a performance penalty when employing SCF.

### 6.3 Data-Transfer Capabilities

Both SHMEM+ and SCF enable new functionality and direct transfers between devices that were previously achieved by employing multiple communication libraries (and APIs). For example, SHMEM+ enables the capability of communicating directly with FPGAs on a remote node by employing active messages for such transfers. Although not a limitation of our framework, such functionality is difficult to implement in SCF, and is currently missing from our implementation. By contrast, an advantage of SCF is that FPGAs are treated as peers to CPUs in the system and have the ability of initiating transfers. To enable such transfers in SHMEM+, FPGAs need a mechanism for accessing the system memory to complete a one-sided communication, a feature which is often missing on many RC systems where FPGAs are connected to the host.
processor through an I/O bus. Consequently, FPGA-initiated transfers are difficult to provide in SHMEM+ and are missing from our current implementation.

6.4 Conclusions

Both SHMEM+ and SCF address system-wide issues and challenges involved in developing scalable applications for reconfigurable HPC and HPEC systems. Each approach has its unique capabilities (and certain limitations) that might make one approach more suitable than the other for a given application. An application developer will typically choose between the two approaches based upon the requirements of a given application and preference for a programming model, which is similar to the choice between different programming models in the traditional parallel-programming community.
CHAPTER 7
CONCLUSIONS

This research addresses system-level issues and challenges involved in developing scalable applications for reconfigurable HPC and HPEC systems. To this end, we have proposed two complementary approaches for establishing communication between tasks of parallel RC applications, SHMEM+ and System Coordination Framework. By providing a high-level abstraction and a uniform communication interface, both of these approaches aim to improve developer productivity and application portability. Further improvements in productivity are obtained by bridging these design approaches with appropriate formulation tools to enable early design-space exploration.

In Phase 1 of this research, we proposed and investigated a new parallel-programming model called multilevel-PGAS, and based on this model, we created SHMEM+, the first known version of the SHMEM library for reconfigurable HPC systems. We developed a prototype for the SHMEM+ library and demonstrated its effectiveness on Novo-G through two case studies. Results from our experiments and case studies demonstrate that SHMEM+ is capable of offering performance comparable to an existing vendor-proprietary version of SHMEM while simultaneously yielding improved productivity and portability.

In Phase 2, a new multilevel communication model for estimating the performance of multilevel transfers present in SHMEM+ is proposed and evaluated. The model is extremely beneficial to application developers for providing a concrete understanding of the underlying communication infrastructure as well as optimizing the communication in these applications. Besides attaining higher performance from the communication infrastructure, our model can lead to considerable improvement in the productivity of application and system developers. Results from our experiments illustrate the merits and accuracy of the multilevel communication model through three different use-case scenarios.
In Phase 3, we proposed and analyzed a novel framework for system-wide coordination which provides an approach based on message passing for establishing communication between various devices in a heterogeneous system. By allowing the use of different tools and languages for describing various tasks of a parallel application, SCF promotes interoperability amongst various tools. In addition, by hiding the low-level communication details, SCF yields improved productivity and application portability. We demonstrated the benefits of developing applications with SCF using two case studies where improvement in application performance was obtained due to rapid design-space exploration. In addition, a productivity improvement ranging from $2.5 \times$ to $5 \times$ was achieved with minimal performance overheads.

Contributions of this research include the multilevel-PGAS programming model, the SHMEM+ communication library, the system coordination framework for task-level coordination, and the multilevel communication model for estimating performance of multilevel transfers. Combined, these tools and frameworks can overcome several existing challenges in developing parallel RC applications that have limited the potential of RC in the fields of HPC and HPEC.

Although this research focuses mainly on scalable RC systems and applications, the communication tools and framework proposed here can be extended to systems based on other types of accelerators such as GPUs, many-core processors, etc. The potential uses and impact of these tools on such systems could be explored in future research. In addition, the prototype of the SHMEM+ library developed in this research is expected to be a foundation for development of a more complete library in the future.
APPENDIX A
DESCRIPTIONS OF RC PLATFORMS EMPLOYED

This appendix describes the different RC platforms that were employed by various case studies throughout this research.

A.1 Mu Cluster

The Mu cluster consists of four Linux servers connected via QsNetII from Quadrics. Each node of the cluster features:

- 2GHz AMD Opteron 246 processor
- 1GB of registered-ECC DDR400 RAM
- PROCStar-III PCIe x8 quad-FPGA card from GiDEL [65]
  - 4 Altera Stratix-III EP3SE260 FPGAs
  - 256 MB DDR II
  - Two banks of 2GB DDR II (SODIMM)
  - High-speed direct-connection between neighboring FPGAs

A.2 Novo-G

Novo-G consists of 24 computer servers connected by DDR InfiniBand and GigE. An architecture overview of Novo-G is provided in Figure A-1. Each node consists of the following hardware:

- 2.26GHz Intel Xeon E5520 quad-core processor
- 6GB of ECC DDR3, 1333 MHz RAM
- Mellanox DDR InfiniBand PCIe card
- PROCStar-III™PCIe x8 quad-FPGA card from GiDEL [65]
  - 4 Altera® Stratix®-III EP3SE260 FPGAs
  - 256 MB DDR II
  - Two banks of 2GB DDR II (SODIMM)
  - High-speed direct-connection between neighboring FPGAs
Figure A-1. System architecture of Novo-G RC supercomputer.

A.3 Heterogeneous Testbed

This testbed consists of two standalone Windows servers connected via Gigabit Ethernet. One of the nodes is comprised of a 3Ghz Xeon processor. The second node consists of

- 2GHz Athlon 3200+ processor
- PROCStar-II™PCI-X quad-FPGA card from GiDEL [66]
  - 4 Altera® Stratix®-II EP2S180 FPGAs
  - 64MB DDR II
  - 1GB DDR II (SODIMM)
  - High-speed direct-connection between neighboring FPGAs
APPENDIX B
DESCRIPTIONS OF APPLICATIONS EMPLOYED

B.1 Target Tracking Using Kalman Filter

Target tracking using Kalman filtering [67] is a method for predicting the trajectory of environmental targets such as vehicles, missiles, animals, hostiles, or even unidentified objects. A Kalman filter is commonly employed in signal processing applications to estimate the dynamic system state in a noisy environment. We selected this application in this research due to its numerous constraints that are often met using heterogeneous devices. There are a variety of factors such as error tolerance, sampling rate of input, target proximity to sensors, etc. that determine the exact operational characteristics required for a particular target-object. Different targets have varying requirements which mandate an appropriate computational platform such as an FPGA, an embedded processor or a desktop processor (CPU).

B.2 Content-Based Image Retrieval (CBIR)

Content-Based Image Retrieval (CBIR) is a common application in computer vision and consists of searching a large database of digital images for the ones that are visually similar to a given query image, where the search is based on contents of the image. The content in this context can be one of the several features present in the image, such as colors, shapes, textures, or any other information that can be derived from the image. CBIR has been widely adopted in many domains such as biomedicine, military, commerce, education, and Web image classification and searching. Each image in a CBIR system is represented by a feature vector, which is based on characteristics of the image as cited above. Similarity between a query image and the set of images in the database is determined by measuring similarity between their feature vectors. The processes of determining the feature vector and analyzing images for similarities are often the most computationally intensive stages in any CBIR system [68]. There
are various forms of parallelism available in the application that can be exploited by RC systems to accelerate the search process [69].

Our implementation presented in this research employs a technique based on auto-correlogram of color components [70], where the feature vector is based on color information in the image. A correlogram of an image corresponds to a table where the rows are indexed by color pairs \((c_i, c_j)\) such that the \(d\)-th column in row \((c_i, c_j)\) stores the probability of finding a pixel of color \(c_j\) at a distance \(d\) from a pixel of color \(c_i\) in the image. For the case of auto-correlogram, the table only consists of rows where \(c_i = c_j\). In this work, we use a modified version of auto-correlogram, which stores an absolute count of the occurrences of a pixel of color \(c_i\) instead of the probability of such an event. Similarity between two images is determined by calculating the sum of absolute differences between their feature vectors.

### B.3 Backprojection

Backprojection [71] is the process (also commonly referred to as a transform in literature) of reconstructing the internal structure of a scanned object from emission data (or cross-section data) acquired by a scanner. The process is employed in various application domains such as medical imaging, synthetic aperture radar (SAR), electron microscopy, etc. Backprojection can be viewed as mapping of raw data into the image space. It is typical for applications to have thousands (to millions) of data points for each cross-section and thousands of cross-sections from different angles. The process of transforming each data element from the sensors to the image space is computationally demanding having a complexity of \(O(n^3)\) when generating 2D images [72].
REFERENCES


[14] O. Storaasli, “Accelerating genome sequencing 100-1000X with FPGAs,” in Many-core and Reconfigurable Supercomputing Conference (MRSC), The Queen’s University of Belfast, Northern Ireland, 2008.


[38] OpenFPGA, “OpenFPGA GenAPI version 0.4 draft for comment,” http://www.openfpga.org/Standards%20Documents/OpenFPGA-GenAPIv0.4.pdf, Retrieved February 2011.


BIOGRAPHICAL SKETCH

Vikas Aggarwal is a Ph.D. graduate from the Department of Electrical and Computer Engineering at the University of Florida. Prior to beginning his doctoral studies, he worked with IBM Research Lab in India, and Tandel Systems Inc. in the United States. He received his M.S. in electrical and computer engineering from the University of Florida in December of 2005. Prior to coming to the States, he received his B.S. degree from the department of ECE at Guru Gobind Singh Indraprastha University, India, in August of 2003.