Using FPGAs to Simulate Novel Datacenter Network Architectures At Scale

Zhangxi Tan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-124
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-124.html

June 21, 2013

Copyright © 2013, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Using FPGAs to Simulate Novel Datacenter Network Architectures At Scale

by

Zhangxi Tan

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor David A. Patterson, Chair
Professor Krste Asanović
Professor David Wessel

Spring 2013

The dissertation of Zhangxi Tan is approved.

Chair Date

Date

Date

University of California, Berkeley
Spring 2013

Using FPGAs to Simulate Novel Datacenter Network Architectures At Scale

Copyright © 2013

by

Zhangxi Tan

Abstract

Using FPGAs to Simulate Novel Datacenter Network Architectures At Scale

by

Zhangxi Tan

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David A. Patterson, Chair

The tremendous success of Internet services has led to the rapid growth of Warehouse-Scale Computers (WSCs). The networking infrastructure has become one of the most vital components in a datacenter. With the rapid evolving set of workloads and software, evaluating network designs really requires simulating a computer system with three key features: scale, performance, and accuracy. To avoid the high capital cost of hardware prototyping, many designs have only been evaluated with a very small testbed built with off-the-shelf devices, often running unrealistic microbenchmarks or traces collected from an old cluster. Many evaluations assume the workload is static and that computations are only loosely coupled with the very adaptive networking stack. We argue the research community is facing a hardware-software co-evaluation crisis.

In this dissertation, we propose a novel cost-efficient evaluation methodology, called *Datacenter-in-a-Box at Low cost* (DIABLO), which uses Field-Programmable Gate Arrays (FPGAs) and treats datacenters as whole computers with tightly integrated hardware and software. Instead of prototyping everything in FPGAs, we build realistic reconfigurable abstracted performance models at scales of O(10,000) servers. Our server model runs the full Linux operating system and open-source datacenter software stack, including production software such as memcached. It achieves two orders of magnitude simulation speedup over software-based simulators. This speedup enables us to run the full datacenter software stack for O(100) seconds of simulated time. We have built a DIABLO prototype of a 2,000-node simulated cluster with runtime-configurable 10 Gbps interconnect using 6 multi-FPGA BEE3 boards.

To my family

Contents

Contents ii

List of Figures vi

List of Tables viii

Acknowledgements ix

1 Introduction 1

1.1 Background and Motivation 1

1.1.1 Datacenter Networking 1

1.1.2 Limitation of existing evaluation methodologies 2

1.1.3 Our approach: FPGA models 4

1.2 Contribution 5

2 Related Work 7

2.1 Evaluation Methodologies for Hardware and Software Innovations in Datacenter 7

2.2 Evaluations of Datacenter Networking Proposals 8

2.3 Simulation and Prototyping of General Computer System 11

2.3.1 Hardware Prototyping 11

2.3.2 Software Simulation 12

2.3.3 FPGA-Based Performance Modeling and FPGA Computers 13

3 Software and FPGA-based Simulation Methodology 15

3.1 Software Architecture Model Execution: SAME 15

3.1.1 In-order Processor Simulators 16

3.1.2 Out-of-Order Processor Simulators 16
3.1.3 Multiprocessor Simulators 17
3.1.4 The Software Premise of Simulation 17
3.1.5 The Multicore and Cloud Revolution 18
3.2 Three FAME dimensions and Terminologies 19
3.2.1 FAME Implementation Techniques 21
3.2.2 FAME Levels 23
3.3 RAMP Gold: An example of a full-system FAME-7 simulator 27
3.3.1 RAMP Gold Microarchitecture 27
3.3.2 Model Verification and Flexibility 29
3.3.3 RAMP Gold Speedup Results 29
3.4 A FAME-7 Datacenter Switch Model 31
3.4.1 Switch Model Microarchitecture 31
3.4.2 Comparing to SAME simulators 33
3.5 Conclusions 34

4 Building a Scalable DIABLO for Datacenter Network Architecutre Simulations 35

4.1 DIABO Design Strategy 35
4.2 Server Models 37
4.2.1 Mapping to FPGAs 37
4.2.2 Host Memory Interface Design 38
4.2.3 Debugging Infrastructure 41
4.2.4 Server software 43
4.3 Switch Models 44
4.4 Network Interface Card Models 46
4.5 Modeling a Datacenter with a Packet-Switched Interconnect 50
4.5.1 Simulated Datacenter Target 50
4.5.2 Modularized DIABLO system design: Array FPGAs and Rack FPGAs 51
4.5.3 Building a 4,000-node DIABLO System with 11 BEE3 Boards 55
4.5.4 Summary 57

5 Using DIABLO to Conduct Datacenter System Research at Scale 58

5.1 Case One: Modeling a Circuit-Switching Datacenter Network 58
5.1.1 Target network architecture 59
5.1.2 Building DIABLO models on FPGA 61
5.1.3 Dryad Terasort Kernel Data-Plane Traffic Studies 62
5.1.4 DIABLO for Software Development 65
5.2 Case Two: Reproducing the TCP Incast Throughput Collapse 65
5.2.1 Reproducing the TCP Incast Effect on Gigabit Switch with Shallow Port Buffers 67
5.2.2 Study TCP Incast at 10 Gbps Link Speed 68
5.2.3 The Need of Simulating O(1,000) Nodes for Rack-Level Problems . . 71
5.3 Case Three: Running *memcached*, a Distributed In-Memory Key-Value Stores 72
5.3.1 Experiment setup 72
5.3.2 *memcached* Workload generator 73
5.3.3 Validating our single-rack *memcached* results 76
5.3.4 Large-scale Experiments 79
5.4 Conclusion 85

6 Experiences and Lessons Learned 87
6.1 Core selection 87
6.2 Design languages and Impact of FPGA ECAD Tools 89
6.3 ECAD Issues 90
6.4 Building a Reliable Simulation Infrastructure 92
6.4.1 Reliable Control Protocol 92
6.4.2 Reliable Inter-chip Links 92
6.4.3 Reliable DRAM Controller 93
6.4.4 Protecting Against Soft Errors 95
6.5 Pitfalls in DIABLO 96
6.6 Building a Low-Cost FPGA Board for DIABLO 100

7 Future Work 103
7.1 Painkillers: Must-Have Features 103
7.1.1 Improving emulated target memory capacity 103
7.1.2 Adding multicore and 64-bit CPU Support 105

7.2 Vitamins: Nice-To-Have Features 106
7.2.1 More flexible micro-code based NIC/Switch Models 106
7.2.2 Applying DIABLO to other I/O research 108
7.2.3 Supporting more real world applications 108

8 Conclusions **110**

Bibliography **113**

List of Figures

3.1 RAMP Gold Microarchitecture 27
3.2 Wallclock time of RAMP Gold and Simics simulations 29
3.3 Geometric mean speedup of RAMP Gold over Simics across benchmarks . . 30
3.4 FAME model for virtual output queue switches. 32
3.5 Parallel software switch simulation performance. 34

4.1 Host cycle breakdown for RAMP Gold running PARSEC. 39
4.2 RAMP Gold Host Cache Miss Rate running PARSEC 39
4.3 The DIABLO frontend link architecture 41
4.4 Software and hardware architecture of generic network interface cards 46
4.5 DIABLO NIC model architecture 49
4.6 A typical datacenter network architecture. 51
4.7 DIABLO FPGA simulator architecture 52
4.8 DIABLO FPGA simulator architecture 53
4.9 The block diagram for a DIABLO system with 11 BEE3 boards 56
4.10 DIABLO cluster prototype with 6 BEE3 boards 57

5.1 Target architecture of the novel circuit-switching network 59
5.2 Frame formats and data paths of the target circuit-switching network 60
5.3 The L0 switch architecture 61
5.4 Three typical traffic patterns in the Dryad terasort 62
5.5 Link throughput on L0 switch for the Terasort phase one traffic 63
5.6 Link throughput on the L0 switch for the fixed length traffic in phase III . . 64
5.7 Link throughput on the L0 switch for the vairable length traffic in phase III 65
5.8 A simple cluster-based storage environment for the TCP incast problem . . 66

5.9 Reproducing the TCP incast effect with a shallow-buffer gigabit switch . . . 68

5.10 Simuating the TCP incast under a 10 Gbps setup using different system calls on machines with different computing power. 69

5.11 Compare CDF of the generated key size vs. Facebook traffic from the paper 75

5.12 Compare CDF of the generated value size vs. Facebook traffic from the paper 75

5.13 Compare CDF of the generated inter-arrival time vs. Facebook traffic from the paper 76

5.14 Simulated memcahed server throughput vs. Measured throughput on real machines 78

5.15 CPU utilization over time on simulated memcached servers and real machines 79

5.16 Average client request latencies on simulated machines 80

5.17 Percentage of kernel time of memcached servers 81

5.18 Average CPU utilization (median of all samples) per server at different scales 81

5.19 Average minimum free memory per server at different scales 82

5.20 PMF and CDF of client request latency at 2000-node using UDP 83

5.21 Impact of system scales on the latency long tail 84

5.22 Comparing TCP vs UDP on cumulative tail distributions of client request latency at different scale with the 1-Gbps interconnect 84

5.23 Comparing TCP vs UDP on cumulative tail distributions of client request latency at different scale with the 10-Gbps interconnect 85

6.1 Prospective FPGA board optimized for DIABLO. 101

List of Tables

2.1 Datacenter network architecture proposals and their evaluations 10

3.1 Number of instructions simulated, number of processors, and instructions simulated per processor for ISCA 2008 vs. 1998. 20

3.2 Summary of four FAME Levels, including examples. 23

4.1 The control logic resource consumption of the DIABLO coherent cache on Xilinx Virtex 5 LX110T. The increased areas over the RAMP Gold unified cache are shown in parentheses in each cell. 40

4.2 The FPGA resource consumption breakdown of the abstracted NIC model on Xilinx Virtex 5 LX110T 50

4.3 The FPGA resource consumption breakdown of the RACK FPGA on Virtex 5 LX155T 55

5.1 FPGA resource usage and lines of Systemverilog for different FAME models. 62

6.1 Soft error rate of individual components and system in DIABLO, assuming a 20 BEE3 boards with 64 GB DRAM each 96

Acknowledgements

I have been long fascinated by building computer systems since a teenager. I am blessed to have a chance of learning how to build computer systems at Berkeley, the best place in the world. Throughout my career as a graduate student, I believe the thing that influenced me the most is the heritage of Berkeley system research, which I will carry on for the rest of my career.

There is no doubt that my incredible advisor Professor David Patterson who taught me everything has implanted this Berkeley DNA into my blood. Like every graduate student, I started searching for projects by looking at professor's web page. I was captivated by a page on Dave's website that lists all Berkeley hardware prototypes in history. This is how I started my academic career in computer system: I want to build one like those prototypes for my PhD. Dave is extremely helpful not only on every aspect of my research, but also on my own personal life. He is the pillar that provides the greatest emotion support when everything seemed to go wrong in my life. He also gave me great freedom and confidence to carry on my own research when many people cast doubt upon my project. Through him, I learned the most important thing to a computer architect – the beauty of simplicity.

I am also extremely grateful to my co-advisor, Professor Krste Asanovic, who is the most resourceful professor I have ever met. I feel just extremely lucky to work with him to learn his past system building experiences dated back to his work on T0 at Berkeley. I faced many technical challenges in this dissertation, but he could always help me to find a concrete solution. His great knowledge and inspiring supervising style helped to shape the entire system design. Without him, I would not be able to have this memorable thesis in the end.

I would like to thank Chuck Thacker, our great alumni, who literally sets a career standard with himself for our young people to follow. I am also very fortunate to collaborate with Chuck in the RAMP project and work with him during an internship on a circuit-switching datacenter network design, which has become an essential part of this thesis later. I also designed several key components in DIABLO by learning from Chuck's designs. I also want to thank John Davis for providing me fantastic support on the BEE3 board and valuable feedback to my project.

My special thanks go to the wonderful Parlab colleagues and staffs: Andrew Waterman, Rimas Avizienis, Henry Cook, Sarah Bird, who contributed on the RAMP Gold infrastructure and the FAME project; Kostadin Ilov, who spent huge amount of time with me setting up the DIABLO prototype physically; Roxana Infante, who helped prioritizing any purchasing order related with DIABLO; Peter Zhenghao Qian, Peter Xi Chen, Phuc Ton Nguyen, who worked with me as undergrad researchers contributing a lot to the software infrastructure and case studies in this thesis.

I also appreciate helps from many industry and academia experts for their invaluable comments and conversations related to this dissertation: Derek Chiou, Eric Chung, Glen Anderson, Bob Felderman, James Hamilton.

Finally, I express my gratitude to my family my father, mother have always been so

supportive during my research at Berkeley. Thank you my friends - Jenny Chung, Benjamin Yi, Meirong Pai for your kindness and emotional support.

Chapter 1

Introduction

1.1 Background and Motivation

Massive warehouse-scale computers (WSCs) [45] are the foundation of widely used Internet services; for examples, search, social networking, email, video sharing, and online shopping. The tremendous success of these services has led to the rapid growth of datacenters to keep up with the increasing demand, and WSCs today have up to 100,000 servers [82]. At the warehouse scale, an application is usually a large-scale internet service, while the hardware platform consists of thousands of individual computing nodes in datacenters with their corresponding networking, storage, power distribution and cooling systems. Driven by flexibility and cost efficiency, datacenters usually use customized open-source software stacks to address critical performance and functionality issues as needed. For similar scalability and cost reasons, customizations also happen at the hardware level. Instead of buying commercial hardware off the market, some leading internet companies like Google and Facebook are building their own hardware [27, 22], including both servers and networking gear. Due to the emphasis on cost efficiency and scalability, datacenters are no longer just a collection of servers running traditional commercial workloads. In other words, the datacenter is a massive computer system with many hardware and software innovations.

1.1.1 Datacenter Networking

In recent years, many key technologies, such as container-based datacenter construction and server virtualization, enable datacenters to grow rapidly to scales of 50,000 to 100,000 servers [82]. Cost-efficient low-end servers are preferred building blocks at this enormous scale [43]. As networking requirements increase with larger numbers of smaller systems, this leads to increasing networking delays and demand for more ports in an already expensive

switching fabric. As observed in many recent papers [72, 122], network infrastructure has become one of the most vital component in a datacenter. First, networking infrastructure has a significant impact on server utilization, which is an important factor in datacenter power consumption. Second, network infrastructure is crucial for supporting-data intensive Map-Reduce jobs. Finally, network infrastructure accounts for 18% of monthly datacenter costs [72], which is the third largest contributing factor. However, there are many practical issues scaling existing commercial off-the-shelf Ethernet switches, especially at a high link speed, such as 10 gigabit per second (Gbps).

1. Current networks are extremely complex, particularly the switch software.

2. Existing networks have many different failure modes. Occasionally, correlated failures are found in replicated million-dollar units.

3. Existing large commercial switches and routers command high margins and charge a great deal for features that are rarely used in datacenter. Therefore, they are very difficult to scale out to larger configurations without complete redesign.

4. Some datacenters require a large number of ports at aggregate or datacenter-level switches at extremely high link bandwidth. But such switches do not exist on the market currently [67]. For instance, Google G-scale network is running on custom-built 10 Gbps switches with 128 ports, with plans for 40 Gbps systems supporting as many as 1,000 ports [22].

. Therefore, many researchers have proposed novel datacenter network architectures [73, 74, 81, 102, 121, 122] with most of them focusing on new switch designs. There are also several new network products emphasizing low latency and simple switch designs [13, 14]. When comparing these new network architectures, we found a wide variety of design choices in almost every aspect of the design space, such as switch designs, network topology, protocols, and applications. For example, there is an ongoing debate between low-radix and high-radix switch design. Most proposed designs have only been tested with a very small testbed running unrealistic microbenchmarks, as it is very difficult to evaluate network architecture innovations at scale without first building a large datacenter. We believe these basic disagreements about fundamental design decisions are due to the different observations and assumptions taken from various existing datacenter infrastructures and applications, and the lack of a sound methodology to evaluate new options.

1.1.2 Limitation of existing evaluation methodologies

Recent novel network architectures employ a simple, low-latency, supercomputer-like interconnect. For example, the Sun Infiniband datacenter switch [13] has a 300ns port-port latency as opposed to the 7–8μs of common Gigabit Ethernet switches. Even at the datacenter level, where traditional Cisco and Juniper monster switches or routers dominate to handle inter-datacenter traffic, simple switches with minimum software are preferred.

The Google G-scale OpenFlow switch [96] runs almost no software except the OpenFlow agent using just the BGP and ISIS protocols. With rapidly evolving set of workloads and software, and a supercomputing interconnect, evaluating datacenter network architectures really requires simulating a computer system with the following three features.

1. *Scale*: Datacenters contain O(100,000) server or more. Although few single apps uses up all servers in a datacenter, O(10,000) scales are desired to study networking phenomena at aggregate and datacenter-level switches.

2. *Performance*: Current large datacenter switches have 48/96 ports, and are massively parallel. Each port has 1–4K flow tables and several input/output packet buffers. In the worst case, there are ∼200 concurrent events every clock cycle. In addition, high-bandwidth switch processors often employ multicore architectures. For example, Broadcom 100 Gbps Ethernet network processor BCM88030 includes as many as 64 custom multithreaded processors.

3. *Accuracy*: A datacenter network operates at nanosecond time scales. For example, transmitting a 64-byte packet on a 10 Gbps link takes only ∼50ns, which is comparable to DRAM access time. This precision implies many fine-grained synchronizations during simulation if models are to be accurate.

In addition, as pointed out in [45], the technical challenges of designing datacenters are no less worthy of the expertise of computer systems architects than any other class of machines. Their size alone makes them difficult to experiment with or simulate efficiently, therefore, system designers must develop new techniques to guide design decisions. However, most proposed designs have only been tested with a very small testbed running unrealistic microbenchmarks, often built using off-the-shelf devices [66] that have limitations when exploring proposed new features. The behavior observed by running a test workload over a few hundred nodes bears little relationship to the behavior of production runs completed over thousands or tens of thousands of nodes. The topology and switches used for small test clusters are very different from those in a real environment. Dedicating tens of thousands of nodes to network research is impractical even for large companies like Amazon and Microsoft, let alone academic researchers.

To avoid the high capital cost of hardware prototyping, computer architects have long used software simulators to explore approaches of architectural implementations at all levels, from microarchitectures, and instruction sets to full systems. The relative low cost of implementation and ease of change have made them the ideal choice of early design-space exploration. In addition, when uniprocessor performance was doubling every 18 months, simulation speed correspondingly doubled every 18 months. Unfortunately, the recent abrupt transition to multicore architecture and high-radix switches has both increased the complexity of the system architect wants to emulate. Parallel distributed datacenter applications exhibit more complex behaviors than sequential programs running on a single node. To mitigate this software simulation gap, many techniques have been proposed to reduce simulation time, such as statistical sampling and parallel simulation with relaxed synchronization. These techniques assume the workload is static and independent of target architecture, but

datacenter networks exhibit highly dynamic target-dependent behavior, as they are tightly coupled with computation servers running very adaptive software networking stacks. Last but not least, the scale of a datacenter renders software full-system simulations prohibitively slow to use in practice [120].

1.1.3 Our approach: FPGA models

We argue that even architecture research for single-node systems is now facing a crisis in computer system evaluations using software simulators. Indeed, we found the instructions architect simulated per benchmark significantly decreased over the past decade in academic conferences [120]. Due to the target complexity and scale, traditional cycle-level simulators have fallen far behind the performance required to support running full datacenter software stack for O(100) seconds, which is typically used in datacenter network flow dynamics studies.

To address the above evaluation issues, we propose a novel evaluation methodology using Field-Programmable Gate Arrays (FPGAs), treating datacenters as whole computer systems with tightly integrated hardware and software. Although FPGAs have become an excellent platform to implement new datacenter switches, limited by the FPGA capacity and slow clock rate implementing all datacenter components on FPGAs directly suffers from many practical issues, such as overall build cost and relative timing not reflecting those of real hardware. Instead of prototyping everything on FPGAs, we build realistic reconfigurable abstracted performance models at the scale of O(10,000) nodes. Each node in the testbed is capable of running real datacenter applications on a full operating system. In addition, our network elements are modeled in great detail and heavily instrumented. This research testbed allows us to record the same behaviors administrators observe when deploying equivalently scaled datacenter software.

The testbed is cost-effective compared to building a real equivalent-size datacenter. Although the raw simulation performance is three orders of magnitude slower compared to a real datacenter, it still fast enough to run the datacenter software stack for O(100) seconds. On the other hand, to the best of our knowledge, the platform can scale to the size of real datacenters at a practical cost that none of existing evaluation approaches can reach. The projected full-system hardware cost of a O(10,000) system using the state-of-the-art FPGAs costs around $120K, as opposed to build a real datacenter with $36M in CAPEX and $800K in OPEX/mo. In section 6.6, we show how to construct a 10,000-node system model from several low-cost FPGA boards connected with multi-gigabit serial links.

Our approach make it plausible for many research groups to own their private platform. This opens up the opportunity to conduct research that would never be possible on a production system, such as software reliability experiments at scale that require taking down a large number of nodes. We make the following contributions in this dissertation.

1.2 Contribution

A taxonomy of Computer Architecture model Execution

In this project, we started by looking at the general computer system evaluation methodologies used by architecture community. We introduce the terms *Software Architecture Model Execution (SAME)* and *FPGA Architecture Model Execution (FAME)* to label the two approaches to simulations in section 3.2. Due to the rapid progress made by the whole FAME community over the past few years, there appears to be considerable confusion about the structure and capability of FAME simulators in the broader architecture community. We propose a four-level taxonomy of increasingly sophisticated FAME levels to help explain the capabilities and limitations of various FPGA-based evaluation approaches. Also, we identify three binary dimensions to characterize differences of FAME. The four-level of classification is analogous to different RAID levels. Higher FAME levels lower the simulator cost and improve performance over the lower levels, while moving further away from the concrete RTL design of the simulation target. In addition, the taxonomy sets the technology foundation for cost-efficient datacenter scale simulations, built on top of higher level FAME simulators (e.g. FAME-7) on low-cost single-FPGA boards.

RAMP Gold FPGA-based Multicore Architecture Simulator

We presented the detailed design of RAMP Gold simulator in section 3.3, a very efficient FAME architecture simulator tailored for early design-space exploration for a single-node computer. RAMP Gold employs a decoupled timing/functional simulation architecture with an FPGA-optimized multithreaded implementation. Despite running at a slow clock rate on an FPGA compared to ASIC implementation, RAMP Gold achieves two to three orders of magnitude speedup compared to state-of-the-art software. Besides improving simulation capacity, the greatly improved simulation performance enables various forms of architecture experiments that require longer run time, especially hardware and software co-designs. RAMP Gold supports a standard SPARC v8 ISA and originally targets shared memory multicore architecture simulation. It is capable of running the full Linux operating system and open source datacenter software through cross-compilation, making it an ideal workload generator for networking infrastructure simulations.

FAME-7 DIABLO (Datacenter-in-a-Box at Low Cost) Simulator

Adopting the FAME simulation technologies, we build a reconfigurable datacenter testbed called *DIABLO (Datacenter-in-a-Box at Low Cost)* on top of a modified RAMP Gold design (section 3.4). Although real commercial datacenter networking gear is extremely complicated and details are not publicly available, we build performance models with higher-level abstrac-

tions on well-known architecture fundamentals with some observations from our communications with industry on real datacenter usage scenarios. The models also support changing architectural parameters such as link bandwidth, delays and switch buffer size without time-consuming FPGA resynthesis. We also show the FAME approach is more promising simulating future high-radix switches at high link speed compared to the traditional SAME simulation.

As a proof of concept, we used DIABLO to successfully reproduce some well-known datacenter networking phenomena, such as the TCP incast [126] throughput collapse. We also ran a large memcached caching service [25] on DIABLO with at scale of 2,000 nodes while having full visibility at the interconnect through hardware performance counters, which is very hard to achieve without a high cost for small research groups using commercial switches. In addition, showing the versatility of our approach in helping early architectural explorations in a provocative design space in datacenter networking, we applied an early version of DIABLO to evaluating a novel network proposal based on circuit-switching technology [124] in Section 5.1, running application kernels taken from the Microsoft Dryad [80] Terasort program.

Lessons and Experiences Learned building DIABLO

In this project, we show that designing large-scale FAME simulators is a dramatically different exercise from prototyping the simulated machine itself both in capital cost and development efforts. The simulator itself is a realistic specialized computer that runs a complex software stack. Given different optimization goals and the base FPGA technology, it is very challenging to build a reliable fully functional system. Moreover, in order to debug the basic functionality we need to build specialized hardware to help pinpoint operating system and application issues. We discuss all lessons learned in section 6. Although FAME has great performance, the large development effort is perhaps the biggest valid concern. We present these practical issues in detail and suggest some potential solutions. DIABLO focuses on datacenter networking infrastructure and could be potentially applied to study other datacenter components, such as storage. However, there are several practical modeling limitations. We address them in our future work in section 8. To further lower the cost and increase the simulation capacity, we did a paper design of an ideal FPGA board in section 6.6.

Chapter 2

Related Work

In section 2.1, we first briefly discuss existing evaluation methodologies for datacenter hardware and software innovations. Then in section 2.2, we look at frameworks people use to evaluate datacenter networking proposals. We discuss hardware platforms, software, and workloads people used in the experiments. Since we view datacenters as warehouse-size computer systems, we discuss existing evaluation methodologies of general computer system in section 2.3. We summarize and group the related work based on their basic modeling technologies for the target system: hardware prototyping, software simulation, and FPGA-based performance modeling.

2.1 Evaluation Methodologies for Hardware and Software Innovations in Datacenter

In industry, the best way to test new hardware is to build the real system at a smaller scale and put it along with the production system. For example, Facebook put prototypes of their 100 Gbps Ethernet implementation along with their production cluster [67]. In this way, new hardware evaluation will benefit from running production software and a more realistic workload. However, this approach suffers from the scalability issue. First, it is very expensive to build prototypes in large quantity. Second, it is not practical to have a large-scale deployment of experimental hardware in a production environment. Third, the majority of the testing workloads are generated by old computers. It is therefore difficult to see how well the new computers will perform at scale.

Another popular approach to test novel hardware is deploying the test equipment in medium-scale testing cluster. Usually, these testing clusters are shared by both research and development. For instance, the Yahoo M45 [8] cluster has 4,000 processors and 1.5-

petabytes of data, which is designed to run data-intensive distributed computing platforms such as Hadoop [18]. Big internet companies, like Google could even afford a testing cluster at a much larger scale, e.g. 20,000 nodes [10]. Although these test clusters have enough computing nodes to help diagnose interesting scalability problems, their construction costs are enormous. Bringing up an equivalent production software stack is also another practical issue [10]. Because of cost, researchers in academia use much smaller in-house clusters at the scale of 40 to 80 nodes (or one or two racks) [105, 73, 127].

Recently, cloud computing platforms such as Amazon EC2 offer a pay-per-use service based on virtual machine technology to enable users to share their datacenter infrastructure at an O(1,000) scale. Researchers can pay $\$ < 0.1/hour$ per node to rapidly deploy a functional-only testbed for network management and control plane studies [53, 139]. The cloud is a straightforward approach for software developers to acquire a large-scale infrastructure. Such services, however, provide almost no visibility into the network and have no mechanism for accurately experimenting with new switch architectures.

2.2 Evaluations of Datacenter Networking Proposals

In recent years, many researchers have proposed novel datacenter network architectures [73, 74, 81, 102, 121, 122, 66, 127, 137, 56, 99] with most of them focusing on new switch designs. Many of the evaluations employ the following technologies.

Software routers with synthetic workload and production traffic traces

Software modular routers, such as Click [86], allow easy, rapid development of custom protocols and packet forwarding operations in software. These kernel-based packet forwarding modules can operate at up to 1 Gbps but cannot keep up with 10 Gbps hardware line speed or higher. To compensate for the poor single-thread software performance, RouteBricks [63] and PacketShader [76] leverage parallelism in commodity multicore processors or GPUs to achieve software-based packet processing at high speed. However, these parallel software implementations still face the limited I/O bandwidth available on host servers, which limits the total number of line cards or switch ports that can be modeled. Given the limited performance of software modeling, these switch models are often used with synthetic workloads or traces collected from an old cluster network.

Small-scale clusters with commodity switches

As mentioned in the previous section, using $< O(100)$ node clusters is an affordable common configuration across most of the new proposals from academia. To emulate more computation nodes, researchers run a few thousand virtual machines on a limited number of physical hosts, usually less than 100 nodes [99]. Such time-shared testbeds resemble the multitenant infrastructure of cloud computing, but suffers from many issues arising from aggressive sharing of hardware resources such as I/O bandwidth and physical NIC buffers, introducing non-deterministic performance behaviors. Even worse, in order to achieve a larger scale of O(1,000), researchers run an order of magnitude more VMs on a single physical host than in a real cloud computing environment. In order to emulate more switches, researchers use the VLAN feature on commodity switches to partition a large switch into several smaller virtual switches [66], at the cost of fewer ports, smaller port buffers, and a shared switching fabric.

Small-scale clusters with Openflow switches

OpenFlow [96] is based on an Ethernet switch with an internal flow-table that has a standardized interface to add and remove flow entries. OpenFlow allows researchers to run experiments at line rate on commodity switches, and is an excellent platform to test and deploy new control plane features. However, there are still several limitations of this approach:

1. There are not many OpenFlow-capable array and datacenter-level switches on the market, especially at a high link speed, such as 40 Gbps or 100 Gbps.

2. It is still very expensive to build a large-scale network. Researchers still need to connect enough real machines to generate an interesting workload. The OpenFlow switch by itself does not solve any cost-related issues. Although there are some academic implementations of OpenFlow switches using subsidized university FPGAs board such as NetFPGA [101], given FPGA resource constraints they support a limited number of ports and are not capable of modeling high-radix switches.

3. Currently OpenFlow is only available for Ethernet and packet-switching networks. It is arguable whether the TCAM-based flow table with the extra OpenFlow overhead is suitable for future datacenter networks at a higher link speed [122].

Table 2.1 summarizes evaluation methodologies in recent network design research, in terms of the scale of testbed and workload. Clearly, the biggest issue is evaluation scale. Although a mid-size datacenter contains tens of thousands of servers and thousands of switches, recent evaluations have been limited to relatively tiny testbeds with less than 100 servers and 10–20 switches. Small-scale networks are usually quite understandable, but results obtained may not be predictive of systems deployed at large scale.

Network Architecture	Testbed	Scale	Workload
Policy away switching layer [81]	Click software router	Single switch	Microbenchmark
DCell [75]	Commercial hardware	~20 nodes	Synthetic workload
Portland (v1) [34]	Virtual machine+commercial switch	20 switches+16 servers	Microbenchmark
Portland (v2) [102]	Virtual machine+NetFPGA	20 switches+16 servers	Synthetic workload
BCube [74]	Commercial hardware+NetFPGA	8 switches+16 servers	Microbenchmark
VL2 [73]	Commercial hardware	10 servers+10 switches	Microbenchmark
Helios [66]	commercial hardware	24 servers + 5 switches	Microbenchmark
c-Through [127]	commercial hardware	16 servers + 2 switches	Microbenchmark + datacenter applications (VM migration, MapReduce, MPI FFT)
NetLord [99]	Virtual machine + commercial switch	74 servers + 6 switches	Microbenchmark
Thacker's container network [122]	Prototyping with FPGA boards	-	-

Table 2.1. Datacenter network architecture proposals and their evaluations

For workloads, most evaluations run synthetic programs, microbenchmarks, or even pattern generators, while real datacenter workloads include web search, email, and Map-Reduce jobs. In large companies, like Google and Microsoft, researchers typically use trace-driven simulation, due to the abundance of production traces. Nevertheless, production traces are collected on existing systems with drastically different network architectures. They cannot capture the effects of timing-dependent execution on a new proposed architecture.

Finally, many evaluations make use of existing commercial off-the-shelf switches. The architectural details of these commercial products are proprietary, with poor documentation of existing structure and little opportunity to change parameters such as link speed and switch buffer configurations, which may have significant impact on fundamental design decisions.

2.3 Simulation and Prototyping of General Computer System

In this section, we do a survey of two main evaluation methods of computer system: *hardware prototyping* and *software simulation*.

2.3.1 Hardware Prototyping

Hardware prototyping has a long history, reaching back to the very first computers built in Universities, such as the Harvard Mark-I and EDSAC, but is much less common today. In the 1980s, many researchers would build prototype chips to provide evidence of the value of their architectural innovations. For example, the case for RISC architectures was substantially strengthened by the prototype RISC chips built at Berkeley and Stanford, which ran programs faster than commercial machines despite being produced by small academic teams. Although time consuming and labor intensive to construct, hardware prototypes allow evaluation on much larger and longer programs than possible with software simulators and prevent designers from neglecting the real implementation consequences of their proposed mechanisms. As feature sizes have shrunk, architectural complexity has grown. Consequently, the engineering skill, design effort, and fabrication cost required to build a compelling hardware prototype have risen to the point where few researchers now contemplate such a project. Even when prototypes are successfully completed, the quality of implementation is often inferior to commercial designs, e.g., the TRIPS chip [112] ran at 260MHz versus multiple gigahertz.

Given the increasing expense of development of prototype chips, some researchers investigated building prototypes using FPGAs. The good news is that such designs can be accurate at the Register Transfer Level (RTL), even being written in the same Hardware Description Languages (HDL) used for hardware prototypes. The speed of the logic implemented in

FPGA is relatively slow compared to the speed of DRAMs, so the time to execute a program on an FPGA prototype may not easily predict performance on real hardware, even adjusting for clock rates. Still, design at this level gives implementation insight not available from pure simulation. More importantly, it allows longer runs on real software, albeit 5 to 10 times slower than a hardware prototype. State-of-the-art FPGAs are equipped with dozens of high-speed multi-gigabit transceivers, which make them an excellent platform for prototyping datacenter networking equipment. FPGA-based prototyping has already been used by several academia and industry research projects [101, 122].

2.3.2 Software Simulation

As feature sizes shrunk, fewer researchers could afford to demonstrate their inventions with believable hardware prototypes using state-of-the-art technology, and so software simulation increased in popularity over time. Initially, software simulations employ simple abstracted performance models with static workload models or traces collected from existing hardware. While application traces would still work for some research like branch prediction, they are no longer sufficient to perform software and hardware co-studies.

The increasing complication of target systems led to more complicated performance models and use of real applications. Execution-driven simulations, which simulate the internal details of target systems, have gained popularity as the standard evaluation platform by the computer architecture community. Assuming software and applications change slowly or are rather static, researchers use collections of one-or two-dozen old programs, such as SPEC [77] and PARSEC [47], as reasonable benchmarks for architectures of the future. Although these benchmark suites changed every 3 to 4 years to reduce gamesmanship by compiler writers and architects, new programming models and applications were largely ignored.

Early software simulators only modeled user-level activity of a single application without operating system support. One popular example was RSIM [106], which provide detailed models of out-of-order superscalar processors connected via coherent shared memory. Later, the SimOS project demonstrated how to run an operating system on top of a fast software simulator [110].

Since the speedup in uniprocessor performance did not mask the increase in architectural complexity, such full system simulators offer multiple modes of execution. The fastest mode, generally called *functional simulation*, simply executes instructions without simulating the underlying microarchitecture. Usually, simulators use dynamic binary translation [133], or Xen-based [41] virtual machine technology to speed up target instruction emulation. This mode helps simulated applications to achieve native performance running on the simulator host machine. Once the simulated program reaches the section of interest, the simulator enters a slow cycle-by-cycle mode, called *timing simulation* to simulate all architecture details with user-developed execution-driven performance models, such as GEMS [95]. One popular example of such a simulator is a commercial product, called Simics [94], which allows researchers to study large application programs and the operating system running together. However, due to the limited size of the architecture simulation market, commercial full-

system simulators are dropping the interfaces to plug in user performance model in favor of server virtualization, focusing more on performance of the fast-mode functional simulation.

A straightforward way to reduce simulation time is simply to reduce the size of the input data set so that the program runs in less time. For instance, SPEC offers multiple sizes, from training sets to full size. Some researchers found the training set too long to simulate, so they created "mini-SPEC" to reduce execution time even further [83].

Another popular technique to reduce simulation time is *sampling.* Given the fast forward and snapshot in mixed-mode simulations, researchers developed statistical models that could safely execute many fewer instructions by taking small samples, for example SimPoint [116]. But serious questions remain for the practical application of sampling for tightly coupled multiprocessor and networked systems.

First, to greatly reduce simulation time, the sampling system has to support flexible microarchitecture-independent snapshots, which remain an open research problem for multiprocessors [42, 130]. Without snapshots, simulation time is dominated by the *functional warming* needed to fast-forward between sample points. Some hardware-accelerated simulator, such as ProtoFlex, was developed, in part, to accelerate functional warming [54].

Second, for parallel networked systems, program behavior depends on architectural details, for example shared hardware resources and dynamic code adaptation. One good example is the simulation of TCP/IP workloads. The TCP protocol has a feedback mechanism to tune the sender's transmission rate based on packet loss in the network. To accurately model packet loss, it requires detailed processor and switch timing models. However, in order to use sampling and fast-forwarding, people use much simpler models to quickly advance machine state to a point of interest in the workload, at which time more detailed models take over. The switchover from simple models to detailed models is equivalent to swapping out a high-performance machine for a far slower system, which has a completely different behavior when retuning TCP transmission rate. Some research shows that using sampling to simulate TCP will not reproduce the system's actual steady-state behavior even in the simplest TCP/IP networking environments [78]. Therefore, people developed software simulators with detailed processor, and network I/O models, such as the M5 simulator [48], and use them to study the performance of networked systems with the ability of running real operating system as well as application code.

Finally, it is not clear that fast accurate sampling simulators and accurate statistical workload models are easier to develop and modify than a brute-force FPGA-based simulator. Moreover, similar to the aforementioned TCP simulation issue, it is questionable whether traces or workload models built on top of older systems could faithfully capture dynamics on the current system.

2.3.3 FPGA-Based Performance Modeling and FPGA Computers

FPGAs have become a promising vehicle for architecture experiments, providing a highly parallel programmable execution substrate that can run simulations several orders of magni-

tude faster than software [128]. Previously, FPGA processor models were hampered by the need to partition a system across multiple FPGAs, which increases hardware costs, reduces performance, and significantly increases development effort. On the other hand, FPGA capacity has been scaling with Moore's Law. Nowadays, depending on complexity, multiple processors, and even the whole system can fit into a single FPGA. Furthermore, future scaling should allow FPGA capability to continue to track simulation demands as the number of cores in target systems grows. Due to high volumes, FPGA boards have also become relatively inexpensive. Multiple groups have now developed working FPGA-based computers or simulators to conduct "What if?" experiments.

One obvious use of FPGAs is to build computations directly on the FPGA fabric, where FPGAs are the final target technology. For example, FPGAs are used to build computers to run real production compute workloads, such as Convey HC-1 [15]. Research processors can also be prototyped using FPGAs at much lower cost, risk, and design effort compared to a custom chip implementation [117, 103, 87, 125]. Although an FPGA research prototype bears little resemblance to a custom chip in terms of cycle time, area, or power, it can yield valuable insights into the detailed implementation of a new architectural mechanism, as well as provide a fast platform to evaluate software interactions with the new mechanisms. Simple FPGA cores like Microsoft Beehive [125], are simple enough to be easily understood and modified, yet are powerful enough to run large programs written in C or high-level languages like C#.

Another use of FPGAs is to build functional verification models or accelerators for software simulations. Protoflex [54] is an FPGA-based full-system simulator without a timing model, and is designed to provide similar functionality to Simics [94] at FPGA-accelerated speeds. ProtoFlex employs multithreading to simulate multiple SPARC V9 target cores with a single host pipeline but lacks a hardware floating-point unit as it targets commercial workloads like OLTP; its performance thus suffers on arithmetic-intensive parallel programs. Another example of a FPGA/software hybrid simulator is FAST [52, 51] , which uses a speculative execution scheme to allow a software functional model to efficiently run in parallel with a decoupled timing model in an FPGA. HAsim [57] is an FPGA-based simulator that decouples target timing from functionality, but models a more complex out-of-order superscalar processor.

Chapter 3

Software and FPGA-based Simulation Methodology

Computer architects have long used software simulators to explore instruction set architectures, microarchitectures, and approaches to implementation. In this chapter, we survey the evolution of simulators as architectures increased in complexity and argue that architecture research now faces a crisis in simulation because of the new requirements and the consequences of the multicore revolution. We label the two paths forward in multicore simulation as Software Architecture Model Execution (SAME) or FPGA (Field-Programmable Gate Array) Architecture Model Execution (FAME). Inspired by the five-level RAID classification, we present four levels of FAME that capture the most important design points in this space. Our hope is these FAME levels will help explain FPGA-based emulation approaches to the broader architecture community. In addition, the FAME taxonomy set the technology foundation of the DIABLO design. At the end of this chapter, we present the design of major components in DIABLO to demonstrates FAME capabilities. In this chapter, we concentrate on FAME models of a single datacenter target instance, such as one server and one switch. In the next chapter, we will present how to scale up this single FAME model up to a whole datacenter.

3.1 Software Architecture Model Execution: SAME

A modern computer system running an application workload is a complex system that is difficult to model analytically, yet building a prototype for each design point is prohibitively expensive. Software simulators have therefore become the primary method used to evaluate architecture design choices. We call the machine being simulated the *target* and the machine

on which the simulation runs, the *host*. In this section, we present a brief chronological review of the evolution of software simulator technology.

3.1.1 In-order Processor Simulators

Much of the architecture research in the 1980s involved in-order processors, and popular topics were instruction set architectures (RISC vs. CISC), pipelining, and memory hierarchies. Instruction set architecture research involved compilers and instruction set simulators to evaluate novel instruction sets. Instruction-set simulators were very slow, and so pipelining and memory hierarchy studies relied on address traces to drive simulators that only simulated the portions of the computer of interest. Even if they were expensive to create, traces could be reused many times in the exploration of ideas. Given the common needs of researchers, some generous researchers would share the trace they collected along with the simulators with the community, for instance cache simulators, such as the popular Dinero tool [1]. In contrast, pipeline hazard simulators tended to be created for each study. At the time, most of simulations studied only single user programs without operating systems or multiprogrammed workload.

3.1.2 Out-of-Order Processor Simulators

Enthusiasm shifted from instruction set architecture research and in-order processors to out-of-order processors in the 1990s. Popular topics included alternative microarchitectures, branch predictors, and memory hierarchy studies. Although traces were still satisfactory for simpler branch prediction research, they were no longer adequate for microarchitecture or memory hierarchy studies due to the complex out-of-order nature of the processors. For example, architects could no longer calculate the impact on performance of a miss by simply adding it as a factor to a CPI calculation, since other instructions could overlap execution during a miss, and a branch misprediction could cause the processor to begin executing instructions not present in the trace.

This complication led to the popularity of execution-driven simulators, which model the internal microarchitecture of the processor on a cycle-by-cycle basis. Unfortunately, the speedup in host uniprocessor performance did not match the increase in target architectural complexity, so the simulation of each instruction took more host clock cycles to complete. In addition, Moore's Law enabled much larger and more elaborate memory hierarchies, so architects needed to simulate more instructions to warm up caches properly and to gather statistically significant results. Moreover, as computers got faster and had more memory, programs of interest became bigger and ran longer, which increased the time to simulate benchmarks.

The complexity of building an execution-driven simulator, together with the increasing use of commercial instruction sets, common benchmarks, and a common research agenda led to the development of shared execution-driven simulator models, of which SimpleScalar is

surely the most widely used example [40]. Architects would either use the standard options provided by the simulators or make modifications to the simulator to explore their inventions. Once again, interesting free simulators let many architects perform the type of research that the simulator supported.

3.1.3 Multiprocessor Simulators

Simulating parallel target machines is much more difficult than simulating uniprocessors. Part of the added complexity is simply that the target hardware is more complex, with multiple cores and a cache-coherent shared memory hierarchy. In addition, a parallel software runtime must be present to support multithreading or multiprogramming across the multiple cores of the simulated target, which adds additional complexity. For multiprocessor research, trace-driven simulation is still often used despite the inability of traces to capture the effects of timing-dependent execution interleaving, due to the difficulty of developing a full system environment capable of running large workloads. As with uniprocessor simulators, many parallel simulators only modeled user-level activity of a single application.

Both academic projects and commercial products, such as SimOS [110] and Simics [94], demonstrated how to run an operating system on top of a fast software simulator for multiprocessor simulation. These simulators supported multiple levels of simulation detail, and the fastest version used dynamic binary translation to speed target instruction emulation while emulating cache hierarchies in some detail [133]. These full-system simulators have become a popular research tool in the architecture community, augmented with detailed performance models developed by academic researchers [95].

Although techniques such as dynamic binary translation, virtualization, and trace-driven simulation help with the interpretation of the functional behavior of each instruction, they do not help with the considerably more compute-intensive task of modeling microarchitecture details of the processor and memory hierarchy. Surprisingly, it is difficult to parallelize detailed target multiprocessor simulations to run efficiently on parallel host machines. The need for cycle-by-cycle interaction between components limits the parallel speedup possible due to the high synchronization costs in modern multiprocessors. If this cycle-by-cycle synchronization is relaxed, parallelized software simulators can attain some speedup but at the cost of needing to validate that the missing interactions do not affect the experiment's results [97, 109].

3.1.4 The Software Premise of Simulation

Implicit in many of the techniques used to reduce simulation time (traces, reduced inputs sets, sampling) was the assumption that software changes slowly and is independent of the target architecture. Thus, suites like SPEC with one- or two-dozen old programs were reasonable benchmarks for architectures of the future, as long as SPEC suite changed every four to five years to reduce gamesmanship by compiler writers and architects. Similarly,

research groups would create their own benchmark suites, and some of these became popular, for example, SPLASH and SPLASH 2 [134] for multithreaded programs. Parsec is a modern example of such a parallel benchmark suite [47].

New programs, new programming models, and new programming languages were ignored [65]. The argument was either that architects should wait for them to prove their popularity and become well-optimized before paying attention to novel systems–which could take a decade–or that there was nothing you would do differently for them than you would for old programs written in the old programming languages. Since benchmarking rules prohibited changing the program, architects generally treated the programs as static artifacts to measure without understanding either the problems being solved or the algorithms and data structures being used.

3.1.5 The Multicore and Cloud Revolution

As has been widely reported, the end of ideal technology scaling together with the practical power limit for an air-cooled chip package forced all microprocessor manufacturers to switch to multiple processors per chip [38]. The path to more client performance for such *multicore* designs is increasing the number of *cores* per chip every technology generation, with the cores themselves essentially going no faster. On the server side, cloud computing lets users tap into the power of massive warehouse-scale computers to run their applications at enormous scales.

Given this new client+server revolution, the biggest architectural research challenges now deal with multiple computing nodes, for example multiple processors or servers, rather than increasingly sophisticated single nodes. Hence, architecture investigation now needs to able to look at many system components in addition to memory hierarchy and processor design. The number of cores per chip, sophistication of these cores, and even the instruction sets of the cores are all open to debate. Issues that have received little recent attention, like both on-chip and off-chip interconnect, are vital. Moreover, old application programs and operating systems are being rewritten to be compatible with a massively parallel environment. New programming models, new programming languages, and new applications are being invented at an unprecedented speed. In particular, some of the new internet applications are being developed and improved along with the production environment in a very short period. For example, it takes only six months for Facebook to develop the Timeline feature from nothing to production [23]. Given these urgency of challenges, we draw the following conclusions that architects should not ignore:

1. Given software churn, new techniques like autotuning, new issues like temperature and power, and the increasing number of cores, architecture research is going to require simulation of many more target instructions than it did in the uniprocessor era.

2. Given the natural non-determinism of parallel programs, to be confident in results, architects need to run programs many times and then take the average, which again increases the number of instructions that should be simulated [35].

3. Given our lack of intuition about how new programs will behave on novel hardware and new metrics like power and absolute time (instead of clock cycles), well need to simulate this greater number of instructions at a greater level of detail than in the past.

4. Given this lack of understanding, we also need to run experiments for all portions of the system and with added instrumentation, since we don't yet know what components or metrics can be safely ignored when proposing novel architectures.

Multiplying these four requirements together suggests an upsurge in the demand for simulation by many orders of magnitude above what can be done with traditional software simulators today. To put into perspective how many instructions are actually being simulated per processor using software simulators today, Table 3.1 compares the number of instructions simulated per paper from the 2008 International Symposium on Computer Architecture (ISCA) to the same conference a decade earlier. Recent papers simulate many more total instructions and cores if you compare medians, but the number of instructions per core was just 100 million in 2008 vs. 267 million in 1998. We might assume that the authors didn't simulate more instructions because they did not need more to generate accurate conclusions from their experiments. However, we see no evidence of the dramatic rise in simulation time that we argue is needed for the multicore challenge. In fact, it is heading in the other direction: these numbers correspond to about 0.05 seconds of target execution time in 2008 vs. about 0.50 seconds in 1998.

As mentioned above, the performance challenge for software simulators is turning the increasing number of host cores into higher simulated target instructions per second. We believe the challenge will be far harder for detailed simulation than for functional simulation, as there is naturally much more communication between components in a target cycle. While we encourage others to make progress on this important but difficult problem, we are more excited by an alternative approach.

3.2 Three FAME dimensions and Terminologies

FPGAs have become a promising vehicle for architecture experiments, providing a highly parallel programmable execution substrate that can run simulations several orders of magnitude faster than software [128].

Multiple groups have now developed working FPGA-based systems [52, 58, 103, 87, 54], but using perhaps an even greater variety of techniques than software simulators, and correspondingly a wider range of tradeoffs between simulator performance, accuracy, and flexibility. Consequently, we have heard much confusion in our discussions with other architects about how FAME relates to prior work using FPGAs for architecture prototyping and chip simulation, and what can and cannot be done using FAME.

In this section, we first present three binary dimensions within which we can categorize FAME approaches. Next, inspired by the five-level RAID classification, we present four

Table 3.1. Number of instructions simulated, number of processors, and instructions simulated per processor for ISCA 2008 vs. 1998.

ISCA 2008				
	Total Instructions	Cores	MInstructions/Core	Programs
	150M	16	9	SPEC2006
	50M	4	12	SPEC2000
	240M	16	15	SPLASH-2
	500M	16	31	Traffic Patterns
	650M	16	40	SPEC2000
	2300M	23	72	STAMP+SPLASH
	100M	1	100	SPEC2000
	100M	1	100	SPEC2000
	1600M	16	100	SPEC2000
	1000M	8	125	SPLASH-2, SPECJBB
	2500M	16	160	Hashtable, Rbtree
	1000M	4	250	MinneSPEC
	1000M	1	1000	MinneSPEC
	35000M	20	1750	SPEC2000
Median	825M	16	100	
ISCA 1998				
	Total Instructions	Cores	MInstructions/Core	Programs
	100M	1	100	SPEC95
	100M	1	100	SPEC95
	100M	1	100	SPEC95, NAS, CMU
	100M	1	100	SPEC95
	171M	1	171	SPEC95, SPLASH-2
	200M	1	200	SPEC95
	236M	1	236	SPEC95
	267M	1	267	SPEC95
	267M	1	267	SPEC95
	267M	1	267	SPEC95
	267M	1	267	SPEC95
	325M	1	325	SPEC95
	860M	1	860	OLTP/DB, SPEC95
	900M	1	900	OLTP/DB
	1000M	1	1000	Synthetic
	84000M	8	10500	DASH/SimOS
Median	267M	1	267	

levels of FAME that capture the most important design points in this space. Our hope is these FAME levels will help explain FPGA-based emulation approaches to the broader architecture community.

3.2.1 FAME Implementation Techniques

We use the following three binary dimensions to characterize FAME implementation approaches.

Direct vs. Decoupled

The *Direct* approach is characterized by the direct mapping of a target machine's RTL description into FPGA gates, where a single target clock cycle is executed in a single host clock cycle. An advantage of the direct approach is that, in theory, a re-synthesis of the target RTL for the FPGA provides a guaranteed cycle-accurate model of the target processor. The direct approach has been popular in chip verification; one of the first uses of FPGAs was emulating a new chip design to catch logic bugs before tapeout. Quickturn [62] was an early example, where boxes packed with FPGAs running at about 1-2 MHz could run much larger test programs than feasible with software ECAD logic simulators. Direct emulation is also often used by intellectual property (IP) developers to supply a new core design to potential customers for evaluation and software porting before committing to an ASIC.

Direct emulation has become much easier as the growth in FPGA capacity reduces the need to partition monolithic RTL blocks, such as CPUs, across FPGAs, but large system designs may still require many FPGAs. The inefficiency of FPGAs at emulating common logic structures—such as multiported register files, wide muxes, and CAMs—exacerbates capacity problems.

A more powerful FAME option, which improves efficiency and enables other more advanced options, is to adopt a *Decoupled* design, where a single target clock cycle can be implemented with multiple or even a variable number of host clock cycles [70]. For example, direct mapping of a multi-ported register file is inefficient on FPGAs because discrete FPGA flip-flops are used to implement each register state bit with large combinational circuits used to provide the read ports. A more efficient decoupled model would implement a target multi-ported register file by time-multiplexing a single-ported FPGA RAM over multiple FPGA clock cycles. The drawback of a decoupled design is that models have to use additional host logic to model target time correctly, and a protocol is needed to exchange target timing information at module boundaries if modules have different target-to-host clock cycle ratios [70, 57].

Full RTL vs. Abstracted Machine

When the full Register-Transfer Level (RTL) of a target machine is used to build a FAME model, it ensures precise cycle-accurate timing. However, the desired RTL design is usually not known during early-stage architecture exploration, and even if the intended RTL design is known, it can require considerable effort to implement a working version including all corner cases. Even if full correct RTL is available, it may be too unwieldy to map directly to an FPGA.

Alternatively, we can use a higher-level description of the design to construct a FAME model. Abstraction can reduce both model construction effort and FPGA resource needs. The primary drawback is that an abstract model needs validation to ensure accuracy, usually by comparing against RTL or another known good model. If the mechanism is novel, an RTL prototyping exercise might be required to provide confidence in the abstraction. Once validated, however, an abstract component can be reused in multiple designs.

HAsim [108] was an early example of the *Abstract* FAME option, where the processor model was divided into separate functional and timing models that do not correspond to structural components in the target machine. Split functional and timing models provide similar benefits as when used in SAME simulators. Only the timing model needs to change to experiment with different microarchitectures, and the timing model can include parameters such as cache size and associativity that can be set at runtime without resynthesizing the design, dramatically increasing the number of architecture experiments that can be performed per day.

Single-Threaded vs. Multi-Threaded

A cycle-accurate FAME model synchronizes all model components on every target clock cycle. Some complex components might experience long host latencies, for example, to communicate with off-chip memory or other FPGAs, reducing simulator performance. For processors, a standard approach to tolerate latencies and obtain greater performance is to switch threads every clock cycle so that all dependencies are resolved by the next time a thread is executed [37]. The same approach can be applied when implementing FAME models in a technique we call *host multithreading*, and is particularly applicable to models of parallel target machines.

When the target system contains multiple instances of the same component, such as cores in a manycore design, the host model can be designed so that one physical FPGA pipeline can model multiple target components by interleaving the component models' execution using multithreading. For example, a single FPGA processor pipeline might model 64 target cores or a single FPGA router pipeline might model 16 on-chip routers.

Host multithreading greatly improves utilization of FPGA resources by hiding host communication latencies. For example, while one processor target model makes a request to a memory module, we can interleave the activity of 63 other target processor models. Provided modeling of the memory access takes fewer than 64 FPGA clock cycles, the emulation will not stall. Multithreaded emulation adds additional design complexity but can provide a significant improvement in emulator throughput.

ProtoFlex is an example of a FAME simulator that host-multithreads its functional model [54]. The same concept has also been used in SAME simulators, e.g. later versions of the Wisconsin Wind Tunnel were also host multithreaded [100].

3.2.2 FAME Levels

Level	*Name*	*Example*	*Strength*	*Experiments per Day*	*Experiments per Day per $1000*
000	Direct FAME	Quickturn, Palladium	Debugging logical design	1	0.001
001	Decoupled FAME	Green Flash	Higher clock rate; lower cost	24	0.667
011	Abstract FAME	HAsim	Simpler, parameterizable design; faster synthesis; lower cost	40	40.000
111	Multi-threaded FAME	RAMP Gold	Lower cost; higher clock rate	128	170.000

Table 3.2. Summary of four FAME Levels, including examples.

A combination of these FAME implementation techniques often makes sense. The next four sections present a four-level taxonomy of FAME that improves in cost, performance, or flexibility. The four levels are distinguished by their choices from the three options above, so we can number the levels with a three-bit binary number, where the least-significant bit represents Direct (0) vs. Decoupled (1), the middle bit represents Full RTL (0) vs. Abstracted (1), and the most-significant bit represents Single-Threaded (0) vs. Multi-Threaded (1). Table 3.2 summarizes the levels and gives examples and the strengths of each level. Each new FAME level lowers cost and usually improves performance over the previous level, while moving further away from the concrete RTL design of the target.

To quantify the cost-performance difference of the four FAME levels, we propose as a performance measure the number of simulation experiments that can be performed per day. Given the complex dynamics of manycore processors, operating systems, and workloads, we believe the minimum useful experiment is simulating 1 second of target execution time at the finest level of detail for 16 cores at a clock rate of 2 GHz with shared memory and cache coherence. We employ this as an approximate unit to measure an experiment. The same experiment but running for 10 target seconds is 10 units, the same experiment but running for 1 second at 64 cores is 4 units, and so on. Note that in addition to host simulation time, experiment setup time (e.g. design synthesis time) must also be included. To obtain a cost-performance metric, we simply divide the number of experiments per day by the cost of that FAME system. To keep the numbers from getting too small, we calculate experiments per day per $1000 of the cost of the FAME system. The last column of Table 3.2 estimates this metric for 2010 prices.

Direct FAME (Level 000): (e.g., Quickturn)

The common characteristic of Direct FAME emulation systems, such as Quickturn, is that they are designed to model a single chip down to the gate level with a one-to-one mapping of target cycles to host cycles.

Let's assume we could simulate the gates of 16 cores on a $1 million Direct FAME system at 2 MHz. Each run would then take $2GHz/2MHz = 1000$ seconds or 17 minutes. Because the model is not parameterized, we have to rerun the CAD tool chain for each experiment to resynthesize the design. Given the large number of FPGAs and larger and more complicated description of a hardware-ready RTL design, it can take up to 30 hours to set up a new design [123]. If Direct FAME can do 1 experiment per day, the number of experiments per day per $1000 is 0.001.

In addition to high simulation turnaround time, Direct FAME requires great design effort to change the RTL for each experimental machine, unlike some of the later FAME levels. Although helpful in the later stages of debugging the design of a real microprocessor intended for fabrication, Direct FAME is too expensive and time consuming to use for early-stage architectural investigations.

Newer gate-level emulation products, such as Cadence Palladium and MentorGraphics Veloce, are no longer based on commercial FPGAs but instead use custom-designed logic simulation engines. However, they still have relatively low target clock rates [4] and cost millions of dollars, though the tools are superior to FPGA tools for this purpose.

Decoupled FAME (Level 001):

(e.g., Green Flash memory system)

Programmable logic is slow compared to hardwired logic, and some ASIC features, such as multiported register files, map poorly to FPGAs, consuming resources and cycle time. For example, Green Flash [129] can fit two Tensilica cores with floating-point units per medium-sized FPGA, but it runs at only 50 MHz [115]. The memory system uses off-chip DRAM, however, which runs much faster than the logic (200 MHz) and so decoupling is used in the memory system to match the intended target machine DRAM timing.

Performing a 16-core experiment needs 2 BEE3 [59] boards, which cost academics about $15,000 per board, plus the FPGAs and DRAMs, which cost about $3000 per board, or $36,000 total. It takes 8 hours to synthesize and place and route the design and about 40 seconds ($2GHz/50MHz$) to run an experiment. Since this level has a few timing parameters, such as DRAM latency and bandwidth, Green Flash can run about 24 experiments per synthesis [115]. Alas, the state of FPGA CAD tools means FPGA synthesis is a human-intensive task; only one synthesis can be run per workday. Thus, the number of experiments per day per $1000 is 24/$36K or 0.667.

Decoupled FAME (Level 001) improves the cost-performance over Direct FAME (Level 000) by a factor of almost 700×. This speedup is mostly due to processor cores fitting on

a single FPGA, thus avoiding the off-chip communication that slows Direct FAME systems; also, Decoupled FAME uses a simple timing model to avoid resynthesis for multiple memory system experiments.

It is both a strength and a weakness of Decoupled FAME that the full target RTL is modeled. The strength is that the model is guaranteed to be cycle accurate. Also, the same RTL design can be pushed through a VLSI flow to obtain reasonable area, power and timing numbers from actual chip layout [118]. The weakness is that designing the full RTL for a system is labor-intensive and rerunning the tools is slow. This makes Decoupled FAME less suitable for early-stage architecture exploration, where the designer is not ready to commit to a full RTL design. Decoupled FAME thus takes a great deal of effort to perform a wide range of experiments compared to Abstract and Multithreaded FAME. These higher levels, however, require decoupling to implement their timing models, and hence we assume that all the following levels are Decoupled (or odd-numbered in our enumeration).

Abstract FAME (Level 011): (e.g., HAsim)

Abstract FAME allows high-level descriptions for early-stage exploration, which simplifies the design and thereby reduces the synthesis time to under 1 hour. More importantly, it allows the exploration of many design parameters without having to resynthesize at all, which dramatically improves cost-performance.

Let's assume we need 1 BEE3 board for 16 cores, so the cost is $18,000. To simulate cache coherency, the simulator will take several host cycles per target cycle for every load or store to snoop on the addresses. Let's assume a clock frequency of 65 MHz, as with HAsim [107], and an average number of host cycles per target cycle of 4. The time for one experiment is then $4 \times 2\text{GHz}/65\text{ MHz} = 123$ seconds.

Since human intervention isn't needed to program the FPGAs, the number of experiments per day is 24 hours / 123 seconds = 702. The number of experiments per day per $1000 is then 702/$18K or about 40. Abstract FAME (Level 011) makes a dramatic improvement in this metric over lower FAME levels: by a factor of almost 60 over Decoupled FAME (Level 001) and a factor of 40,000 over Direct FAME (Level 000).

In addition to the improvement in cost-performance, Abstract FAME allows many people to perform architecture experiments without having to modify the RTL, which both greatly lowers the effort for experiments and greatly increases the number of potential experimenters. Once again, the advantages of abstract designs and decoupled designs are so great that we assume any subsequent level is both Abstract and Decoupled.

Multithreaded FAME (Level 111):

(e.g., RAMP Gold)

The main cost of Multithreaded FAME is more RAM to hold copies of the state of each thread, but RAM is one of the strengths of FPGAs — a single programmable logic unit can be exchanged for 64-bits of RAM in a Virtex-5 FPGA. Hence, Multithreaded FAME increases the number of cores that can be simulated efficiently per FPGA. Multithreading can also increase the clock rate of the host simulator by removing items on the critical path, such as bypass paths.

Since we are time-multiplexing the FPGA models, a much less expensive XUP board ($750) suffices. Multithreading reduces RAMP Gold's host cycles per target core-cycle to 1.90 (measured) and enables a clock rate of 90 MHz. Since the simulator is threaded, the time for a 16-core simulation is $16 \times 2GHz/90MHz \times 1.9 = 675$ seconds. The number of experiments per day is 24 hours / 675 seconds = 128. The number of experiments per day per $1000 is then 128/$0.75K or about 170. Multithreaded FAME (Level 111) improves this metric by more than a factor of 4 over Abstract FAME (Level 011), by a factor of about 250 over Decoupled FAME (Level 001), and by a factor of 170,000 over Direct FAME (Level 000).

In addition, Multithreaded FAME lowers the cost of entry by a factor of 24–48 versus Abstract or Decoupled FAME, making it possible for many more researchers to use FAME for parallel architecture research.

Other Possible FAME Levels

By definition, direct mapping cannot be combined with abstract models or multithreading. An RTL design can be multithreaded, however, whereby every target register is replicated for each threaded instance but combinational logic is shared by time multiplexing. We ignored this Multithreaded RTL combination (101) as a FAME level because, although plausible, we have not seen instances of this combination in practice.

Hybrid FAME Simulators

Although we present levels as completely separate approaches for pedagogic reasons, real systems will often combine modules at different levels, or even use hybrid designs partly in FPGA and the rest in software. For example, System-on-a-Chip IP providers will often use a mixed design to provide a fast *in situ* emulation of their IP blocks for customers. The IP block is modeled by mapping the final ASIC RTL to the FPGA (Direct FAME, Level 000), but the enclosing system is described at an abstract level (Abstract FAME, Level 011). FAST [52] is an example of a hybrid FAME/SAME system, where the functional model is in software and the timing model is in hardware.

3.3 RAMP Gold: An example of a full-system FAME-7 simulator

The RAMP Gold simulator was initially designed to run on a single $750 Xilinx Virtex-5 FPGA board, targeting a tiled, shared-memory manycore system. We use a variant of RAMP Gold in DIABLO, described in the following chapter, to model datacenter servers. In this section, we describe the original design of RAMP Gold and how the FAME-7 style simulator attains high efficiency.

RAMP gold is a full-system simulator, which simulates up to 64 single-issue, in-order SPARC V8 cores. It boots the Linux 2.6.39 kernel, as well as ROS [84, 93], a manycore research operating system developed at Berkeley. RAMP Gold's target machine is highly parameterized and most simulation options are runtime configurable: Without resynthesizing a new FPGA configuration, we can vary the number of target cores, cache parameters (size, associativity, line size, latency, banking), and DRAM configuration (latency, bandwidth, number of channels). This extensive runtime parameterization accelerates design space exploration and comes at little cost to simulator performance: with a detailed memory system timing model, we can simulate over 40 million target-core cycles per-second.

3.3.1 RAMP Gold Microarchitecture

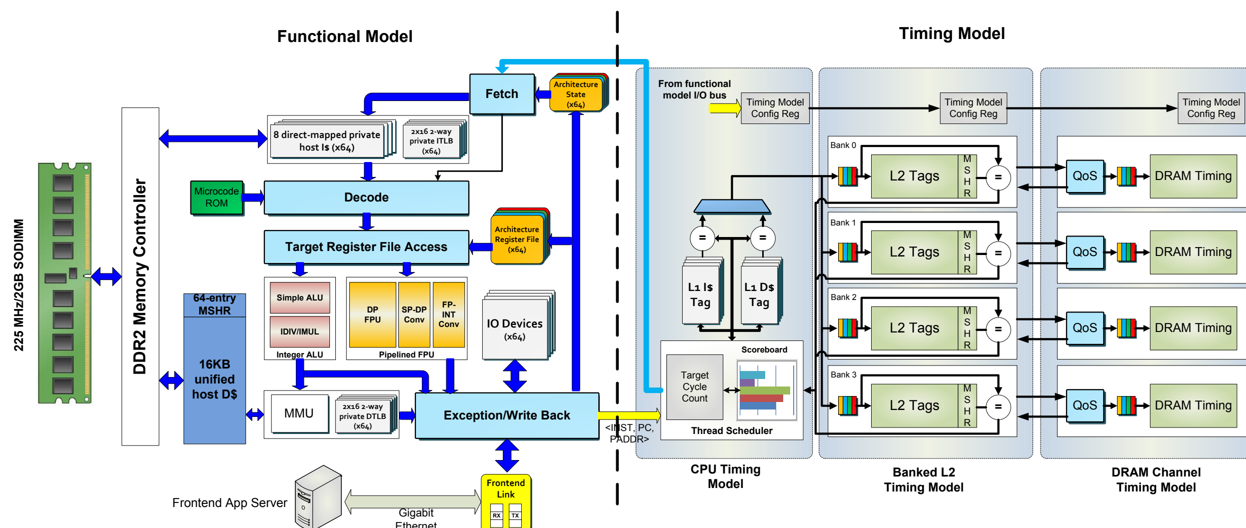


Figure 3.1. RAMP Gold Microarchitecture

Figure 3.1 shows the microarchitecture of RAMP Gold. The simulator decouples timing from function. The functional model faithfully executes the SPARC V8 ISA and maintains architected state, while the timing model determines instruction execution time in the target

machine. Both models reside in one FPGA to minimize synchronization costs, and both are host-multithreaded to achieve high utilization of FPGA resources.

The functional model implements the full SPARC V8 ISA in hardware, including floating-point and precise exceptions. It also provides sufficient hardware to run an operating system, including MMUs, timers, and interprocessor interrupts. The functional model is deeply pipelined, and avoids features such as highly ported register files and wide bypass muxes that map poorly to FPGAs. The functional model carefully exploits Virtex-5 features, for example, double-clocking block RAMs and mapping target ALU instructions to hardwired DSP blocks. The single in-order issue functional pipeline is 64-way multithreaded, enabling functional simulation of 64 target cores. Each thread's private state includes a 7-window register file, a 32-entry instruction TLB and 32-entry data TLB, a 256-byte direct-mapped instruction cache, and the various processor state registers. Although 64 copies of this state seems large, trading state for increased pipeline utilization is attractive in the FPGA fabric, wherein storage is cheap relative to computation.

The threaded functional pipeline has a single, shared, lockup-free host data cache. It is 16 KB, direct-mapped, and supports up to 64 outstanding misses. Sharing a very small host cache between 64 threads is a design point peculiar to the FPGA fabric: the low latency to the host DRAM, approximately 20 cycles in the worst case, is covered easily by multithreading. Thus, the lower miss rate of a large, associative host cache offers little simulation performance advantage. Indeed, across a subset of the PARSEC benchmarks, the small host cache incurs at most a 3.8% performance penalty compared to a perfect cache. (The tiny 256-byte instruction caches have even less of an impact on performance—at most 2.3% worse than a perfect cache.)

The timing model tracks the performance of a target 64-core tiled manycore system. The target core is currently a single-issue, in-order pipeline that sustains one instruction per clock, except for instruction and data cache misses. Each core has private instruction and data caches. The cores share a unified lockup-free L2 cache via a magic crossbar interconnect. Each L2 bank connects to a DRAM controller model, which models delay through a first-come, first-served queue with a constant service rate.

Separating timing from function expands the range of systems RAMP Gold can model, and allows the effort expended on the functional model to be reused across many different target machines. For example, we can model the performance of a system with large caches by keeping only the cache metadata inside the FPGA. The functional model still fetches from its host instruction cache and performs memory accesses to its host data cache when the timing model schedules it to do so. Moreover, the flexibility of splitting timing and function allows us to configure RAMP Gold's timing models at runtime. To model different cache sizes, for example, we fix the maximum cache size at synthesis time, and at runtime we program configuration registers that determine how the cache tag RAMs are indexed and masked. Most timing model parameters can be set at runtime; among these are the size and associativity of the L1 and L2 caches, the number of L2 cache banks and their latencies, and DRAM bandwidth and latency. The current implementation of the timing model runs at 90 MHz on the Virtex-5 FPGA, and supports up to 12 MB of aggregate target cache, while using over 90% of the on-chip FPGA block RAM resources.

Synthesis of RAMP Gold takes about two hours on a mid-range workstation, resulting in 28% logic (LUT) and 90% BRAM utilization on a mid-size Virtex-5 FPGA. The low logic utilization is due in part to mapping computation to the built-in DSPs and by omitting bypass multiplexers. The high block RAM utilization is mainly due to the large target caches we support. More details on the RAMP Gold implementation can be found in [119].

3.3.2 Model Verification and Flexibility

RAMP Gold comprises about 36,000 lines of SystemVerilog with minimal third-party IP blocks. We liberally employ SystemVerilog assertions to aid in RTL debugging and verification. The functional model is verified against the SPARC V8 certification suite from SPARC International. Because it uses abstracted RTL, RAMP Gold requires the same simulator timing verification as SAME simulators, but the far greater performance eases the verification effort. We verify our timing models with custom microbenchmarks.

The timing model and its interface to the functional model are designed to be simple and extensible to facilitate rapid evaluation of alternative memory hierarchies and microarchitectures. Despite its extensive runtime configurability, the timing model comprises only 1000 lines of SystemVerilog. It is thus easy to understand and prototype new architectural ideas. For example, we implemented a version of a novel quality-of-service framework, Globally-Synchronized Frames [89], in about 100 lines of code and three hours of implementation effort.

3.3.3 RAMP Gold Speedup Results

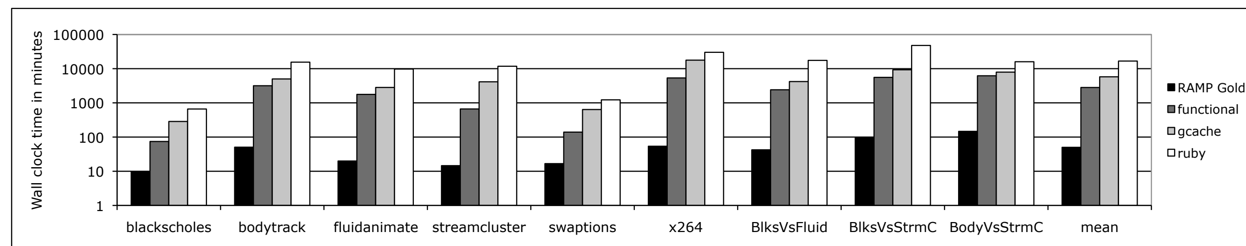


Figure 3.2. Wallclock time of RAMP Gold and Simics simulations. The target machine has 64 cores. Possible Simics configurations are functional modeling only, `g-cache` timing modules, and the GEMS Ruby module, with an interleave of 1 instruction. In the cases where two applications are run, each gets 1/2 of the partitionable hardware resources.

To compare against RAMP Gold's performance, we run the PARSEC [47] benchmarks inside Virtutech Simics [94], a popular SAME simulator. We run Simics with varying levels of architectural modeling detail: pure functional simulation, Simics `g-cache` timing modules, and the Multifacet GEMS [95] Ruby timing module.

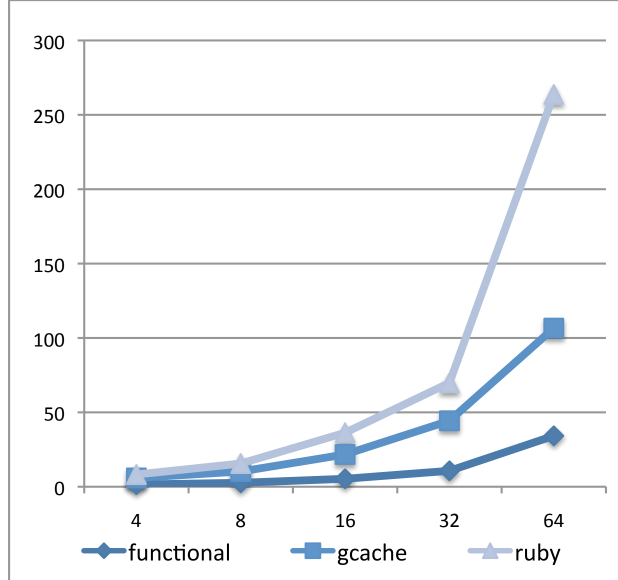


Figure 3.3. Geometric mean speedup of RAMP Gold over Simics across benchmarks. Possible Simics configurations are functional modeling only, `g-cache` timing modules, and the GEMS Ruby module, with an interleave of 1 instruction. The x-axis is target cores.

We configure Simics to model the same case study target machine as closely as possible. However, both `g-cache` and GEMS Ruby modules implement timing for a MESI coherence policy, whereas RAMP Gold does not at present do so. We configure Ruby to not simulate contention in the on-chip interconnection network (neither `g-cache` nor RAMP Gold do so presently).

We vary the number of target machine cores simulated in both RAMP Gold and Simics. The applications spawn as many threads as the target machine has cores, but the workload size is fixed. Simics was run on 2.2 GHz dual-socket dual-core AMD Opteron processors with 4 GB of DRAM. Reducing the frequency at which Simics interleaved different target processors offered a limited performance improvement.

The longest running Simics simulation of a single benchmark point takes over 192 hours (8 days), whereas the longest RAMP Gold simulation takes 66 minutes. Figure 3.2 plots the wall clock runtime of a 64-core target machine simulated by RAMP Gold and different Simics configurations across benchmarks and pairs of co-scheduled benchmarks. RAMP Gold is up to two orders of magnitude faster. Critically, this speedup allows the research feedback loop to be tens of minutes, rather than hundreds of hours.

RAMP Gold runtimes generally improve as the number of cores is increased because multithreading becomes more effective, whereas Simics' performance degrades super-linearly with the number of cores simulated. With 64-core target machines, RAMP Gold is even faster than Simics's functional simulation. Figure 3.3 shows the geometric mean speedup of FAME over SAME across the different benchmarks and for different SAME configurations. The maximum speedup is a factor of 806×.

The slowdowns incurred by Simics are due nearly entirely to host machine performance, as the benchmarks themselves scale in performance across more target cores equivalently on Simics and RAMP Gold. The fact that the slowdowns also correlate with the size of the benchmarks' inputs and working set suggests that host cache and TLB misses may present a major performance bottleneck. Unfortunately, Simics is closed-source, so we were not able to diagnose its poor performance more precisely.

3.4 A FAME-7 Datacenter Switch Model

Another concrete example of FAME-7 is our abstracted simulation model for output-queue datacenter switches. When modeling switches, the real challenges arises from design complexity and proprietary architecture specifications. Using FPGA resource-efficient high-level abstracted FAME models make sense when studying key architecture features. For instance, the biggest differences between existing commercial packet switches are the switch buffer architecture and configurations, which is also an active area for packet switching researchers as well. Therefore, in our FAME model we preserve these architecture details as much as possible, while simplifying other features. We further describe our abstractions and reasoning of the rest of the switch components in the context of a datacenter world in the next chapter.

3.4.1 Switch Model Microarchitecture

Figure 3.4 shows the architecture of our abstracted simulation model for output-queue switches, such as the Fulcrum FocalPoint FM4000 [55]. One of the biggest differences between existing commercial packet switches is the packet buffer size. For instance, the Force 10 S60 switch has 1280 MB of packet buffering, the Arista Network's 7048 switch has 768 MB, and the Cisco Systems' 4948-10GE switch has 16 MB.

Similar to RAMP Gold, to make efficient use of host DRAM burst accesses, we designed a shared host cache connected by a ring-like interconnect to all switch models using the same host DRAM channel. The host cache is composed of two simple buffers, one for write and one for read, partitioned equally among all physical ports. Due to the limited size of on-chip FPGA BRAM, the write and read buffers only hold 64 bytes for every physical port, which is the minimum flit size for many packet switches. In addition, the write and read buffers for each port have a write-lock to ensure they are kept coherent.

Inside each switch model, a key component is a queue management model responsible for all virtual queue pointer operations. It also keeps track of queue status and performs packet drops when necessary. The length of every simulated virtual queue can be configured dynamically without requring another FPGA CAD flow run before a simulation starts. We select these configurable parameters according to a Broadcom switch design [88]. Along with this module, a performance-counter module, implemented with a collection of BRAMs

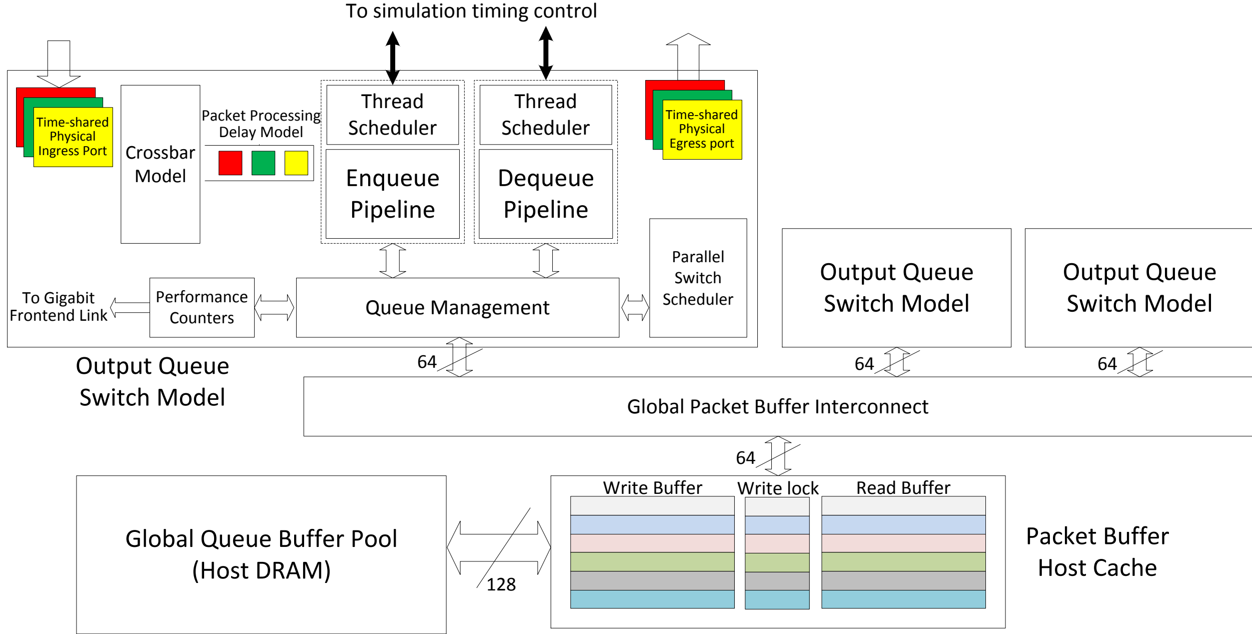


Figure 3.4. FAME model for virtual output queue switches.

and LUTRAMs, maintains all statistics for every virtual queue. The performance counter module reports all of its content periodically to a remote PC through the gigabit frontend link, with which we can construct queue length dynamics offline for every virtual queue running any workload. To send unicast statistics every 6.4 μs in target time, a 10 Gbps 32-port output-queue switch model demands a bandwidth of approximately 40 Mbps on the frontend link.

Each model has an independent 3-stage enqueue pipeline and a 4-stage dequeue pipeline that are controlled and synchronized by global simulation timing control logic. Ideally, the two pipelines send commands to the queue management model through simple FIFOs in order to simplify and decouple the control logic design. However, this appears to be a significant area and performance overhead on FPGAs, consuming a large amount of distributed LUTRAM and making routing very hard. Instead, we implement two independent static thread schedulers for the two pipelines and replay commands that were deferred due to dependencies.

To guarantee good simulation performance, the switch scheduler model processes scheduling decisions for multiple virtual queues in every host FPGA clock cycle. To further improve performance, given the hundreds or thousands of virtual queues existing in our simulated switches, the parallel scheduler model only processes active events that happen between two scheduling quanta instead of naively scanning all virtual queues. Overall, the simulation performance of a single switch model has a slowdown of 150× compared to real hardware. This is four times faster than a software single-threaded network simulator used at Google [10], which does not simulate packet payloads and does not support full software stack scaling to 10,000 nodes as does our system.

3.4.2 Comparing to SAME simulators

The FAME-7 switch model is relatively easy to build, with only around 3,000 lines of SystemVerilog code. Although FAME models allow us to conduct datacenter experiments at enormous scale with greater architecture detail, they do require more design effort. For example, a comparable cycle-accurate software packet-switch model only requires 500 lines of C++ code.

As a comparison, we also optimized and parallelized the equivalent C++ simulation model using *Pthreads*. We compiled the C++ module using 64-bit GCC4.4 with '-O3 -mtune=native -march=native', and measured the software simulator in a trace-replay mode with the CPU cache already warmed up. We ran the software simulator on an 8-Core Intel Xeon X5550 machine with 48 GB memory running the latest Linux 2.6.34 kernel.

Figure 3.5 shows slowdowns of the multithreaded software model simulating different size 10 Gbps switches under two types of workload, i.e. full load with 64-byte packets and random load with random-size packets. When simulating a small 32-port switch, the single-thread software model has better performance than our threaded 100 MHz FAME-7 FPGA model. However, the simulation performance drops quickly when increasing the number of switch ports. Due to many fine-grained synchronizations (approximately every 50ns in target time), software multithreading helps little when simulating small switches.

When simulating a large switch configuration, we saw small sublinear speedups using two, or sometimes four, threads but the benefit of using more threads diminishes quickly. Profile results show that crossbar scheduling, which scans multiple virtual queues, accounts for a large fraction of the total simulation time. Other large overheads include cache misses for first time accesses to virtual queue structures as well as updating in-memory performance counters for each simulation quanta. On the other hand, CPU memory bandwidth is not at all a limiting factor, even when simulating a large switch configuration. Moreover, Figure 3.5 also illustrates that the workload significantly affects the simulation performance for large switch configurations.

Note that we measured the software simulation performance under an unrealistic setting. The sole reason to use a random workload here is to generate reasonable amount of traffic that will fully exercise the switch data path for simulator host-performance comparisons. In a real usage scenario, switch traffic will be generated dynamically by other models connected to the switch, which requires many more synchronizations over the input and output ports of the simulated switch. When simulating a large system containing many switches and servers, we believe it will be difficult to see any performance benefit by partitioning the software model across a high-performance cluster. Besides, future datacenter switches are very likely to be high-radix switches. Simulating architectures in even greater detail could also easily render the software approach impractical.

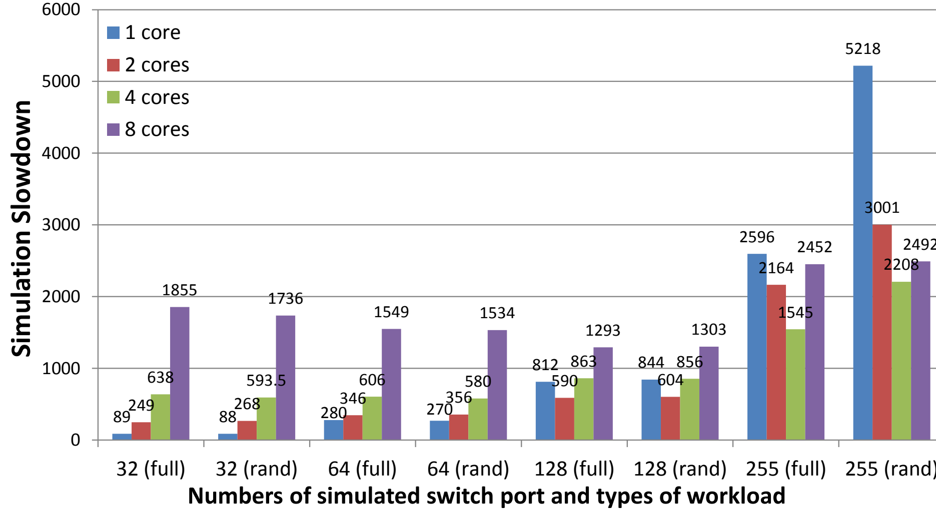


Figure 3.5. Parallel software switch simulation performance.

3.5 Conclusions

In this chapter, we show the importance of increasing the simulation performance for the architecture research community. To clarify the many efforts at using FPGAs to deliver that performance boost, we propose a four-level taxonomy for FPGA Architecture Model Execution (FAME). By estimating experiments per day per dollar, we show improvements in cost-performance by factors of 200,000 between the lowest and highest FAME levels. We use RAMP Gold, which simulates 64 SPARC on a single FPGA, and a 32-port output-queue 10 Gbps switch model to demonstrate the benefits of the highest level. Due to the intrinsic fine-grained synchronization nature of architecture simulation, we also show that even with the help of multicore the Software Architecture Model Execution (SAME) simulator offers limited performance improvement simulating a highly paralleled target. Although SAME simulators achieve comparable performance simulating target with a smaller configuration, high-level FAME simulators outperform by orders of magnitude for larger target configurations.

Chapter 4

Building a Scalable DIABLO for Datacenter Network Architecutre Simulations

In this chapter, we describe our implementation of high-throughput datacenter simulator using FPGAs, which is a dramatically different exercise from prototyping the simulated target itself. In the following sections, we first discuss our design strategies of DIABLO. Then, we cover the DIABLO model abstractions, and the FPGA implementations for three major datacenter components: server, switch, and network interface card. Finally, we show how to use 11 BEE3 boards to scale up DIABLO to simulate 3,968 servers with a packet switching interconnect.

4.1 DIABO Design Strategy

The most intuitive approach of implementing a datacenter simulator is to build a "mini-datacenter" using FPGAs. For instance, mapping servers and interconnects directly onto FPGAs, using soft-core processor and switch implementations. Naively mapping these components to FPGAs, however, is inefficient, inflexible, and impossible to model key datacenter architecture features at scale. DIABLO's efficient design is based on several observations that distinguish it from other FPGA-based simulators and system implementations:

- *FPGAs don't implement complex combinatorial logic well, such wide multiplexers.* Any logic is mapped with a number of multi-input lookup tables (LUTs). Complex com-

binatorial logic occupies a large number of lookup tables in many levels. Although modern FPGAs have plenty of logic resources, routing resources remain to be a critical component. The more logic levels the more routing delays. This observation led to simpler DIABLO microarchitecture designs avoiding complex combinatorial logic. For instance, our processor core utilized an unbypassed pipeline design that avoids wide forwarding-path multiplexers. We found by removing forwarding logic in a popular FPGA soft-core processor [11], pipeline area is reduced by 26%-32% and frequency is boosted by 18%-58%.

- *FPGAs have plenty of RAM.* This observation combined with the lack of bypass paths, led to a multithreaded design of all large modules. Simulation performance arises from many simulation threads per FPGA rather than from complex simulation pipelines optimized for single-thread performance. We use RAMs on FPGAs to store simulation thread state, and dynamically switch threads to keep simple module pipelines saturated. This strategy, we called *host-multithreading* in earlier chapters, uses multiple threads to simulate different target. Note that host-multithreading neither implies nor prohibits a multithreaded target architecture design.

- *Modern FPGAs have hard-wired DSP blocks.* Execution units, especially FPUs, dominate LUT resource consumption when implementing a processor on an FPGA. If we map functional units to DSP blocks rather than just LUTs, we can devote more resources to timing simulation

- *DRAM accesses are relatively fast on FPGAs.* Logic in FPGAs often runs slower than DRAM because of on-chip routing delays. This insight greatly simplifies DIABLO's host memory system, as large, associative caches are not needed for high performance. This observation led to modeling large target buffers, such as packet-switch port buffers in DRAM, with minimum simulation performance impact.

- *FPGA primitives run faster but have longer routing delays.* FPGA primitives, such as DSPs and BRAMs, run at high clock rates compared to random logic, but their fixed on-die location often exacerbates routing delays. This observation led to a deep model pipeline. Given that DIABLO is a throughput optimized system with a feed-through data path design, we could easily insert pipeline stages to boost host FPGA clock frequency. Longer pipeline bubbles can be filled with more hardware threads. In addition, we can double-clock these FPGA primitives to have shorter access latency and more read/write ports on the primitives.

Like many software simulators [40, 64], DIABLO separates the modeling of target timing and functionality. The *functional model* is responsible for executing the target software correctly and maintaining architectural state, while the *timing model* determines the time the target machine takes to run an instruction. The benefits of this functional/timing split are:

- *Simplified FPGA mapping of the functional model.* The separation allows complex operations to take multiple host cycles. For example, a highly-ported register file can

be mapped to a block RAM and accessed in multiple host cycles, avoiding a large, slow mapping to FPGA registers and muxes.

- *Improved modeling flexibility and reuse.* The timing model can be changed without modifying the functional model, reducing modeling complexity and amortizing the functional model's design effort. For instance, we can use the same switch functional model to simulate both 10Gbps switches and 100Gbps switches, by changing only the timing model.
- *Enable a highly-configurable abstracted timing model.* Splitting timing from function allows the timing model to be more abstract. For example, a timing model might only contain target cache metadata. Different cache sizes could then be simulated without resynthesis by changing how the metadata RAMs are indexed and masked at runtime.

4.2 Server Models

The goal of the server model is to have a credible workload generator that drives more detailed networking models. Given constant datacenter software churns and short development cycles, building highly-accurate analytical models for the workload is a less practical approach. Therefore, the server model must be capable of running complex server application software with minimum modifications.

4.2.1 Mapping to FPGAs

We build the server models on top of the RAMP Gold FAME-7 simulator with a heavily-modified host-cache design to better support running multiple Linux kernels from different hardware threads. The server model supports the full 32-bit SPARC v8 ISA in hardware, including floating-point instructions and precise exceptions. It also models sufficient hardware to run an operating system, including MMUs, timers, and interrupt controllers. The functional model adopts a 64-thread feed-through RAMP Gold pipeline. We map one server to one hardware thread. One 64-thread RAMP Gold hardware pipeline simulates up to two 32-server datacenter racks. Each simulated server uses a simplified fixed-CPI timing model. A more detailed timing model could be implemented, but it would reduce simulation scale as each server model would require additional host hardware resources. The interface between the functional and timing models is designed to be simple and extensible to facilitate rapid evaluation of alternative target memory hierarchies and microarchitectures.

The functional model has been highly optimized for the Xilinx Virtex-5 FPGA fabric, and employs the following mapping optimizations:

- *Routing-optimized pipeline:* The functional pipeline is 13 stages long. Some pipeline stages are dedicated to signal routing to BRAMs and DSPs.

- *Microcode for complex operations:* The functional/timing split allows us to implement the functional pipeline as a microcode engine. Complex SPARC operations, such as atomic memory instructions and traps, are handled using microcode in multiple pipeline passes. The microcode engine also makes it easier to prototype extensions to the ISA.

- *DSP-mapped ALU:* DSP blocks in FPGAs have been greatly enhanced in recent generations to support logical operations and pattern detection, in addition to traditional multiply-accumulate operations. We mapped the integer ALU and flag generation to five Virtex-5 DSPs and the FPU to fourteen DSPs.

- *Simple host cache and TLB:* Each thread has a private direct-mapped 256-byte instruction cache and coherent data cache, a 32-entry instruction TLB, and a 32-entry data TLB. The host caches and TLBs have no effect on the target timing — they exist solely to accelerate functional simulation. The unusually small host data cache is a deliberate, albeit peculiar, design decision that we discuss in the next section.

- *Fine-tuned block RAM mappings:* The server functional model is a BRAM-dominated design. In the functional model, the register files, host caches, and TLBs are manually mapped to BRAMs for optimal resource usage. In addition, we double-clocked all BRAMs for higher bandwidth. Each BRAM is protected by either ECC or parity for longer and larger scale experiments that require many FPGAs.

4.2.2 Host Memory Interface Design

The memory subsystem of a datacenter is one of the most heavily engineered components in the architecture to improve performance and energy efficiency. Similarly, the memory subsystem is a critical component for building DIABLO performance model. In particular, the functional host-cache of the server model receives great attentions during the design time. As mentioned earlier, an FPGA is a different design space compared to ASIC, where the memory access is very "fast". Therefore, we focus our attention of the functional host-cache design on correctness and design simplicity.

Initially, DIABLO adopted an unmodified RAMP Gold host cache design. The key feature of the host-cache is its small size and simple architecture. The instruction cache is physical-indexed per-thread independent direct-mapped with a small size of merely 256 bytes. The cache line size is 32-byte that matches the DRAM burst size of our DDR2 memory controller. Since the cache size is smaller than a memory page, the pipeline can perform instruction TLB lookups concurrently with the instruction cache look up to reduce the pipeline depth. The host data cache is slightly more complicated in the architecture, which is a 16 KB physical-indexed directly-map non-blocking cache shared by all 64 hardware threads using a simple direct-mapped 64-entry MSHR. The shared host data cache easily provides data coherency among all threads for multicore workloads. Another criterion of choosing the cache size is how well the host cache can be mapped to block rams on FPGA.

We choose the minimum number of block rams that satisfies the bandwidth requirement of one-cycle cache access for a single cache line, given the 32-byte DRAM burst constraint.

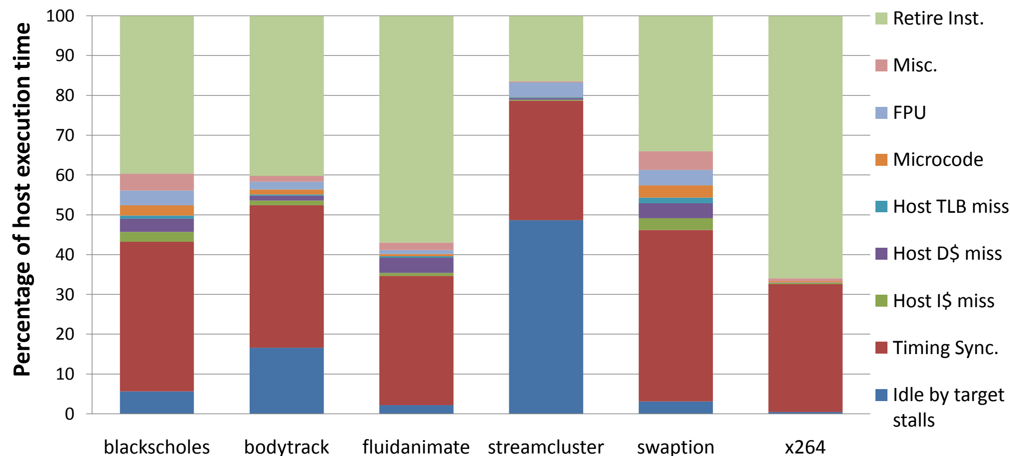


Figure 4.1. Host cycle breakdown for RAMP Gold running PARSEC.

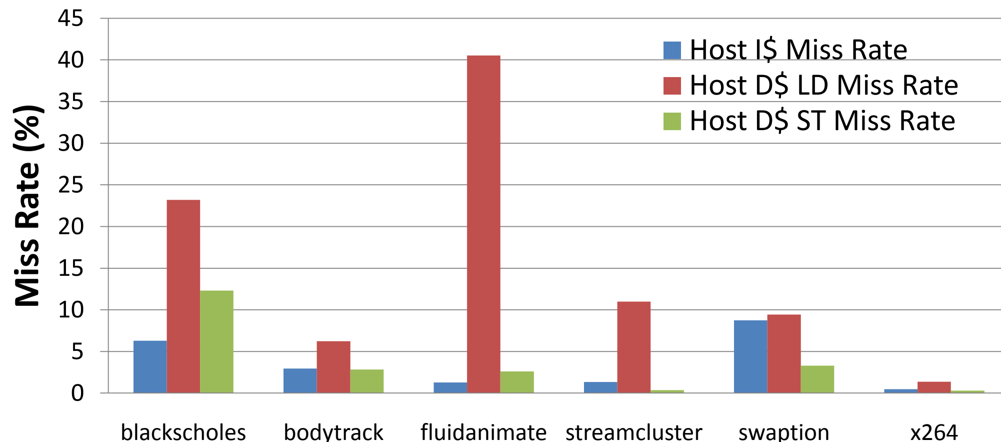


Figure 4.2. RAMP Gold Host Cache Miss Rate running PARSEC

To quantify the effectiveness of this simple host cache design and performance impact of pipeline 'replays' caused by various sources in the host, such as host cache/TLB misses, we added several host performance counters. Figure 4.1 plots the detailed host cycle breakdown running the PARSEC benchmarks with 64 target cores on the ROS research OS in using the original RAMP Gold multicore timing model. Figure 4.2 shows the host cache miss rate of the small host cache running the benchmark. Unsurprisingly, the small host cache results in high miss rate. Nevertheless, host cache misses collectively account for no more than 6% of host clock cycles. DRAM's relatively low latency - about 20 clock cycles - and the ability of multithreading to tolerate this latency are largely responsible for this apparent contradiction. Thus, rather than spending BRAMs on large, associative host caches, we can dedicate these resources to other models. Nevertheless, providing a small cache is still valuable to exploit the minimum 32-byte DRAM burst size. We also measured host DRAM

bandwidth utilization, and found we never exceed 15% of the peak bandwidth, indicating that a single-channel memory system is sufficient for these benchmarks.

Interestingly, the most significant overhead is timing synchronization: not until all instructions from a given target cycle have retired do we begin instruction issue for the next target cycle. We expect most of this overhead can be recovered by more efficient thread scheduling. The functional pipeline is also idle when target cores are stalled. Streamcluster, for example, has a high target cache miss rate. The functional model is thus underutilized while target stalls are modeled. Other causes of replay include host TLB misses, floating-point operations, integer multiplication and division, and three-operand store instructions. Collectively, these account for less than 10% of host cycles across these benchmarks.

Although the original RAMP Gold host data cache is coherent among all hardware threads, it is not coherent with the instruction cache. When we repurpose RAMP Gold to datacenter emulation and run independent Linux instances in every hardware thread, we face many software correctness challenges posed by the incoherent instruction and data host-cache. For an in-house developed research operating system, such as ROS, we can easily modify the source code to support coherency through proper cache invalidations in the OS kernel. However, real operating systems have bugs and a great deal of legacy code, which makes it hard to run correctly without a coherent cache. We discuss this software experience in details in a later chapter.

As illustrated above, a small host cache has a very limited simulation performance impact on FPGAs. We can therefore choose the simplest coherent implementation, a coherent store buffer like those in early SPARC chip implementations that only holds one cache line of store data. The coherency is easily achieved by invalidating corresponding lines in the instruction cache. In terms of the actual implementation, we map this store buffer to FPGA BRAMs, whose depth gives us more entries. Thus, we design a per-thread partitioned direct-map 8-line host data cache implementing a simple invalidation-based coherency scheme. Our double-clocked BRAM mappings provide us extra access ports on the cache RAM to process coherent traffic without stalling normal cache accesses from pipeline stages.

Adding the coherency does increase the design complexity in terms of FPGA resources quite a bit, especially in the host instruction cache shown in Table 4.1. However, the control logic is relative easy to implement compared to a non-blocking shared cache on a feed-through pipeline. Besides, it improves simulation performance while booting multiple Linux operating systems by having fewer contentions from different host hardware threads.

Component Name	*LUT*	*Register*	*LUTRAM*
Host D $	1,370 (-3.6%)	1,775 (67.3%)	20 (0%)
Host I $	740 (318.0%)	717 (27.8%)	64 (-)

Table 4.1. The control logic resource consumption of the DIABLO coherent cache on Xilinx Virtex 5 LX110T. The increased areas over the RAMP Gold unified cache are shown in parentheses in each cell.

4.2.3 Debugging Infrastructure

To communicate with the simulator, we embedded a microcode injector into the functional pipeline, which we connect to a front-end Linux PC via a gigabit Ethernet link. This front-end link doubles as a debugging and control interface: we use it to start and stop simulations, load programs, and to modify or checkpoint architectural states without affecting the target timing.

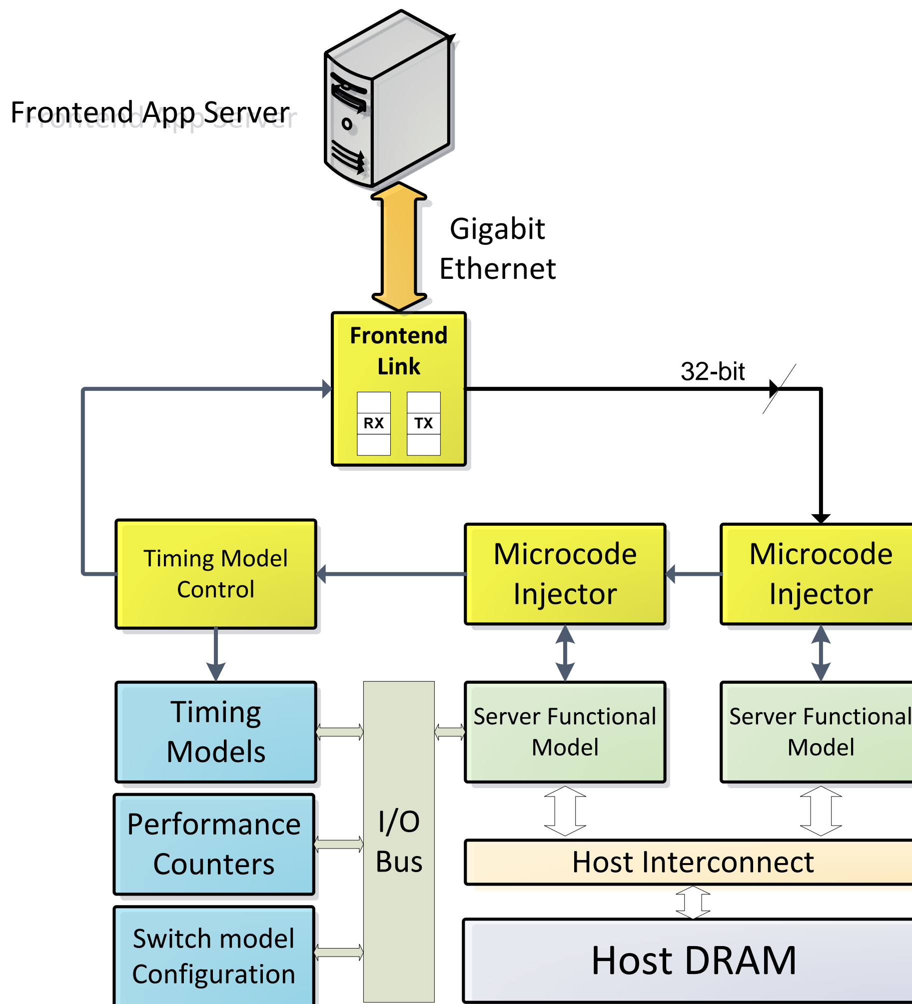


Figure 4.3. The DIABLO frontend link architecture. We show only two microcode injectors on the ring for illustration purposes only.

Figure 4.3 shows the architecture diagram of the frontend link. We design a single 32-bit token-ring interconnect running at the same clock frequency as the server model pipeline. Each server model pipeline has its own microcode injector sitting on each ring station. Using a ring reduces routing congestion substantially, allowing the chip to support more DIABLO pipelines. The Ethernet controller servers as the master node of the ring, which runs in

a different clock domain and is decoupled with the core logic using asynchronous FIFOs. When there is free space in the asynchronous FIFO, it emits a single token into the ring. As the token passes each node, the node can choose to pass it on immediately if it has no traffic, or to add some number of payload that follow the token. The token mechanism works in a similar fashion as the processor interconnect ring in Beehive [125], which can be best analogous to a train system that starts at a main station and passes through a number of local stations before arriving back at the main station. The "main station" (Ethernet) sends a "locomotive" (the token) with zero cars. At each local station, the "station" will add couple of "cars" (payload) to the end of the train. When the train reaches the main station, all the cars will be unloaded (payload is transmitted) before the locomotive starts again. Since the microcode injector works in a polling mode, i.e. the frontend PC initiates any request, we could use a simple software-controlled credit-based flow control mechanism to prevent Ethernet controller data FIFO from overruns.

Similarly, to configure, debug, and collect runtime performance of other DIABLO modules, we implement simple memory mapped I/O devices on the server model pipeline, such as hardware performance counters and timing model controls. The front-end link also allows us to forward I/O functional requests to the Linux control PC due to lacking of hardware models on FPGAs. For example, currently DIALBO does not have a FAME model for hard drives. However, real server software often requires dynamically loaded libraries from a local disk. Therefore, we implement a simple block-based virtual disk driver in the Linux kernel, which forwards all system disk I/O requests over the Ethernet front-end link to the front-end Linux PC, which provides a functional local disk storage. Reusing the same microcode injector, the functional disk also works only in a polling mode. The performance is limited by the Ethernet communication latency between the frontend PC and the DIABLO hardware. To ensure functional correctness, We also put a checksum unit in the Linux driver that verifies checksums for each block. To sum up, the goal is not to provide a fast and reliable emulated disk with accurate target timing but rather a basic functionality of a disk to run the server software.

The original reason of embedding the microcode injector into the processor pipeline is to make the coherency story easier between the processor host-cache and the target DRAM. This approach avoids many potential functionality issues updating simulation states in the simulated memory as well as in connected I/O devices through the frontend link, while the simulation is actively running. The additional benefit of implementing a microcode injector is that we can use this hardware to easily access more detailed architecture states in the processor pipeline, such as register files and host-cache content. It also helps to collect pipeline checksum and instruction traces when debugging the processor. However, in practice, at the datacenter level, such fine-grained debugging features are rarely used. On the other hand, since the microcode injector has to interact with the processor pipeline, it complicates the pipeline control design.

4.2.4 Server software

We run full-fledged Linux on each node, as opposed to a reduced version of Linux like uClinux, often seen in embedded platforms. We also support dynamically linked applications with full Glibc. Being SPARC v8 compliant, the DIABLO server model can run user binaries acquired from the Debian GNU/Linux repository without the trouble of cross-compiling SPARC binaries on an x86 host. This is also one of the main reasons we chose the SPARC ISA. The 64-bit implementation of the ISA, i.e. SPARC v9, is still actively developed by Oracle/Sun in production for the enterprise server market. The Debian SPARC port supports sun4u (Ultrasparc) and sun4v (Niagara) machines with a 32-bit user land. Although the Debian SPARC port uses a 64-bit kernel in production, most of the applications run in 32-bit mode. Due to significant memory and disk overhead, Debian maintains a 64-bit port as an add-on for the current port for applications that really benefit from being in 64-bit mode. Consequently, building a simpler 32-bit implementation is sufficient to run many user applications for DIABLO.

The heavy-lifting work of supporting many SPARC applications is all about adding a DIABLO CPU support in the Linux Kernel. All kernel modifications are based on the most up-to-date version we could get at the beginning of the development, which is 2.6.39.3. Since the 32-bit SPARC port is considered as stable, we believe the same porting work can be easily applied to a more recent 3.x kernel. In addition, implementing the SPARC reference MMU (SRMMU) greatly simplifies our kernel porting efforts.

When developing the DIABLO port, we devoted most of our energies to two aspects:

1. Build a bootloader that supports the SPARC OpenPROM.
2. Modify corresponding kernel code to allow a proper MMU and processor detection.

We also hacked the TLB/Cache flush interface to support the host cache/TLB architecture on our CPU. Just like supportting a new CPU in the kernel, we added our own console and interrupt controller drivers. Moreover, we wrote a block-based disk driver to interact with our frontend server to provide functional disk access. Similarly, we designed a keyboard driver that allows users interactively access the simulated servers in DIABLO like accessing a real machine.

Although the overall kernel porting process appears to be straightforward for adding a new CPU support to an existing ISA, one big practical issue is that we found quite a few kernel bugs. Changing the kernel source code can fix some bugs, such as those in the SRMMU code and those in the kernel spinlock implementation triggered by a recent version of GCC. On the other hand, some bugs are very harder to fix directly with software approaches, which results in changing the hardware instead. We describe this experience in chapter 6.

To improve productivity, we also developed a C-functional simulator that runs the same SPARC binaries on a general x86 host machine. We use this functional simulator extensively verifying the application functionality before deploying to the DIABLO FPGA hardware. We could also run a native SPARC GCC inside the functional simulator to eliminate the need of cross-compiling of many server software.

4.3 Switch Models

There are two categories of datacenter switches: *connectionless packet switching*, also known as *datagram switching*, and *connection-oriented virtual circuit switching*. In the first case, each packet includes complete routing information, and is routed by network devices individually. The second case requires a pre-allocated virtual circuit path before transferring any packet. To demonstrate the flexibility of our approach, we build FAME-7 models for both types of switches.

In order to provide more predicted latencies and take advantage of new high-speed switching technologies, researchers propose new designs of circuit-switching switches for datacenters. The proposed circuit-switching switches are still in early research prototypes, which are simple and open in their implementations. Some of the switches [124] are directly implemented on FPGAs. Therefore, it is straightforward to build highly-accurate models for these circuit-switching switches. As an application of DIABLO, we describe modeling a research circuit-switching network in the following chapter.

On the other hand, the real challenges for modeling the packet switches used in existing production datacenters arise from design complexity and proprietary architecture specifications. To work around these barriers, we build abstract models by simplifying features that are seldom used in a datacenter. Here are the abstractions we employed and the rationale behind our choice:

- Ignore Ethernet QoS related features (e.g. support of IEEE 802.1p class of service (CoS)): Although QoS features are available on almost every switch today, many datacenters only utilize switches for basic connectivity without turning on QoS features.
- Use simplified source routing: Many packet switches use a large ternary CAM to hold flow tables and look up the destination address for each packet. When an unknown MAC address is seen, the forwarding engine sends an interrupt to a slow-path control processor that updates the table using software. Many switches [16, 17] already support flow tables that have at least 32K entries. Given the total number of machines in datacenters, the slow-path flow-table update is rarely executed, making the TCAM lookup time constant in practice. Besides, datacenter topologies do not change frequently, and routes can be pre-configured statically. We use source routing to simplify modeling of packet routing, and we note that source routing is actually a component of many datacenter-switch research proposals. To emulate more complicated flow table operations, we could implement d-left hash tables [98] using host DRAM. Recent datacenter switches that implement large flow tables, such as the Cisco Nexus 5000, use similar techniques instead of TCAMs.
- Abstract packet processors: Commercial datacenter switches include many pipelined packet processors that handle different tasks such as MAC address learning, VLAN membership, and so on. The processing time of each stage is relatively constant regardless of packet size, and the time can be as short as a few hundred nanoseconds [55] to a few microseconds [16]. We simply employ FIFOs with runtime-configurable delays to model packet processing.

Although commercial switch implementation details are generally not publicly available, the fundamentals of these switch architectures are well known. Examples include the architecture of a virtual-output-queue switch and common scheduling algorithms. We build our abstracted model focusing on these central well-known architectural features, and allow other parts that are unclear or of special interest to researchers (For example, packet buffer layout) to be configurable during simulation.

Specifically, in our switch models, we focus on the data path features, such as switch buffer management and configurations, which have become an active area for packet switching researchers according to our conversations with datacenter networking researchers in industry. As network guru Andy Bechtolsheim has recently pointed out [46], datacenter network protocols and applications need to be tuned together to better understand underlying network's capacity, which is a nontrivial task. At the meantime, the datacenter network switch design is all about right sizing buffers. For example, some of the new data networking protocols such as DCTCP [36] can improve the network performance but requires mega-buffers on aggregation switches.

Another important feature for simulation is to model high link bandwidth and cut-through switching fabrics for low port-to-port latencies. These are essential features to evaluate future high performance datacenter switches, which are very hard to deploy at scale in real world because of cost issues.

While there has been an increasing research interest in the area of Software Defined Network (SDN) using Openflow-capable switches in datacenter, researchers put more emphases on simplifying the switch control software rather than solving data path performance issues of traditional packet switching network. The data path architectures of these SDN switches are actually identical to that of conventional switches. As mentioned above, our simplified source-routed switch models can be easily modified to support the traditional flow table architecture used by SDN switch.

The basic architecture of existing commercial datacenter switches fall into two categories, output-queue and combined input-output queue with virtual output queue. The former has no on-chip congestion, and minimum buffering latencies. Therefore, it is the ideal memory architecture for switches. However, this architecture requires that all switch ports can simultaneously read/write into the shared buffer memory, which demands enormous bandwidth. This requirement poses a significant challenge to switch chip architects designing novel on-chip SRAM architectures, such as Fulcrums RapidArray shared memory [55]. It limits the scalability of the switch in terms of number of ports, per-port bandwidth, as well as the size of the packet buffers.

The combined input-output queue architecture uses separate egress and ingress memory structures to reduce the memory bandwidth requirement at the cost of a more complex internal switching arbitration design. The input queues employ Virtual-output-Queue (VoQ) like structures to eliminate the issue of head-of-line (HOL) blocking. Many Cisco datacenter switches adopt this architecture. Combined input-output queue switches have lower performance compared to an ideal output-queue switch. Nevertheless, the basic queuing architectures of either input or output queues are similar.

In DIABLO, we use an abstracted output-queue switch model with a simple round-robin scheduler described in the previous chapter for all level of switches: top of the rack switches, array, and datacenter level switches. The differences between switch models of different layers in the network hierarchy are link latency, bandwidth, and buffer size. We present this switch model architecture in the previous chapter.

4.4 Network Interface Card Models

The DIABLO NIC models an abstracted Ethernet device, whose internal architecture resembles that of the Intel 8254x Gigabit Ethernet controller used by the popular Intel PRO/1000 MT server adapter. Many software virtual machines also emulate a full or part of the Intel 8254x Ethernet controller, such as Virtual Box [28], VMWare Virtual Server [32], QEMU [29], and Microsoft Hyper-V [26].

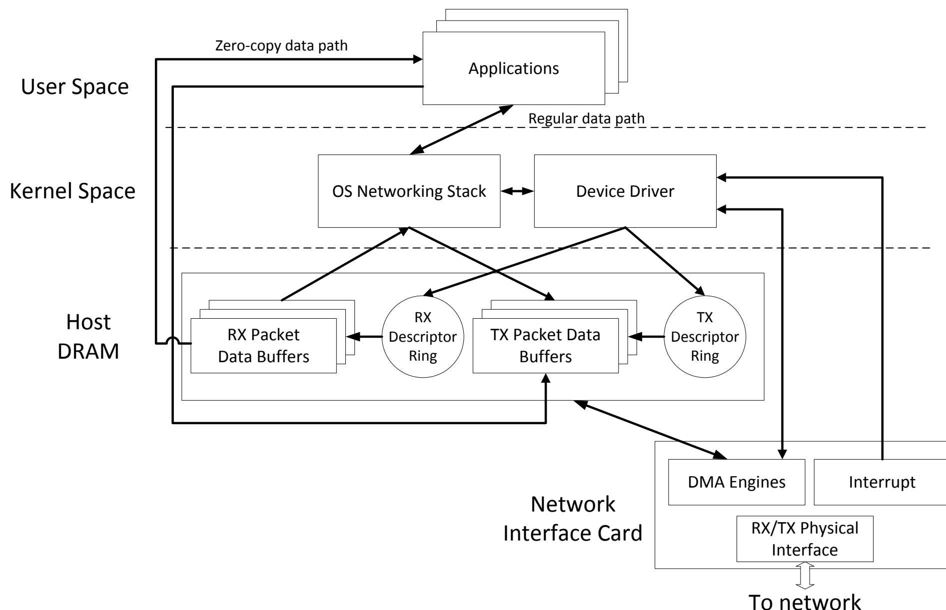


Figure 4.4. Software and hardware architecture of generic network interface cards

Figure 4.4 shows the target architecture of our abstracted NIC model. The core feature of the NIC is a scatter/gather DMA with ring-based packet buffers stored in the main system DRAM. The scatter/gather DMA is the most important feature of a NIC card to support the zero-copy feature in Linux kernel, which is essential for any high-performance networking interface. In our current prototype, we support only one hardware ring buffer for each of the receive (RX) and transmit (TX) queue. In every queue, there is a head and a tail pointer stored in the NIC control registers. The NIC hardware uses one-level of indirection to store packet data payload separately, while putting a 16-byte buffer descriptor in the ring. The

descriptor rings are pre-allocated by the NIC device driver during the initialization time. To send a packet, the device driver fills out the free TX descriptor using packet fragments from the Linux kernel socket buffers (sk_buff). The driver updates the ring buffer tail pointers before the hardware DMA engine can walk through the ring data structures in memory and send the packet to network. Similarly, when a packet is seen from the network, the NIC RX DMA engine fills out the pre-allocated RX ring buffers and notifies the OS using a hardware interrupt.

Our NIC device driver supports all features of what a generic Linux Ethernet device driver has, except that we encode the destination MAC address in the Ethernet header for our source-routing switches. Nevertheless, we can run unmodified TCP/IP user applications using the standard socket programming interface.

Besides the above architectural features, we make several simplifications and list the reasoning of the abstraction as the following. Many of the features we do not model are optional performance optimizations under our hypothetical usage model:

- *No hardware checksum*: Calculation of TCP/IP and Ethernet frame checksum is a computation intensive operation. Many modern NIC hardware can offload checksum calculations of various network layers from the CPU to the NIC hardware. Since DIABLO is a reliable simulator design, it is impossible to see a corrupted data packet that will results a checksum verification failure. Hence, we do not model the hardware checksum acceleration. Instead, our Linux device driver tells the kernel to completely turn off the checksum calculation in the networking stack. This emulates having a hardware checksum offloading engine in the NIC without taking extra time.

- *No on-chip hardware queue and descriptor prefetcher*: Since the packet descriptors are stored in the system DRAM that is shared by CPU and other I/O devices, the DRAM read/write operations are bursty. In addition, commodity NICs access the main DRAM through more complex I/O interfaces, such as PCI express. In order to operate at the link speed, there are usually small SRAM-based buffers on the Ethernet controller to cache the DRAM descriptors. To improve performance, some controllers can speculatively prefetch future descriptors from the DRAM ring buffer. Due to the data structure of a ring, the speculative prefetch is very effective. The overall performance is only limited by the I/O interface bandwidth, which appears to be less an issue given a link speed of 1 Gbps or even 10 Gbps. In addition, some newer server chips, such as Intel Xeon E5 processors [24], support directly forwarding data to the last-level processor cache thereby eliminating the need of DRAM prefetching. Thus, in our current abstracted model, we do not model this architecture detail for the sake of model simplicity.

- *No multiple receive hardware queues*: Modern 10 Gbps Ethernet NICs, such as Intel 82599, support multiple hardware queues for different TCP flows, virtualization, and multicore receive processing. Multiple receive queues are always used with multicore when processing multiple flows in parallel. The DIABLO server model only adopts a simple computation model, where the multicore effect is straightforwardly modeled

with having a single faster host processor. Although having a multiple hardware queues support is interesting, the basic ring buffer architecture remains the same for all queues. We could easily extend the current abstracted NIC model in the future.

- *No fancy protocol filters*: Newer 10 Gbps Ethernet NICs have CAM-based hardware flow filters to speed up flow classifications to work with multiple hardware queues. Powerful 10 Gbps NICs like Intel 82599 support as many as 128 5-tuple filters for TCP/IP flows. However, in real datacenter scenario, on average only around 10 flows are active per node [73]. Given the O(100) time simulation period DIABLO is targeting, lacking of a hardware flow filter is unlikely to have a significant performance impact. Hence, TCAM-based protocol filters can be a future work to expand the functionality of our current model.

- *No hardware send segmentation offloading*: Send segmentation offloading is an effective hardware approach to assist the TCP/IP segmentation feature in Kernel, thus reducing the CPU load when handling at a higher link speed above 10 Gbps. Hardware segmentation requires a more complex target design. In our initial abstract model, we compensate this target performance optimization by simple adjusting the server timing model to have a faster target processor. In a later chapter, we will describe a future RAMP Gold-based NIC model, which can easily model such complex performance features using a programmable micro-code engine.

Figure 4.5 shows the abstracted NIC model. To create a more balanced system, the NIC model has the same number of hardware threads as those for the server model. Each hardware thread simulates an independent NIC in target. The NIC processes incoming and outgoing packets in units of 64-byte in fixed amount of time in target decided by the line rate, which renders the NIC timing model trivial to design. The data path is 64-bit matching that of our switch model, which simulates sending and receiving data in eight host cycles before synchronizing with other models in the system.

Similar to our switch and server model, we design a simple host cache that optimizes for the descriptor-based DRAM references from the NIC DMA engine. Each entry in the host-cache caches a 64-byte data for either send (TX) or receive (RX) DMA engine. The host 64-byte data reference is always issued in burst to DRAM. Along with each entry, there is a base address register, which is automatically updated while prefetching a ring buffer descriptor. We use the cached base address to determine physical addresses of packet data.

We also build a small receive alignment buffer in the host cache that ensures the IP/TCP header of every packet received from the network is aligned to a 4-byte boundary. This buffer is a cache performance optimization in the Linux kernel. However, it is mandatory on SPARC machines that do not support misaligned load/store instructions. Some low-end NICs, such as Realtek 8169, lack of a byte-addressable a RX DMA engine. Instead, they use *memcpy* in the device driver to work around this issue at the cost of performance. In practice, having a fixed-size alignment circuit is sufficient to run the Linux networking stack, while still maintaining the overall design simplicity.

The majority of the design complexity comes from supporting a byte-addressable scatter-

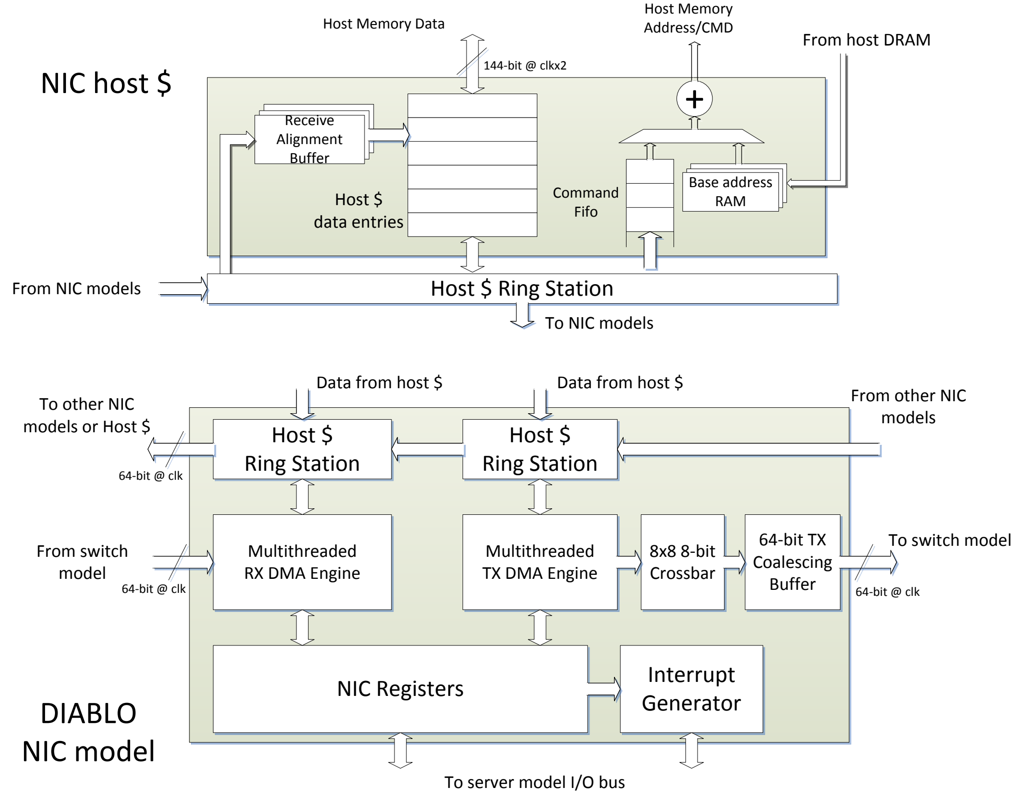


Figure 4.5. DIABLO NIC model architecture

gather DMA. Hence, we model a byte-alignment hardware unit using highly behavioral RTLs, which is essentially an 8x8 8-bit crossbar with complex controls. A byte-addressable DMA is required for the send data path, as application software could generate send data aligned at arbitrary byte boundaries. Given the interaction of host-cache, host-multithreading, and multi-cycle access, the state machine design of the NIC model is not trivial. In fact, it is more difficult to design a FAME-7 NIC model than to design the equivalent target.

The total resource consumption of the abstract NIC model is comparable to that of the RAMP Gold integer pipeline. The current NIC model is a straightforward FAME-7 implementation of the target hardware. The basic functionality of a NIC is to manipulate packet data stored in the DRAM, while DRAM performance is less an issue for FAME models. The NIC model is not on the performance critical path of DIABLO. Table 4.2 shows the FPGA resource consumption of the NIC model. The TX engine with the byte-alignment hardware accounts for the majority of the logic resource consumption. This design gives us insights on future NIC model designs described later.

Component Name	*LUT*	*Register*	*BRAM*	*LUTRAM*
RX engine	278	221	0	92
TX engine	1255	344	0	132
Host cache	745	560	5	72
NIC registers	288	30	0	406
Miscellaneous	11	275	0	4
Total	2577	1549	5	1040

Table 4.2. The FPGA resource consumption breakdown of the abstracted NIC model on Xilinx Virtex 5 LX110T

4.5 Modeling a Datacenter with a Packet-Switched Interconnect

In this section, we describe how to use aforementioned individual DIABLO models to build a large-scale testbed to simulate thousands of servers with switches using 11 BEE3 boards.

4.5.1 Simulated Datacenter Target

Datacenters use a hierarchy of local-area networks (LAN) and off-the-shelf switches. Figure 4.6 shows a typical datacenter network arranged in a Clos topology with three networking layers. At the bottom layer, each rack typically holds ∼20–40 servers, each singly connected to a commodity *Top-of-Rack (ToR)* switch with a 1 Gbps link. These ToR switches usually offer two to eight uplinks, which leave the rack to connect up to several *array switches* to provide redundancy and bandwidth. At the top of the hierarchy, *datacenter switches* carry traffic between array switches usually using 10 Gbps links. All links use Ethernet as the physical-layer protocol, with either copper or fiber cabling depending on the connection distance.

As we move up the hierarchy, one of the most challenging problems is that the bandwidth "over-subscription" ratio (that is, the bandwidth entering from below versus bandwidth to the level above) gets worse rapidly. This imbalance is due to the cost of switch bandwidth, which grows quadratically in the number of switch ports. The resulting limited datacenter bisection bandwidth significantly affects the design of software and the placement of services and data, hence the current active interest in improving network switch designs.

We pick the standard Clos topology and an interconnect with source-routed all 10 Gbps Ethernet interconnect as our simulated target. As a proof of concept, each rack contains 31 servers and one top-of-the-rack switch (ToR) without redundancy, which can be easily modeled at the cost of more FPGA resources.

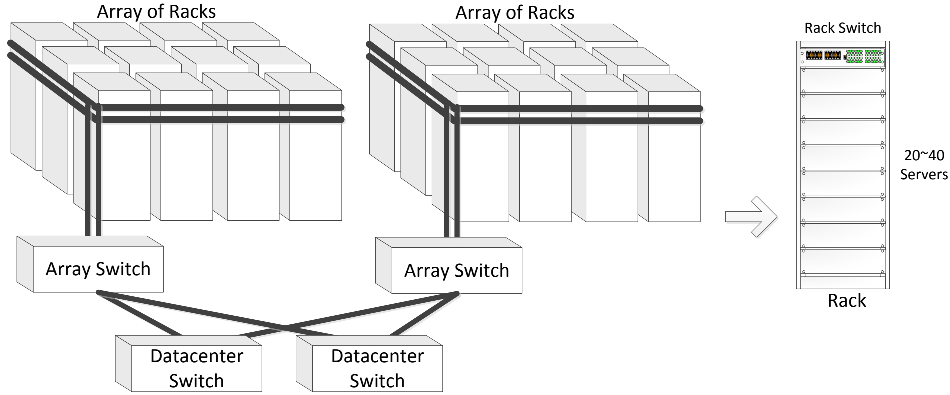


Figure 4.6. A typical datacenter network architecture.

4.5.2 Modularized DIABLO system design: Array FPGAs and Rack FPGAs

In order to easily scale up the emulated datacenter, DIABLO employs a modularized design that includes only two distinct types of FPGA system designs. Figure 4.7 shows the high-level simulator architecture for the typical target datacenter configuration presented in Figure 4.6. We map all server models along with the ToR switch models into *Rack FPGAs*, and array and datacenter switch models to separate *Switch FPGAs*. Besides scaling up the number of simulated servers, this partitioning enables a more modularized model design that eases experimentation with new array and datacenter switch designs. To further simplify switch model design, we keep any switch model within a single FPGA. Following the physical topology of the target system, we connect Rack FPGAs to Switch FPGAs through several time-shared multi-gigabit serial transceivers using low-cost copper cables, such as standard SATA cables. Each FPGA has its own simulation scheduler that synchronizes with adjacent FPGAs over the serial links at a fine granularity to satisfy simulation accuracy requirements. For example, a 10 Gbps switch with a minimum flit size of 64 bytes requires a maximum synchronization interval of 51.2ns.

We reduce host communication latency by using our own protocol over the serial links, achieving FPGA-FPGA communication latencies of around 20 FPGA logic cycles, which is roughly the latency for a host DRAM access on the FPGA. Including all overhead, the overall round-trip latencies between FAME models on different FPGAs is only around 1.6 microsecond. In addition, the host-multithreaded design further helps to hide the simulator communication latency, removing model synchronization latency as a simulator performance bottleneck. Since we partition the model on boundaries of physical switch, we also take advantage of physical link latencies in target between switches to relax the simulation synchronization need.

To make the design simpler and more modular, we select multi-gigabit serial transceivers

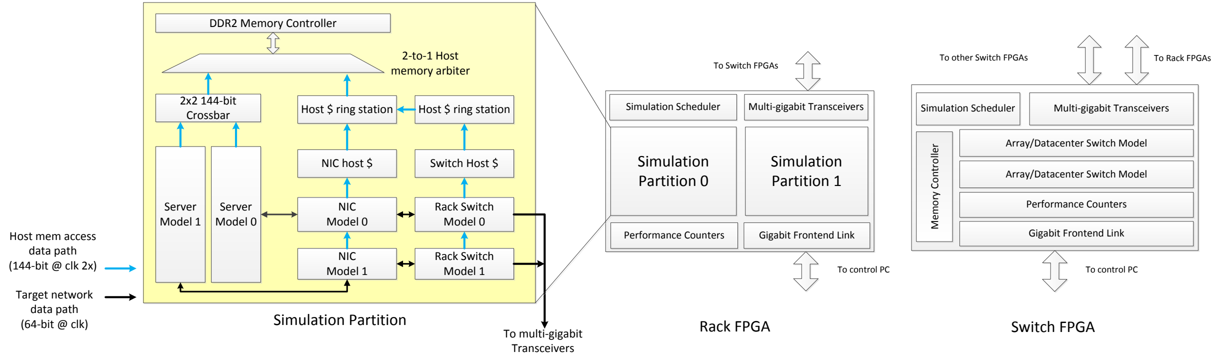


Figure 4.7. DIABLO FPGA simulator architecture

as the only inter-FPGA connection instead of the high-speed parallel LVDS links often seen on multi-FPGA boards. Specifically, parallel LVDS links increase design complexity. To ensure reliable transmission, designs require complicated dynamic calibration and special eye-opening monitoring circuit on groups of I/O signals. In addition, designs with LVDS links are less portable because of varying I/O layouts on different boards, making connections between Rack FPGAs and Switch FPGAs less flexible. Moreover, LVDS links increase both PCB board and FPGA cost because they require more FPGA I/O pins and link wires for a given link bandwidth. Finally, we found that the multi-gigabit serial transceivers provide enough bandwidth between FPGAs considering our overall simulation slowdown between 250× and 1000× of real time. For example, 2.5 Gbps transceivers are common on three-year-old Xilinx Virtex 5 FPGAs. The bandwidth of a single transceiver translates to 500 Gbps to 2500 Gbps in the target, which far exceeds the bandwidth between a few racks and several array switches. Moreover, recent FPGAs have significantly enhanced serial transceiver performance, supporting up to 28 Gbps bandwidth [20] in the 28nm generation.

Each Rack FPGA contains multiple sever model pipelines in a 32-thread configuration, simulating a rack of servers. We evenly distribute the sever models to multiple host DRAM channels and partition the host DRAM for server computations. On the Xilinx Virtex LX155T FPGA of BEE3 board, each rack FPGA uses up both memory controllers with fully populated DIMM modules to support more DRAM capacity. We manage to put down four server model pipelines per FPGA, two for each DRAM channel, simulating four racks of 124 servers. Each server model has one ToR switch model attached to the same DRAM controller. Therefore, we can perfectly divide the rack FPGA into two identical physical simulation partitions, each consisting of one DRAM channel with 8 GB storage.

Figure 4.8 illustrates the floor plan of the rack FPGA, showing the two host DRAM partitions occupying half of the chip. One is on the top, while the other is at the bottom. This partition also allows us to use simple physical placement constraints to speed up the time-consuming place and route process and improve the quality of result from the CAD tool. We employ one 3 Gbps transceivers on each rack FPGA as the external connection to a switch FPGA, connecting the uplink on all rack switch models.

We equally divide the 16 GB physical DRAM resources into 128 128 MB-partitions.

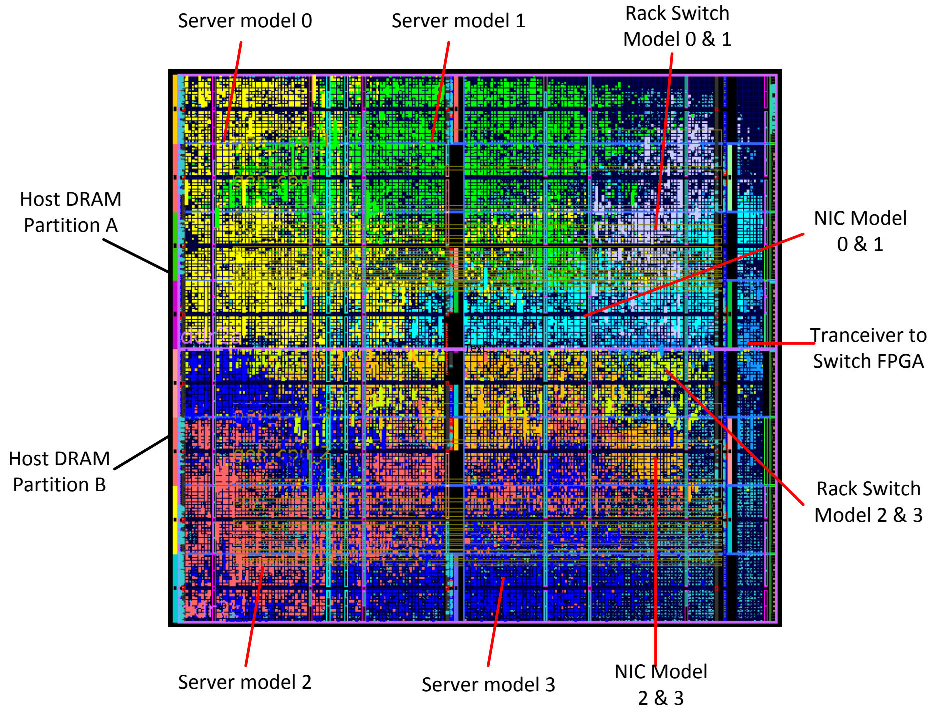


Figure 4.8. DIABLO FPGA simulator architecture

Each simulated server takes one partition as the target memory storage. For each server pipeline, we use 31 threads out of the 32 available threads and save the DRAM storage of the remaining thread for simulating packet buffers on the ToR switch. We discuss how to increase effective DRAM partition size in Chapter 7.

As discussed in the previous chapter, given a 1000x simulation slowdown, the DRAM bandwidth is not a performance bottleneck. On the other hand, the DRAM capacity is a critical shared resource among all DIABLO models. To connect DIABLO models, we built a distinct host memory interconnect. Server models account for the majority of the host DRAM traffic, but there are relative small number of server models pipelines per host memory controller. Therefore, we design a pipelined 144-bit crossbar interconnect to connect all host caches on the server models, optimizing for low-latency accesses with single burst size. The number of ports of this crossbar is parameterized, and it can be decided during the synthesis time.

In contrast, we use a 144-bit ring interconnect for NIC and switch models. The ring interconnect optimizes for simple controls to support non-uniform burst sizes from the NIC scatter-gather DMA engines and switch packet buffers. A ring interconnect is also routing-friendly on FPGAs to improve the actual circuit performance.

Both the ring and the crossbar uses credit-based flow controls to simplify pipelining on FPGA, which helps to meet timing closures and reduce the CAD place and route time. The host memory interfaces of all DIABLO models employ a non-blocking design that allows all

hardware threads issue memory requests without being blocked by pending requests from other hardware threads.

We put an arbiter in front of the memory controller to select requests from the ring and the crossbar. The arbitration is done at the burst boundary to take advantage of the DRAM row hit and maximize the DRAM command bus utilization through the DDR2 additive latency. Although the arbiter could provide hints to the DRAM controller based on the request addresses to speculatively issue row PRECHARGE commands for better performance, we do not do so currently for the reason of design simplicity. For the same reason, the whole host memory interconnect only supports in-order accesses.

In terms of the physical design, we double clock the data path to keep up with the data rate of the memory controller. We also use ECC to protect all memory data paths that interact with the ECC memory DIMMs on the BEE3 board.

All DIABLO models on the same FPGA share the single Gigabit Ethernet connection as the frontend connection for bootstrapping, console output and functional I/Os. We design a simple ring interconnect for the frontend connection to ease the routing pressure. Every model in the design has numerous hardware performance counters that periodically send performance statistics through the frontend link.

To further simplify the switch model design, we keep any switch model within a single FPGA. Specifically, we put several array/datacenter switch models in a *switch FPGA*. The switch FPGA is actually a shrunken down version of the rack FPGA with only switch models and less server model pipelines. We use only one host DRAM channel without the need of fully populated DIMMs, which is sufficient to simulate all switch port buffers. We do put a single server model functional pipeline without a timing model, which we only use to run functional configuration for our switch models. The server functional pipeline can also be doubled as a control-plane processor for the simulated switch. Other than these differences, the switch FPGA uses the same host memory network and frontend interconnect found in the rack FPGA. Each switch FPGA supports up to eight 3 Gbps transceivers that connect to either rack FPGAs or other switch FPGAs.

Table 4.3 shows the overall rack FPGA resource utilization on Xilinx Virtex 5 LX155T-2 FPGA on the BEE3 board after place and route with Xilinx ISE 14.3. Including the resources dedicated for FPGA routing, the device is pretty close to fully utilized with 95% of logic slices occupied at 90 MHz.

We also spent a significant amount of register resources, shown in the miscellaneous row in Table 4.3, to build a pipelined host interconnect to glue all DIALBO models with host DRAM resources. On overall, the rack FPGA is a BRAM bounded design. This utilization implies that we could support more server pipelines and simulate more target nodes by using a larger FPGA with more memory DIMMs.

On the other hand, our abstracted switch models consume a very small amount of FPGA resources. Hence, the switch FPGA simulating one array or datacenter switch consumes less than a quarter of the overall FPGA resources. The low resource utilization suggests that there is a great potential to support a more detailed switch model in the future. In addition, the physical connectivity of the switch FPGA is high-speed transceiver bounded. This

physical connection limitation is constrained by the choice of FPGA and the board design rather than by the DIABLO system architecture.

Component Name	*LUT*	*Register*	*BRAM*	*LUTRAM*
Server Models	28,445	37,463	96	6,584
NIC Models	9,467	4,785	10	752
Rack Switch Models	4,511	3,482	52	345
Miscellaneous	3,395	16,052	31	5,058
Total	45,818	62,811	189	12,739

Table 4.3. The FPGA resource consumption breakdown of the RACK FPGA on Virtex 5 LX155T. The overall occupied slice utilization is 95% and BRAM utilization is 89% after place and route at 90 MHz

4.5.3 Building a 4,000-node DIABLO System with 11 BEE3 Boards

We need several multi-FPGA BEE3 boards to build a sizable DIABLO system. Although the BEE3 board uses a 2007 FPGA and older DRAM DIMMs, it is optimized for computer system emulations offering a large DRAM capacity. The board also supports eight 10 Gbps CX4 connectors, two per FPGA, on the front panel for external connections. However, BEE3 is not designed specifically for DIABLO.

For instance,in practice not all high-speed transceivers are easily accessible on the board. Each CX4 connector bundles four independent transceivers and we have to use a CX4-SATA breakout cable to access the single transceiver. In addition, BEE3 uses half of the total 16 transceivers for a PCIe x8 connector, which requires a custom break-out daughter card for the DIABLO usage scenario. All these BEE3 hardware features increase the cable cost of prototyping DIABLO. To keep the wiring flexible, we choose to use the eight CX4 connectors exclusively with breakout cables for simple inter-board connections.

Figure 4.9 presents the block diagram of a DIABLO system with 3,968 computing nodes, 128 rack switches, 8 array switches, and one datacenter switch using 11 BEE3 boards. We program all four FPGAs on eight of the 11 boards as rack FPGAs to model all computing servers and rack switches. Hence, we are getting a near 100% FPGA and DRAM capacity utilization on these boards. We use two BEE3 boards for simulating the eight array switches, having each FPGA simulating one switch. One FPGA on the remaining board is dedicated to the single datacenter switch model. Each rack FPGA has one 3 Gbps link connecting to a switch FPGA on another board. We connect all four rack FPGAs on one board to the same switch FPGA on another board through one 4-lane CX4 connector. We use one transceiver in the other CX4 port of the switch FPGA to connect to the datacenter switch.

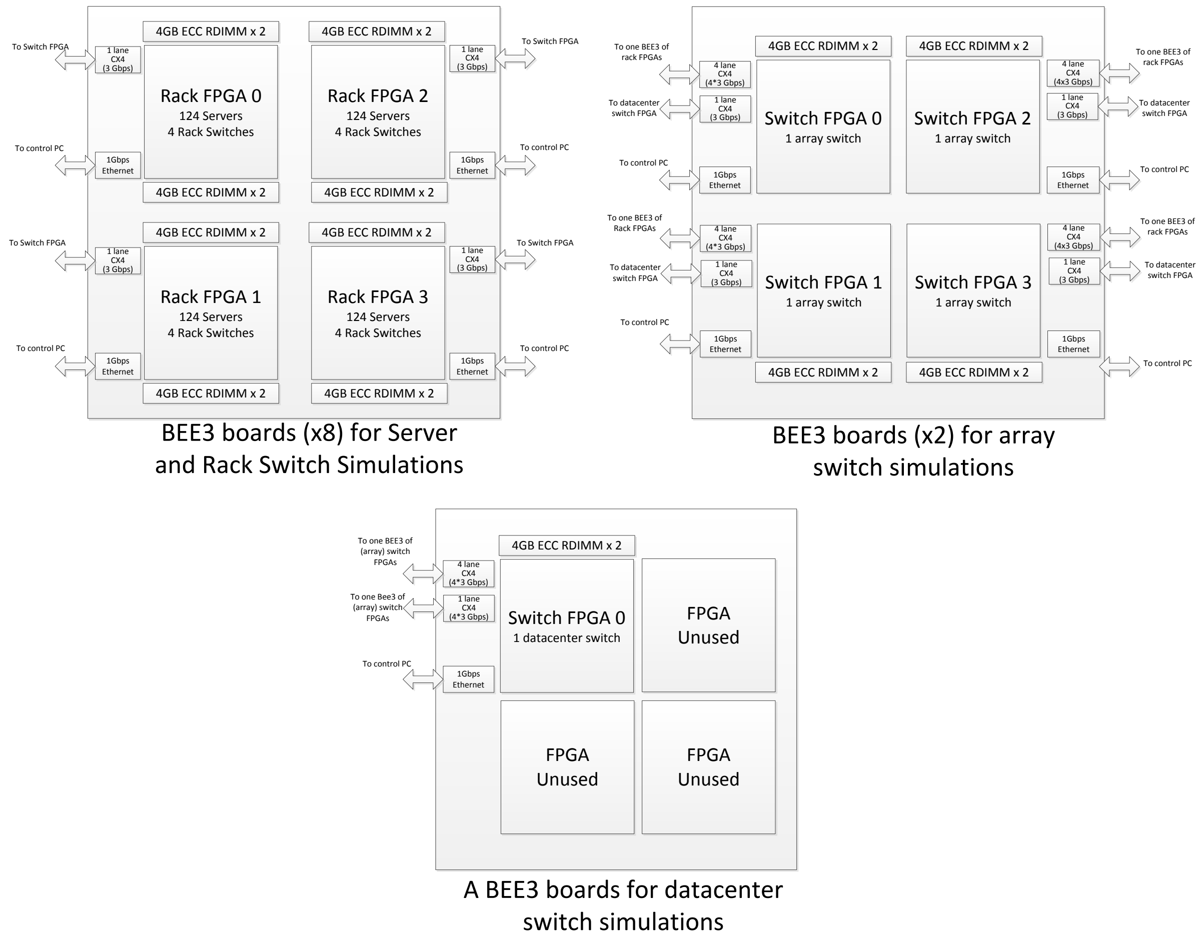


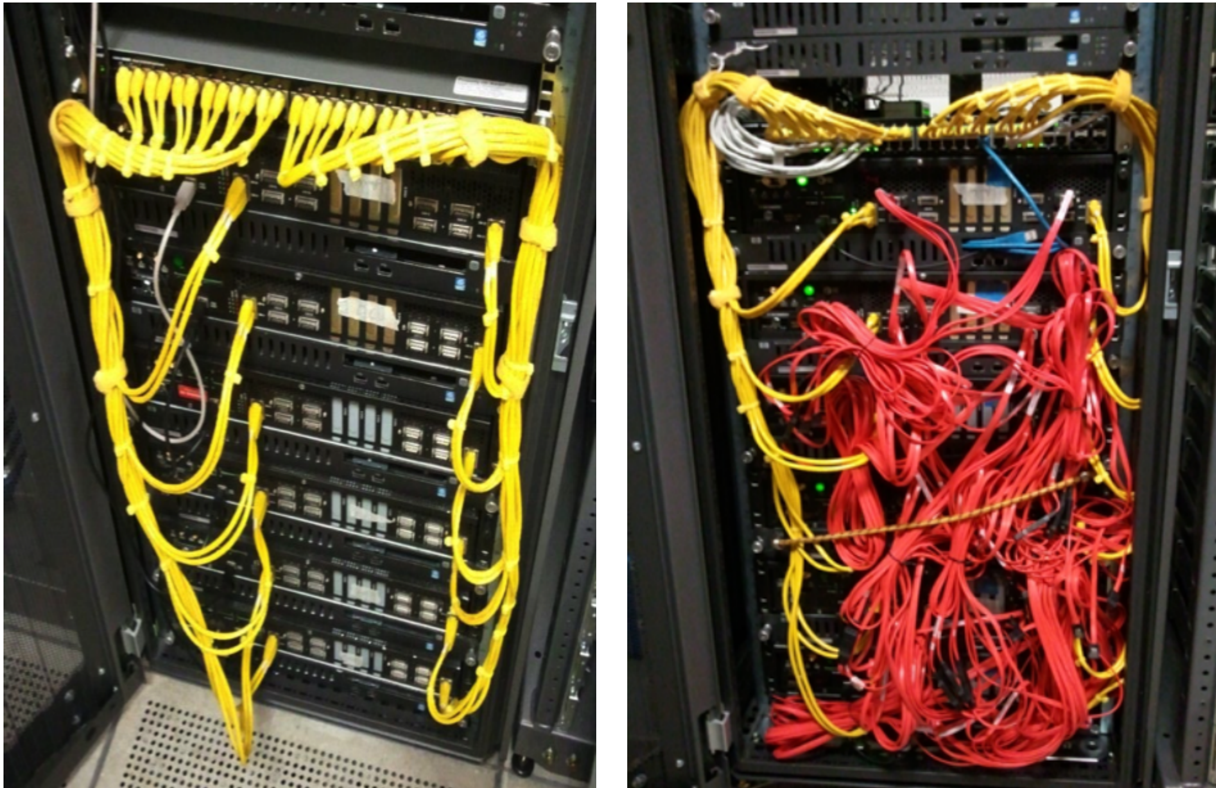
Figure 4.9. The block diagram for a DIABLO system with 11 BEE3 boards

This topology matches the simulated target, where 496 servers in 16 racks connect to a single array switch.

The scalability of switch FPGAs is limited by the number of available transceivers on BEE3. If we built PCIe x8 daughter cards to break out transceivers occupied by the PCIe connector, each switch FPGA can connect to two more boards simulating more servers. We could therefore add additional 16 boards to the existing system to build an emulated datacenter with 11,904 servers with 384 rack witches. As a demonstration, we designed a six-board DIABLO system (constrained by the BEE3 board availability and DRAM cost) with 1,984 nodes, illustrated in Figure 4.10. The system contains 384 GB DRAM in 48 independent DRAM channels, with a peak bandwidth of 179 GB/s. It occupies half of a standard server rack space, and consumes about 1.2 kwatt when active. Besides the CX4-SATA cable connections, every FPGA connects to frontend control servers through a 48-port Gigabit Ethernet switch.

We store boot disk images and console I/O logs of all 1,984 simulated servers in three multicore x86 servers. Each server has three 1 Gbps Ethernet links to the control switch.

There are 21 independent frontend server processes each controlling one active FPGA in the system. In order to isolate these processes from interfering with each other while still maintaining a modular design, we build a software virtual switch bridging all processes with three shared physical Ethernet ports, which is similar to the VDE switch [31] used by popular virtual machines like QEMU and VirtualBox.



(a) Without inter-board connections (b) Fully-connected with high-speed cables

Figure 4.10. DIABLO cluster prototype with 6 BEE3 boards

4.5.4 Summary

In this chapter, we described DIABLO hardware architecture and design strategies. We presented many host techniques we used on FPGA to build a high-density FAME simulator. We show that DIABLO by itself is a big machine built on FPGAs with a different design space. Some of the design choices we made are not intuitive, if building a real target implementation instead. Finally, we discussed the implementation trade-offs of DIABLO prototype in real-world.

Chapter 5

Using DIABLO to Conduct Datacenter System Research at Scale

We present three examples of using DIABLO to conduct datacenter network architecture research at all levels, including hardware, transport protocols, and applications. In the first case, we use DIABLO to model a novel circuit-switching research datacenter interconnect hardware from Microsoft Research. The second case reproduces the famous datacenter TCP Incast [126] throughput collapse effect with DIABLO packet switching models. The last case shows a 1,984-node DIABLO system running an unmodified *memcached* [25] service, which is an in-memory key-value store caching system currently used by many major websites, such as Facebook and Twitter.

5.1 Case One: Modeling a Circuit-Switching Datacenter Network

Although packet-switching network dominates today's datacenter network, circuit-switching has become an attractive alternative approach in recent years to many datacenter researchers for providing a simple and easily manageable network with more quality-of-service guarantees. To demonstrate the flexibility of our FPGA-based modeling approach, we employ DIABLO to study an early version of a novel circuit-switching datacenter network using FPGAs designed by Chuck Thacker at Microsoft Research [122].

As part of the design team for three months, I also participated in the initial switch architecture design with early feedback from our DIABLO models. During that time, we

were able to finish a fully-working FAME-7 system, modeling all levels of switches in the target network architecture. We also ran some traffic patterns sampled from Microsoft Dryad Terasort application. The experience we gained through software and hardware co-design using DIABLO affected several design decisions in later versions of the architecture.

5.1.1 Target network architecture

The context of the circuit-switching network we model is a datacenter built in shipping containers. In this architecture, there are two rows of server racks with 16 44U racks per row in every container. Figure 5.1 shows the circuit-switching switches in a typical container. Fifteen server racks contain two Level-0 (L0) switches that aggregate traffic from 20 2.5 Gbps end-nodes linking onto two 10 Gbps uplinks. In one rack of each row, the L0 switch and 20U of server is replaced by the L1 switch for the row and its *route controller (RC)*. There are two 128-port L1 switches, each having 64 inward-facing and 64 outward-facing 10 Gbps ports. The two L1 switches are used to provide failure-tolerance. Up to 64 containers are connected using an all-to-all topology without further central switching needed. The bisection bandwidth per container is 640 Gbps, while the overall bisection bandwidth for the whole center is 20.5 Tbps.

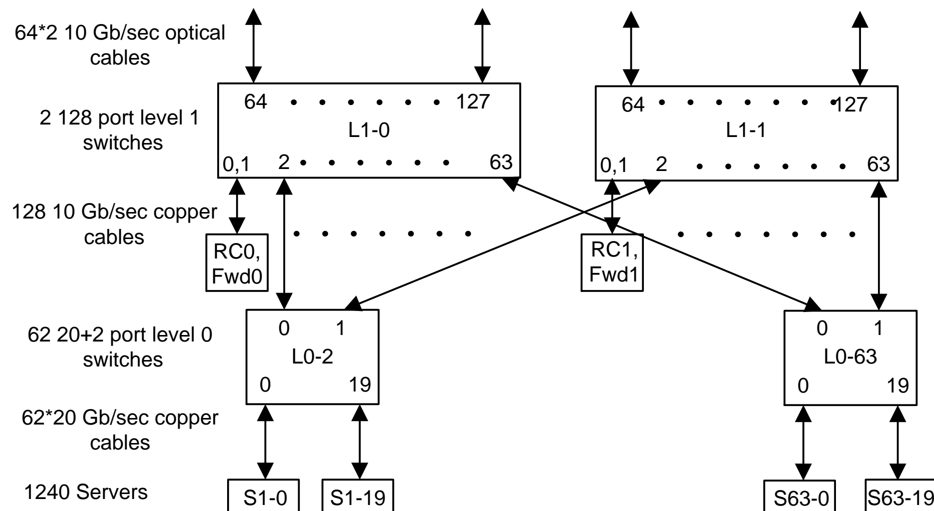


Figure 5.1. Target architecture of the novel circuit-switching network

Because the datacenter network is a well-controlled environment with a known topology, this network employs a much simpler arrangement compared to IP switching, similar to ATM but with smaller cells and different routing algorithms. There are two types of traffic in this network: *Scheduled Traffic* and *Datagram traffic*. The former accounts for long data flows, which require circuit path setup and teardowns. The latter is for short flows that do not require path setup. Figure 5.2 illustrates the frame format and typical data paths involved for both types of traffic. The L1 switch is an input-buffered 128x128 crossbar. The buffers

are managed by the *route controller* attached to port 0 of each L1 switch. An end node needs to send route setup request to the RC before sending any scheduled traffic and path teardown request to release L1 buffer resources. All the datagram traffic is handled by the *datagram forwarder* attached to port 1 of the L1 switch. The L1 switches route both type of traffic in fixed length repeating frames of 3.68 microsecond at 10 Gbps link rate. Each frame consists of 128 slots, and a slot carries 32 bytes of payload and a four-byte cell header. A single slot represents a unit of bandwidth of $10,000/128Mb/sec$, or $78.125Mb/sec$, which is the minimum bandwidth allocation unit in this network. Slot 0 is dedicated for all route setup and teardown traffic, while slot 1 is for the datagram forwarder. The L1 switches have small buffers and are very simple to implement with several low-cost FPGAs through bit slicing. The network employs simple source-routing. The round trip time for setup/teardown messages is less than 15 microsecond.

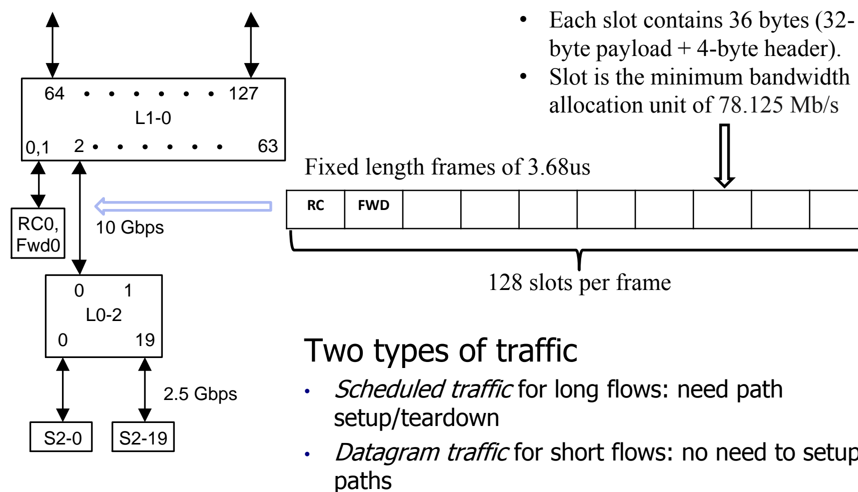


Figure 5.2. Frame formats and data paths of the target circuit-switching network

The L0 switches are also simple, as Figure 5.3 shows. There are five line units that each serves four low-speed lines at 2.5 Gbps. Data arriving on the uplinks are deserialized and transmitted to the line units on two pipelined 16-bit time-slotted ring. The ring also serves local traffic within the L0 switch. During each frame, the NIC sends a cell for each of the active connections (slots) it has, providing it has traffic queued for the slot. Data from NIC is unscheduled, buffered in the line units and sent during the next frame. The NIC also utilizes a non-standard signaling that requires a custom-designed software interface as well.

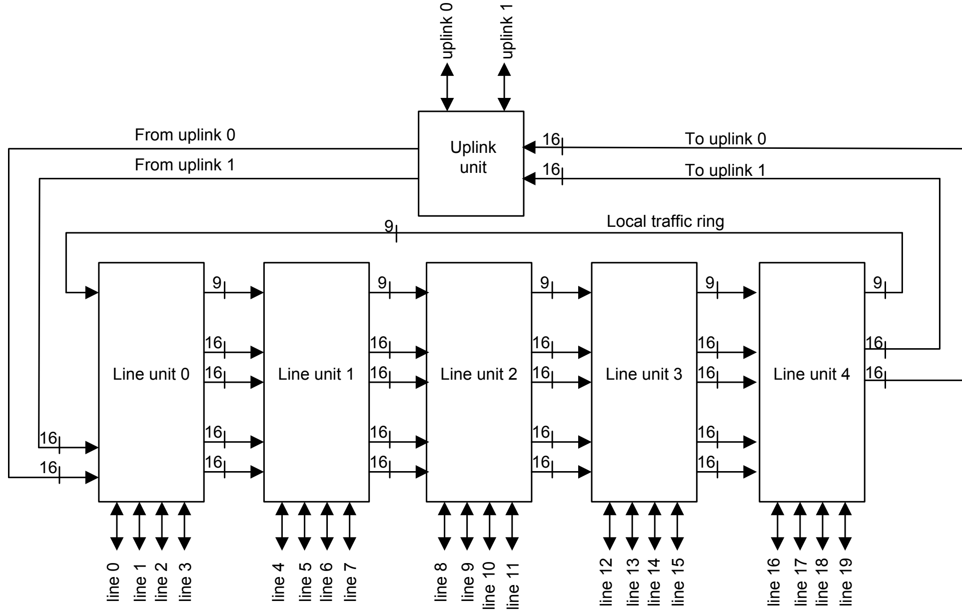


Figure 5.3. The L0 switch architecture

5.1.2 Building DIABLO models on FPGA

Our DIABLO models focus on scheduled traffic within a container, because the network architecture design between containers and datagram forwarding was still under development when this work had started. We modeled a network of 64 servers, one L1 switch with RC and four 16-port L0 switches.

The 64 servers are evenly distributed across four L0 switches of each connecting 16 servers each. On each simulated server, we build a simple NIC model working in a polling mode. To model these servers, we use a single DIABLO server pipeline with a 64-thread configuration. The simulated server has a simple timing model with perfect memory hierarchies (CPI=1) running at 2.2 GHz. The target network has a simple architecture with tiny buffer size that is designed to implement with multiple entry-level FPGAs. Due to the simple target design, both L0 switches and NICs can be easily modeled in the FAME-0 style with minimum abstractions. Furthermore, the source-routing target design eliminates the need of modeling expensive TCAM structures. Consequently, the DIABLO models of the L0 switch and NIC models look exactly like the target implementation except that our models are host-multithreaded in order to work with other DIABLO models.

On the other hand, most of the complexities of the network come from the 128-port crossbar in the L1 switch and the real-time route allocation circuit in the RC. Therefore, we use the FAME-7 approach to build much simpler hardware in our FPGA host to replace these complex multiport crossbar structure in target.

For instance, we built a time-multiplexed 64-bit ring network to connect all 128 ports on the L0 switch. Instead of processing crossbar input-buffer allocation in one clock cycle,

we built a decoupled FAME model on FPGA, which handles the crossbar scheduling in four cycles. Since the whole network is synchronous and runs in frames of fixed length, the corresponding timing model is trivial to design with a few counters and comparators.

Table 5.1 shows the FPGA resource utilization of the L1 switch model with full architecture details in comparison to the server models. We measure the design efforts in the lines of code. The implementation of a full-fledged FAME-7 circuit switch is straightforward. The resource consumption is moderate compared to simulating computations. However, the FPGA BRAM again becomes a critical resource that limits our modeling density. The overall DIABLO system runs at 90 MHz on a single Xilinx Virtex 5 LX110T FPGA.

FAME Model	Registers		LUTs		BRAMs		Lines of Systemverilog
64-server model	9,981	(14%)	6,928	(10%)	54	(18%)	35,000
L1 switch model	859	(1%)	1,498	(2%)	28	(9%)	2,550

Table 5.1. FPGA resource usage and lines of Systemverilog for different FAME models.

5.1.3 Dryad Terasort Kernel Data-Plane Traffic Studies

One of the key applications for this network is to support Map-Reduce style jobs. To test the effectiveness of the target design, we use the Dryad Terasort kernel to generate data-plane scheduled traffic. The original Dryad Terasort is written in C# and runs on Windows, so we have to hand code the kernel in C and run it on the bare-metal RAMP Gold proxykernel. We also wrote a NIC device driver with software managed send/receive queues running along with the Terasort kernel. Unlike traditional circuit-switching network, applications have better knowledge on path usage in the datacenter, so we add circuit path setup and teardown control to the Terasort application logic. There are three traffic patterns in the Terasort data plane, illustrated in Figure 5.4.

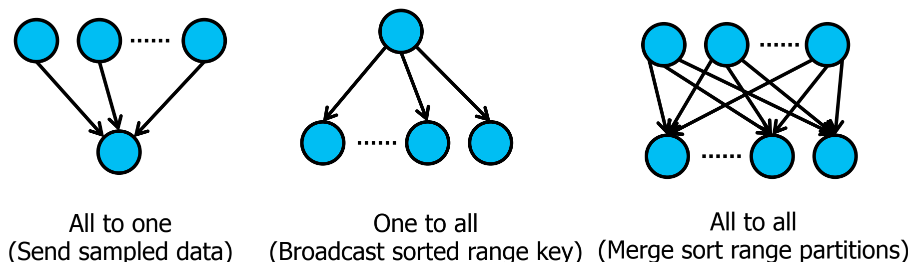


Figure 5.4. Three typical traffic patterns in the Dryad terasort

1. *All to one*: All nodes randomly sample ranged keys from the input data stored in local disks to a central node.

2. *One to all*: The center node sorts the received keys from phase one and broadcasts the sorted key back to all nodes.

3. *all to all*: All nodes use the sorted keys to perform merge sort and send bulk data to all other nodes.

Phase I and II use small sampled data instead of the full dataset, so they account only for a small portion of time of the total runtime. The performance bottleneck of the first two phases lies in the link of the aggregate node. However, the traditional packet switching network is susceptible to the subtle interactions between the store-forward packet buffer architecture and congestion control network protocols, so it could not efficiently handle the phase-I type traffic, for example the famous TCP incast throughput collapse issue.

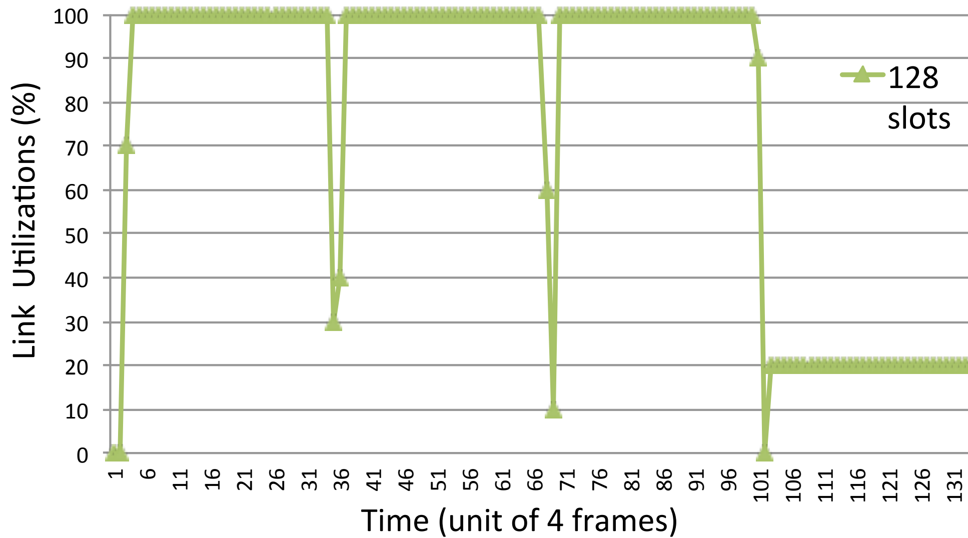


Figure 5.5. Link throughput on L0 switch for the Terasort phase one traffic

Figure 5.5 shows the aggregate node network link utilization running the Phase-I traffic. In our experiment setting, 63 nodes send 128-slot scheduled traffic of 4096 bytes to the central aggregate node. Each client requests one circuit path with capacity of one slot to the RC. We measured the link utilization through hardware performance counters on the NIC in time unit of four frames, i.e. $14.72\mu s$.

The circuit setups and teardowns are extremely fast for the all-to-one traffic. The performance impact to the link utilization is minimal. The duration of performance dips are less than four frames, shown as utilization dips at time 35 and 69 on the graph. The target circuit-switching network does not have any application throughput collapse, unlike those found in store-forward packet switching switches. The link kept saturated majority of time before 1.5ms, when 80 percent nodes finished Phase I.

After 1.5ms, the link became underutilized because there are fewer active circuit paths. This suggests that in the future better frame slot allocation schemes can improve the overall performance while there are fewer active links to keep the pipe saturated.

The link utilization of Phase II traffic is similar to the one in phase one. However, the performance of Phase II is very intuitive and less interesting in the context of switch designs, because the bottleneck is at the sender node instead of the network.

The all-to-all traffic in Phase III is the most important one for the Map-Reduce style batch processing jobs. We tried two types of traffic. One is fixed payload in 4KB units (virtual memory page/disk block size), while the other is a variable-size workload with payload size randomly selected from 3KB to 5KB. Figure 5.6 presents the utilization of link 0 on the L1 switch for the fixed workload over time.

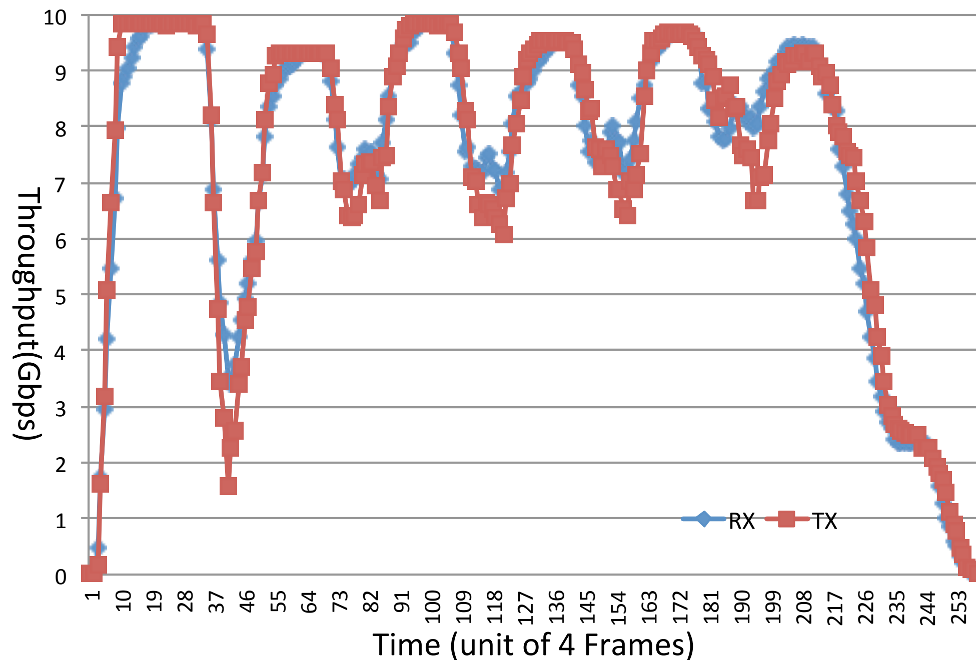


Figure 5.6. Link throughput on the L0 switch for the fixed length traffic in phase III

Figure 5.7 illustrates the link utilization with the random workload. Link utilizations of the rest three links show similar behaviors. Since the target network is synchronized on frames, the circuit setup on route controller has become a bottleneck, when all NICs try to send circuit setup and teardown requests simultaneously. The prototype only dedicates one slot of 78.125 Mbps for the control traffic. On the other hand, the control traffic is very bursty leaving the data frames under-utilized before a new path has been set up by the route controller.

Moreover, even taking into account the effect of full device driver, the randomness introduced by software could not offset these throughput dips. This performance issue is because the route control requests are queued locally inside the L0 switch and will be sent at specific time in a frame defined by the target protocol. On the contrary, random traffic works around this problem by amortizing circuit path setup and teardowns over time. This performance behavior we observed from DIABLO led to a redesign of the circuit setup/teardown architecture.

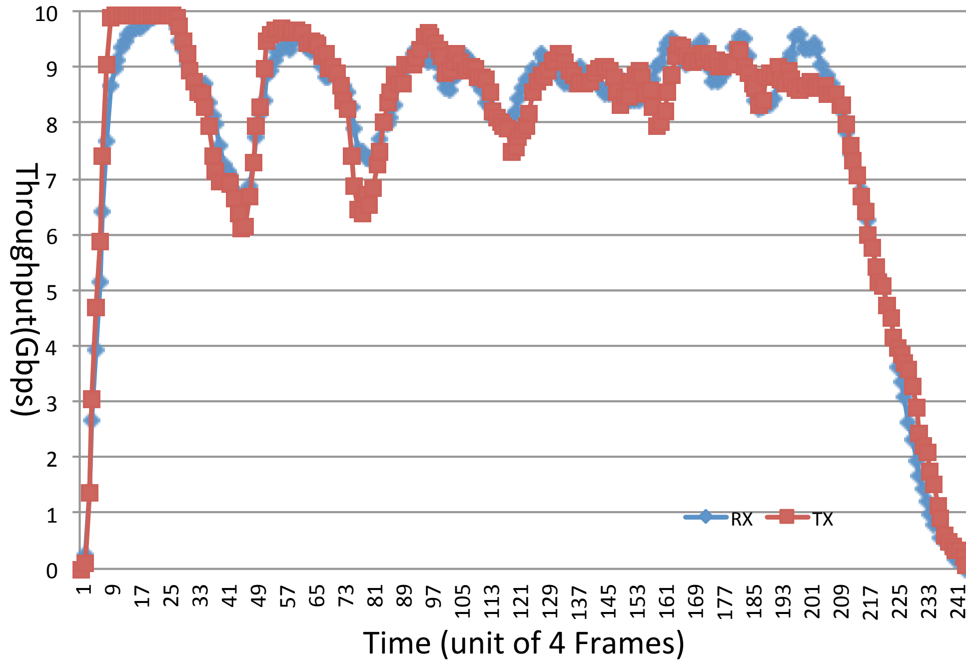


Figure 5.7. Link throughput on the L0 switch for the vairable length traffic in phase III

5.1.4 DIABLO for Software Development

Having an accurate DIABLO performance model enable studying the target architecture in great detail along with software applications, which provides early feedback for designers and helps to discover unexpected real implementation issues. The accurate performance models not only provide a platform to evaluate architectural design ideas, but also serve as an excellent platform for faster driver development.

In addition, DIABLO is capable of simulating various host processor speeds, which allows us to study the impact of NIC software. We found even at a link speed of 2.5 Gbps, the software processing is still every challenging. Although the circuit-switching network is designed to be lossless, the software can actually drop frames in practice. This observation also led to a better design of link-level retransmission and fault-handling in a future revision of this design.

5.2 Case Two: Reproducing the TCP Incast Throughput Collapse

Incast is a many-to-one communication pattern commonly found in many datacenters implementing scale out distributed storage and computing frameworks, such as Hadoop and

Map-Reduce. In particular, the TCP Incast problem [126, 114] refers to the application-level throughput collapse that occurs as the number of servers sending data to a client increases past the ability of an Ethernet switch to buffer packets. Figure 5.8 shows the simple network topology setup of the TCP Incast problem.

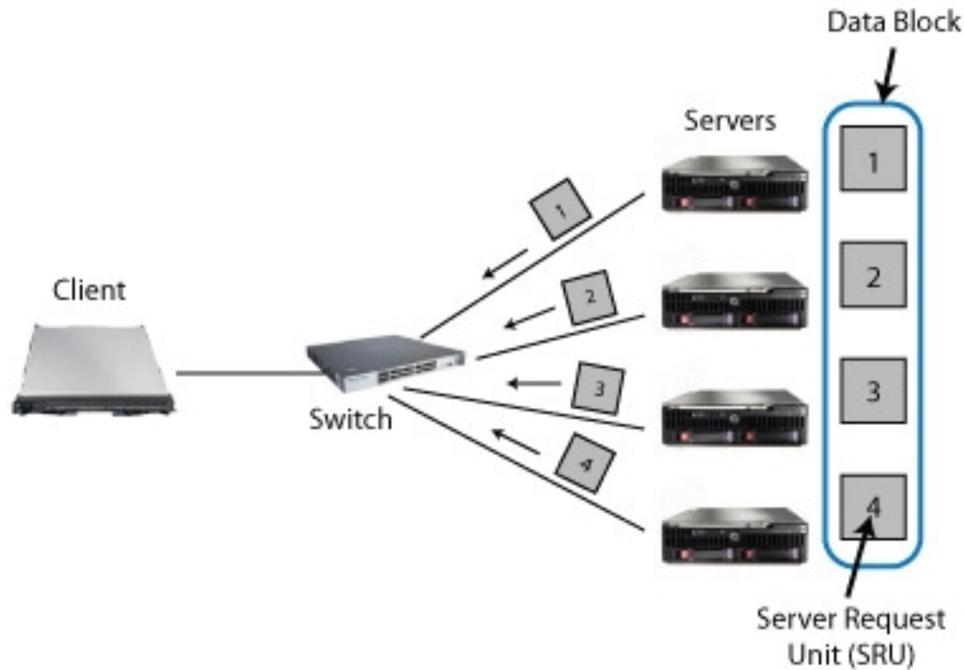


Figure 5.8. A simple cluster-based storage environment for the TCP incast problem

Previous work [126, 50] has focused on studying the interplay between the TCP transport protocol retransmission time out (RTO) value and small switch buffers on low-cost top-of-the-rack switches. Unfortunately, many switch vendors do not publish details of their packet buffer architecture designs and provide near zero visibility into their product. Hence, researchers were forced to use poplar event-driven simulation [126] and analytical models [50, 140] to model an abstract switch in order to reverse engineer the real cause of the problem, focusing only on the switch buffer and TCP protocol. People have also validated their models against measurements from real machines, showing their models could successfully reproduce the application throughput collapse. As a consequence of these studies, there have been many proposed solutions on the network transport protocol and the switch architecture for Incast. Representative approaches include modifying TCP parameters [126, 50], congestion control algorithm [135], adding large switch buffers [114], and supporting the explicit congestion bit (ECN) on each switch.

However, as pointed out in [114], TCP incast is only obvious under specific hardware and software setups, for instance with low-cost switches that has very small shared packet buffers. Some simple switch configurations such as turning on vendor specific QoS features and use larger request block size can significantly delay the onset of the throughput collapse. During our conversations with industry researchers, quite a few people call TCP Incast "artificial".

In addition, almost all performance models people developed are based on observations from low-cost Gigabit Ethernet switches. It is apparently the cause that triggers the application throughput collapse, giving that modern servers are fast enough to saturate a gigabit link.

Inspired by the end-to-end system design argument, the solution of the TCP Incast problem is to improve application throughput. Besides, there are still so many layers below the simple application in addition to the network transport protocol and switch, such as host processing performance, choice of OS syscalls, kernel scheduler, NIC hardware and drivers. Considering the fact that TCP incast is observable only under specific settings, could we really conclude the tiny switch buffer is the solo direct cause? If we are not constrained by the limited design space of today's TCP incast setup, will our conclusions change using a faster link speed? Is the current performance model accurate without simulating the processor, the operating system, and the NIC?

As opposed to existing performance models, DIABLO aims to simulate the full-software stack at scale with great detail. The following sections will address several questions:

1. Can we reproduce the Incast throughput collapse effect with DIABO?
2. Is simulating computation and full software stack necessary to see TCP incast?
3. Do we need thousands of simulated instances to simulate rack-level networking effects? That is, is DIABLO an overkill for TCP incast?

5.2.1 Reproducing the TCP Incast Effect on Gigabit Switch with Shallow Port Buffers

The first validation of DIABLO is to see if we can successfully reproduce the application throughput collapse with shallow-buffer Gigabit switches used in many previous work. Without resynthesizing the FPGA, we configure the DIABLO packet-switching model at 1 Gbps link speed with 4KB packet buffer per port, found in Nortel 5500 switch [49]. The switch port-to-port delay is set to $1\mu s$. Compared to existing ns-2 and analytical switch models, DIABLO models a more advanced configurable packet buffer architecture that supports virtual-queues to prevent head-of-line blocking, which is usually used in high-end Cisco [16] or Fulcrum switches [55]. We did not use this advanced buffer architecture during our current experiments on TCP incast, which is more observable in low-end shared-buffer switches. However, we configure the packet buffer on our switch model to emulate cheap switches for a fair comparison.

The test program uses a workload in which a single client requests data from multiple servers, replicating the behavior of cluster-based storage systems, written by Berk Atikoglu and Tom Yue of Stanford University [6]. The same program has been used to test 10 Gbps production datacenter switches from Arista and Cisco [14]. We picked a typical request block size of 256KB for the client application. We configured DIABLO to simulate a single-issue 4 GHz CPU using our simple fixed CPI timing model.

The *goodput* of the client link is the number of useful bits transmitted over the network as seen by the application. We compile the Linux kernel to use TCP Reno with the default RTO_{min} at 200ms. The TCP and networking models in DIABLO are better than any of the existing simulation or analytical models, because it is from unmodified Linux implementation that includes every nuance of the kernel network stack.

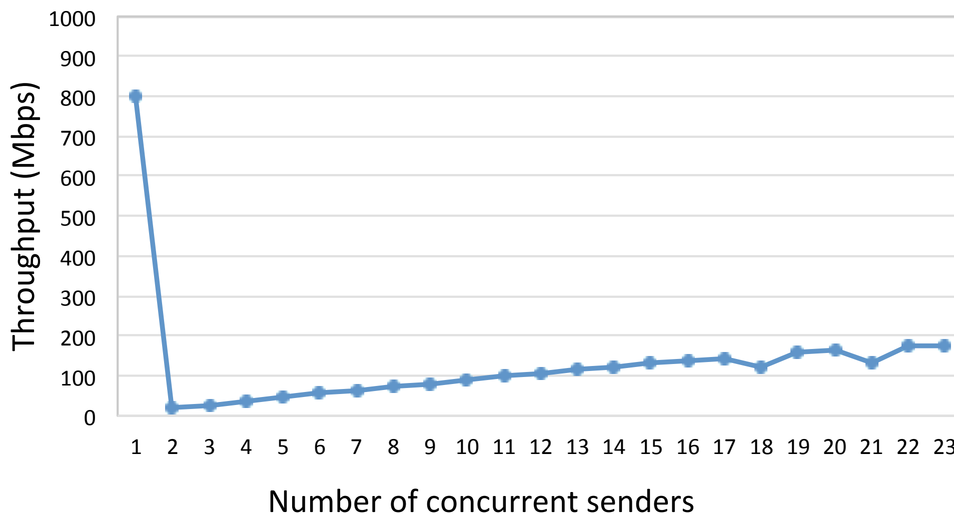


Figure 5.9. Reproducing the TCP incast effect with a shallow-buffer gigabit switch

We use up to 24 ports of the simulated switch, and run the network transaction for 40 iterations and average the goodput in Figure 5.9. DIABLO successful reproduced the application throughput collapse starting from 2 servers, found in measurement on real physical machines in [50]. After the collapse, the application throughput gradually recovers to over 150 Mbps. The simulated throughput before collapse also matches that on the real measurement at around 800 Mbps.

On the collapsed throughput, the absolute value of the simulated goodput is slightly lower than the real measurement, but we successfully reproduce the trend when increasing the number of servers. There could be many contributing factors to the absolute difference, such as details of NIC software, switch architecture, and host computing power.

5.2.2 Study TCP Incast at 10 Gbps Link Speed

Based on validated simple switch models at 1 Gbps, networking researchers have generalized their observations to 10 Gbps, emphasizing the TCP protocol and switch buffers only. In addition, prior work [126] makes unrealistic assumptions using an unrealistic request block size at 80 MB scaling to over two thousand servers on a single switch. Although these simple performance models seem to be reasonable at low link speed, they still face the question whether they are able to answer questions at other scales. With the help of DIABLO,

we found there are many factors that could significantly affect the application throughput at 10 Gbps link speed beyond TCP protocol and switch buffers.

In our experiments with DIABLO, we increased the switch per-port packet buffer to 16KB and lower the port-to-port latency to around 100ns to match the latest 10 Gbps cut-through switch specifications. In the incast application, we scale the requesting block size to 4MB to better utilize the increased bandwidth but still maintain a reasonable size for applications, where MB-size requests are common for distributed file systems [69]. Similar block sizes can also be found in performance testing used by industry for production switch benchmarking [2].

To investigate the impact of the operating system and kernel scheduler, we modify the TCP Incast benchmark using the *epoll* syscall, while the original program utilizes blocking socket syscalls inside server client threads created by *pthread*. The *epoll* syscall has been used by many datacenter applications, such as *memcached* to efficiently handle many client network connections. Applications using *epoll* have very different behavior compared to putting blocking syscalls in multiple user threads, where the application proactively polls the kernel for available data as opposed to waiting for OS to notify user threads. We also configure the processor timing model in DIABLO to simulate the processing effect of 2 GHz CPU versus a 4 GHz CPU.

To achieve an optimal network performance, switch benchmarks or storage network applications often choose a larger MTU (e.g. use the jumbo frame feature), which offsets the extra processing overheads by having fewer packets. However, Web 2.0 applications tend to use the standard 1500-Byte Ethernet MTU. For example, Twitter's datacenter is still running with an MTU of 1500 Byte. Therefore, we did not use a large MTU in our test. We plot the *goodput* curve of in Figure 5.10, with the right showing the results using the original *pthread* and the left showing the new *epoll* client.

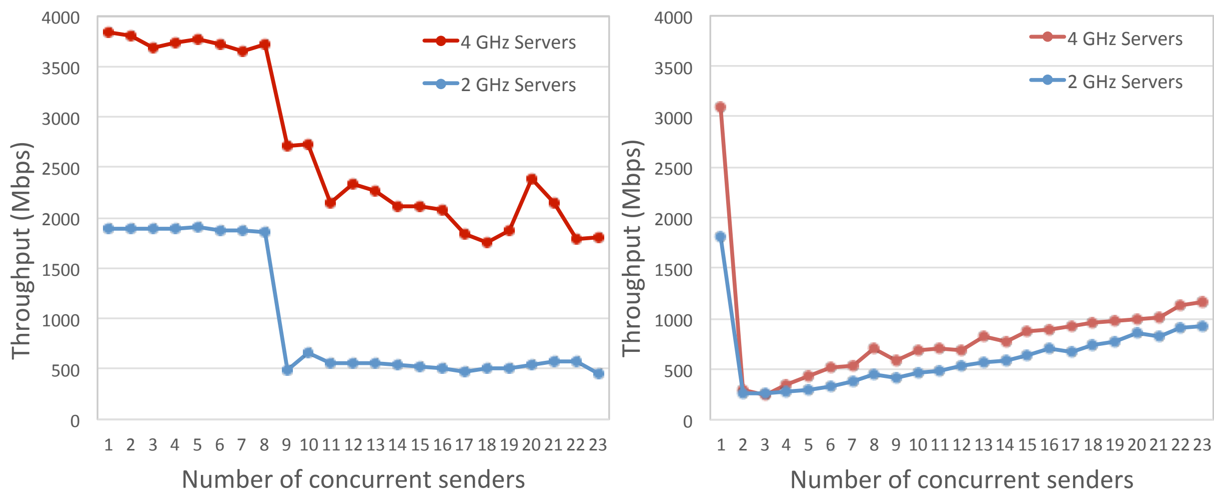


Figure 5.10. The left shows the throughput using the *epoll* syscall, whille the right utilizes the original *pthread*.

Under the same switch and TCP configuration, both graphs show some degree of applica-

tion throughput collapse. However, different host processing speed and choices of OS syscalls significantly affect the application throughput. With a 10 Gbps link, it is very challenging for applications to achieve the wire speed without significant effort. Even with the help of scather/gather zero-copy feature in the NIC, the simulated 2 GHz CPU could only achieves 1.8 Gbps throughput when there is no throughput collapse.

The use of *epoll* syscall significantly delays the onset of throughput collapse, we only observe a moderate throughput collapse to 2.7 Gbps starting from 9 servers and 1.8 Gbps with 23 servers on 4 GHz processors. On the slower hosts, the collapse is much more significant to $400 \sim 500$ Mbps, which is less than a third of the throughput on 4 GHz servers. The collapsed throughput with the *epoll* client is only half of that of the original *pthread* client. On the contrary, using the *pthread* version, the throughput collapses quickly even with a faster CPU. The throughput recovers to only 10% of the link capacity, which matches the observations from measurements and simulations in [126]. Moreover, the absolute throughput numbers do not seem to be correlated with the host processor performance when collapse happens.

Our results clearly suggest that there are many components in the software stack and even the host processor performance could significantly affect the TCP performance. As pointed out in an early work at the Fermi national lab with thousands of machines running 10 Gbps network for High Energy Physics (HEP) research [136], there are many factors in the Linux kernel that could lead to packet losses, such as different queues for slow and fast path handling of received packets. In addition, the kernel does not always notify the application immediately upon receiving a packet. Our head-to-head comparison between *epoll* and *pthread* clients shows that different choices of syscall have a strong impact on how kernel handling these internal queues. If any of the queues is full or is not drained in a timely fashion, the kernel can drop a packet that eventually triggers the TCP 200ms retransmission timer.

Moreover, modern high performance NIC drivers employ the new polling NAPI interface [111] to mitigate receiving interrupt load on CPUs. Basically, upon receiving a packet from the NIC RX interrupt, the driver turns off the hardware RX interrupt and tells the kernel to poll the driver at its convenience, for instance, rescheduling with softirq, traps, and exiting system calls. The downside of this performance optimization is that it makes the system behavior less deterministic. The overall performance is not only decided by the NIC hardware, but is also determined by how fast the kernel schedule a poll event. In our simulation with a slower host processor, we saw the NIC has to drop many incoming packets because the receive ring buffer allocated by the NIC drive is full due to a slower draining rate.

In conclusion, although packet losses trigger the TCP protocol retransmission that reduces application throughput, adjusting the TCP retransmission timer does not address the performance bottleneck in a system. The trip of a request from one application to another one on a different machine is a very complicated process even under the simple TCP Incast setting. Furthermore, there are many software buffers besides the hardware switch buffers. If the overall system is not balanced, a packet could be dropped at any place. Switch buffers are not always the one and only limiting factor. The long TCP retransmission timeout is just a consequence of an imbalanced system. Simple analytical or network simulation models

happen to work with faster machines connected with a slower network, but they cannot be safely extended to draw conclusions for a scaled-up system.

In terms of the solution of the TCP Incast problem, many proposals only focus on the interactions between the TCP protocol and switch buffers. We would like to raise two legit questions to prior work:

1. Are the design space limited by inaccurate simulations without looking at the problem from a whole system prospective?
2. Does reducing the retransmission time out really address the system bottleneck?

5.2.3 The Need of Simulating O(1,000) Nodes for Rack-Level Problems

Simulating the full software stack requires a significant amount of computing power. Moreover, in order to achieve stable simulation results, we need to perform the same experiment multiple times. For our 10 Gbps simulations, we ran 40 iterations for each data point up to 30 to 40 seconds in target time. This translates to hours in simulation. To plot a *goodput* curve from 2 to 24 machines, it requires moving about 40 GB in the simulated network. Additionally, when the design space is large including many knobs in addition to switch buffers and transport protocols, the simulation demand could be enormous.

In our experiments, we populated all six BEE3 boards with around 3,000 simulated hosts in 96 racks, simulating 8.6 billion instructions per second. Each FPGA simulates four racks of servers along with a top-of-the-rack switch. Although the raw simulation performance is 1000x slower than a real physical target, we can build a large number of nodes running in parallel with each generating different data points. This greatly compensates the slower simulation performance. For networking gear related performance tests, it is not very common to build a large scale testbed in practice. Usually, researchers have to run all desired experiments on the same small testbed sequentially. Any time-shared tricks such as cloud computing and virtual machines does not help here, because all of these performance tests require white-box testing to the underlying interconnect. Time multiplexing effects could introduce further nondeterministic undesired behaviors. This means it could still take a few hours to collect all data needed for a research paper on a small-scale real physical implementation. In contrast, with the great parallel simulation bandwidth of our six-board DIABLO, we can collect the same data overnight, which is not significantly longer.

To sum up, though DIABLO is designed to conduct experiment at scale beyond prototyping, the simulator scale offers great bandwidth that can be used to speed up rack-level simulations. This makes DIABLO a very practical platform for research problems at a smaller scale as well.

5.3 Case Three: Running *memcached*, a Distributed In-Memory Key-Value Stores

One of the important goals of DIABLO is to run production datacenter software. To demonstrate this feature, we run a popular distributed key-value store application, *memcached* [25], on DIABLO. Distributed in-memory key-value stores have become a central piece of many large website, such as Facebook, Zynga and Twitter. Essentially, these systems storing ordered *(key, value)* pairs are a distributed hash table. The key role of these systems is a cache for expensive-to-obtain values, usually disk-based back-end databases. The typical hit rates of production in-memory caching service in Facebook could be as high as up to 81% to 99% [39], suggesting that they are on the performance critical path of majorities of web requests.

In real datacenters, both *memcached* servers and clients (usually the web frontend) scale up to thousands of machines [9] connected with standard Ethernet-based datacenter interconnect. Previous work [92, 9] shows that the performance bottleneck of *memcached* are packet processing overhead in the NIC and the OS kernel networking stack, with some issues only showing at scales of thousands of machines. Clearly, running the full-software stack is a prerequisite to study *memcached* performance behavior. Cloud-based evaluation platforms do not work here, because the inefficiency I/O handling of its virtual machine based foundation technologies. As a result, these evaluation challenges make *memcached* a perfect showcase for DIABLO.

5.3.1 Experiment setup

Because of its popularity, *memcached* has been actively developed by both industry [3, 7] and the open source community. We use the latest vanilla *memcached* version 1.4.15 and compile it to SPARC without a single code change. We built our client using *libmemcached* 1.0.14 [5], which includes some simple application-level fault tolerant features such as timeout retry for UDP traffic.

We reuse the same simulated hardware configuration as that in our TCP incast experiment. In order to validate our results, we run *memcached* at both rack scale and a large scale up to 2,000 nodes for research purposes. The rack-level experiments are practical to validate with results from real physical machines.

For our rack-scale experiments, we simulate 4 GHz servers with our fixed CPI timing model connecting to a Gigabit switch with the virtual-output-queue packet buffer architecture. We equally partition the input-port packet buffer among all virtual queues on the same physical port, with each supporting up to 16 KB buffer space. We emulate a $1\mu s$ port-to-port latency. When conducting our scale-up experiment at 2,000 nodes, we also model a homogenous 10 Gbps interconnect with a low 100ns switch port-to-port latency in addition

to the Gigabit interconnect. All switches share the same packet buffer configuration as those in our 1 Gbps setup.

5.3.2 *memcached* Workload generator

In a real production environment, the client workloads of *memcached* servers are generated by front-end web tier. Although the API is simple, from a performance standpoint the *memcached* workload is more complex than it may appear in practice. In particular, previous studies show that the object size distribution has a large impact on the system behavior [92]. There are simple microbenchmark like tools such *memslap*, but they do not attempt to reproduce the statistical characteristics of the real traffic. Besides, *memslap* does not utilize the standard *libmemcached* API, but sends requests directly over socket.

In order to saturate the tested *memcached* server, researchers tend to pack hundreds of emulated clients on a single machine through separated TCP connections [92], or use fixed-rate dummy load generators [21].

Limited by the size of testbeds, researchers have no choice but to put the proposed improvements to *memcached* servers under an open-loop testing environment, ignoring any computation from client as well. However, from our experiments with DIABLO, we found that host computing performance does affect application throughput and service request latency. The client is part of the closed-loop system. Putting too many emulated clients could have negative impact on the overall application performance. In addition, the physical network data path from clients to servers would be drastically different from the perspective of real usage scenarios.

Although DIABLO is powerful enough to run a full set of frontend logic, without knowing the real user pattern from the Internet we build our own client based on a recent published Facebook live traffic characteristics [39]. At Facebook, *memcached* servers are divided based on the concept of *pools*. A pool is a partition of the entire key space, and typically represents a separate application or data domain to ensure adequate quality of service of each domain. One study [39] analyzes five pools of traffic:

- user-account status information (USR)
- object metadata of one application (APP)
- general-purpose and nonspecific(ETC)
- server-side browser information (VAR)
- system data on service location (SYS)

Among the five pools, the ETC is the most representative and accounts for the majority of overall traffic. Therefore, we build our client models for request key/value sizes and inter-arrival gaps based on the statistical models focusing only on the ETC traffic.

Random Generated Workload

Basically, our client load generator randomly picks a floating number between 0 and 1, using the current time stamp as the seed. We plug the random number into the inverse accumulative distribution function to get random variables we need. The unit for key and value size is byte, while the unit for inter-arrival gap is microsecond. There are three parameters to set:

1. *Key size*: The model for key sizes in bytes is Generalized Extreme Value distribution with parameters $\mu = 30.7984$, $\sigma = 8.20449$, k = 0.078688. The CDF of Generalized Extreme Value Distribution is:

$$CDF = e^{-t(x)} \quad (5.1)$$

$$t(x) = (1 + \xi z)^{-1/\xi} \quad (5.2)$$

$$z = \frac{x - \mu}{\sigma} \quad (5.3)$$

   We calculate the inverse of CDF:
   $x = \sigma \frac{(\frac{1}{-ln(p)})^k - 1}{k} + \mu$
   p is a random variable between 0 and 1 and x is the random key size value we need in our client program. Figure 5.11 plots the CDF of our generated data. We compare them against the data from [39] using the same scale. From the figure, we can see that our generated data match both Facebook key size statistics. In addition, most of the key sizes are less than 100 bytes.

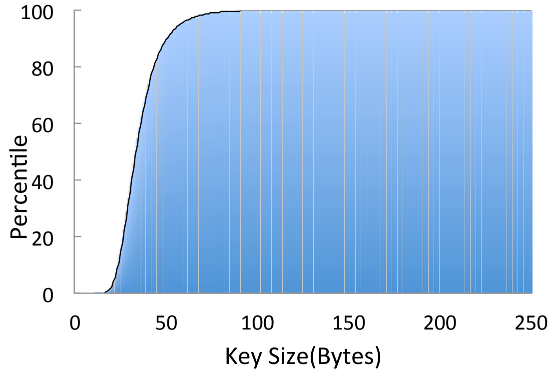
2. *Value size*: The model for value sizes in bytes is Generalized Pareto Distribution with parameters $\mu = 0$, $\sigma = 214.476$, k = 0.348238.

$$CDF = 1 - (1 + kz)^{-1/k} \quad (5.4)$$

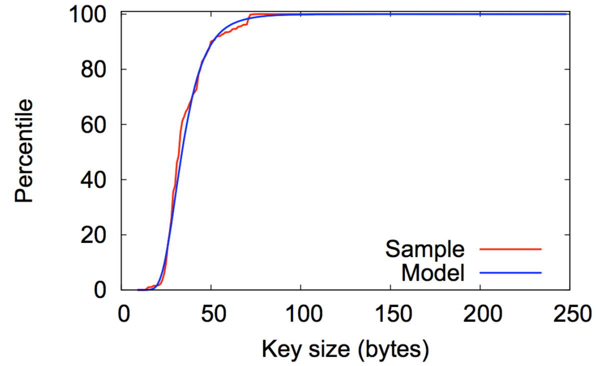
$$z = \frac{x - \mu}{\sigma} \quad (5.5)$$

   We calculate the inverse of CDF:
   $x = \sigma \frac{(\frac{1}{1-p})^k - 1}{k} + \mu$
   Similarly, plug in p and we can get the desired value size x. Figure 5.12 shows the CDF of our generated value size, compared to the Facebook workload model. We can see most of value sizes fall between 100 bytes and 1000 bytes.

3. *Inter-arrival gaps*: The model for inter-arrival gap is also Generalized Pareto distribution but with parameters $\mu = 0$, $\sigma = 16.0292$, k = 0.154971. Again, we plot the CDF of our generated inter-arrival gap comparing the Facebook data in Figure 5.13. Almost all of the inter-arrival gap variables are under 100 microseconds, we use *usleep* to model this gap in our client application emulating client application processing delays.

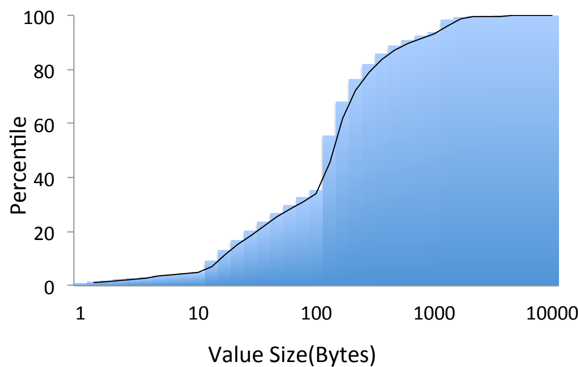


(a) Our model

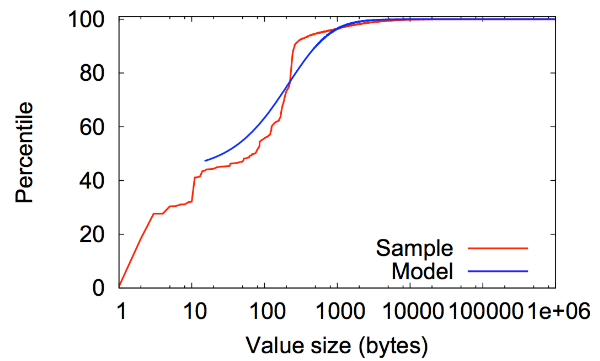


(b) Facebook

Figure 5.11. Compare CDF of the generated key size vs. Facebook traffic from the paper



(a) Our model



(b) Facebook

Figure 5.12. Compare CDF of the generated value size vs. Facebook traffic from the paper

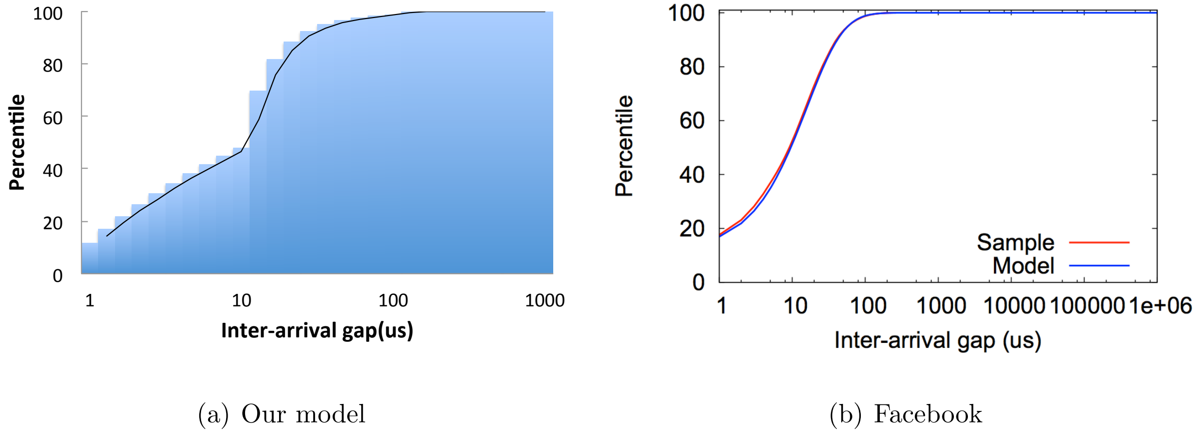
The client programs

There are four different client programs that we use with the *memcahed* server.

1. *Off-line traffic generator*: This program generates key/value sizes as well as the inter-arrival gaps employing the aforementioned techniques. It runs offline before any simulation and stores the generate data in three separate files on disk. We put these data input file into the disk image of each simulated node.

2. *data_init*: This program extracts pre-generated key files and warm up the memory pool of *memcached* servers before load testing can start. To improve the efficiency, each program utilizes 10 threads to warm up the server memory pool. When it finishes initializing the *memcached* server, it writes a predefined "done_key" in the server to signal the start of load testing.

3. *ping_memcached*: This program pings the servers using the *memstat* API every second. It also collects server load information, such as CPU and memory utilizations, through



(a) Our model

(b) Facebook

Figure 5.13. Compare CDF of the generated inter-arrival time vs. Facebook traffic from the paper

the */proc* file system every 200ms. In addition, it reports these statistics to the host control *appserver* on each FPGA through a dedicated Kernel driver with near zero target performance footprint.

4. *auto_memcached*: This program reads the same key files as those by *data_init*. It sends queries to random selected *memcached* servers. We partition the key space using pseudorandom prefixes generated from Linear Feedback Shift Registers (LFSRs). Since we do not have the full application logic, the application sends deterministic miss/hit requests according to published miss/hit ratios [39]. When a miss occurs, the client uses a fixed latency of 2ms emulating the delay hitting to a disk based storage. *auto_memcached* is also multithreaded, and each thread sends queries independently. It also maintains some basic performance statistics through simple counters, such as average request latencies.

5.3.3 Validating our single-rack *memcached* results

First, to validate our DIABLO models at an understandable scale, we deploy a set of *memcached* experiments at the rack-level scale of 16 machines connecting to a single gigabit switch. Like many other researchers with real physical implementations, we are limited by the availability of networking and computing hardware. Our physical testbed includes a 16-port Asante IntraCore 35516-T gigabit switch, and 16 3.0 GHz Intel Xeon D7950 servers running Linux 2.6.34. We use two of the servers as the *memcached* server with the rest as client.

To conduct our comparisons, we configure *memcached* with 64 MB memory pool, and let each client thread send 10,000 requests till completion. We also tried 100,000 requests with up to 256 MB server memory pool, and the steady-state performance are similar. On real machines, we use 100,000 requests but sample the results of the middle 10,000 requests.

Because we launched our jobs through scripts using SSH, which do not have a simultaneous start behavior. As a results, the client on physical machines takes slightly longer to reach a steady state. We run several configuration combinations of servers and clients. For instance, we run each server with 4 or 8 worker threads using TCP or UDP connections. Each client runs with one, two, four and up to eight worker threads. We compare performance of the two systems from perspectives of both server and client below.

Server performance

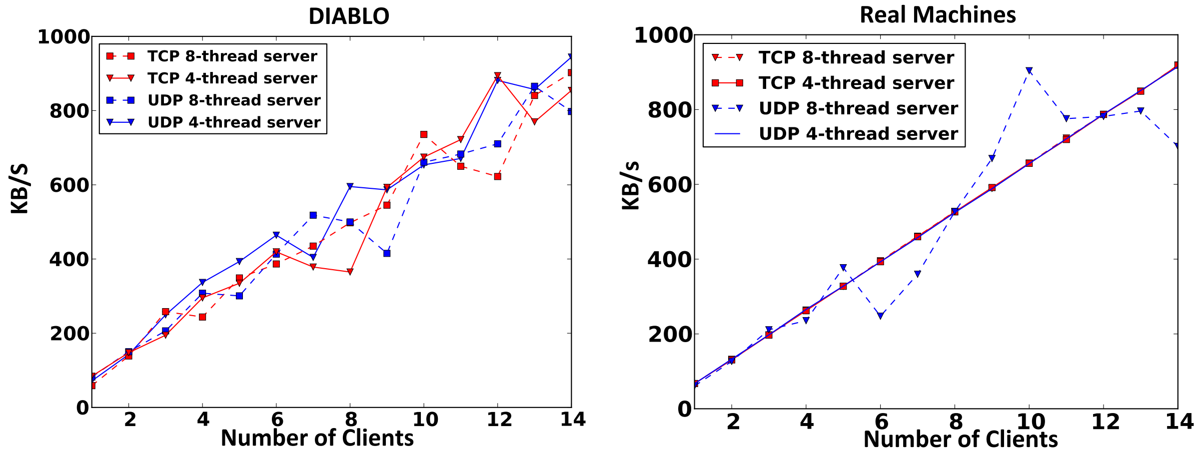
We choose to plot our results from only one server, as the other one is very similar. Figure 5.14 shows the *memcached* server throughput under different number of clients. The right is the application throughput measured on the real cluster, and the left is the result from DIABLO. To demonstrate impact of server performance, Figure 5.15 plots the server CPU utilization for 1, 7, and 14 clients configurations. We show the results using both TCP and UDP. For all the configurations we have tested at this small scale, there is no significant difference between the two networking protocols. TCP has a slightly better throughput than UDP. Having more clients helps to saturate the server faster. There are no big differences changing the number of server threads. Our simulation shows when the server CPU load is high, having more threads hurts the overall performance because of the *pthread* overhead.

There are absolute performance differences between simulated clusters and real clusters. However, both throughput numbers have the same order of magnitude. There are many factors that contribute this absolute difference. First, the 3.0 GHz Xeon CPU is a wide issue out-of-order super-scale x86 core with hyper-threading, but we simulate a 4.0 GHz single issue CPU with a fixed CPI. The Linux bogoMIPS is around 6,000 per virtual CPU, while the simulated CPU only has a bogoMIPS of 4,000. The server CPU utilization data also suggests that we are simulating a slower processor, with a lower server CPU utilization number on the real Xeon server. Second, the switch and NIC are different on the real machines. The subtle architecture differences could also affect the overall performance.

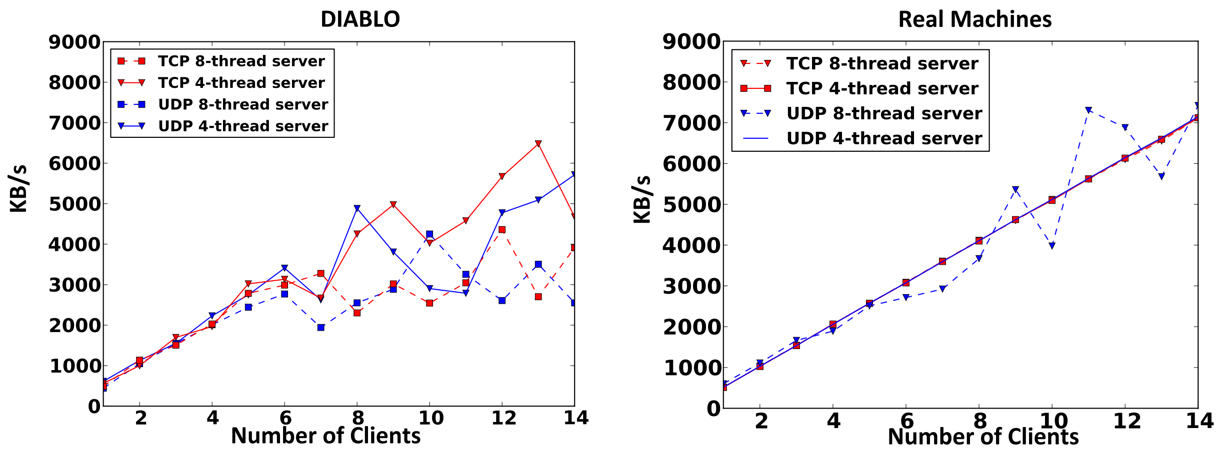
Most importantly, the DIABLO simulation reproduces the trend of server throughput. Having a full software stack, we can also see some similar system behaviors on DIABLO. For example, we see the system memory utilization gradually increased over time while running the *memcached* server program.

Client request latencies

Figure 5.16 illustrates the average request latencies measured at every client thread. Like the results from real machines, the simulated cluster shows the trend of increased latencies with more clients. For single-thread clients, both UDP and TCP requests stay low under 80 microseconds on either real or simulated cluster. The client latency stays low and scaled linearly with a small number of clients. There is a “break-out” point after 6 to 7 clients,



(a) 1-thread clients



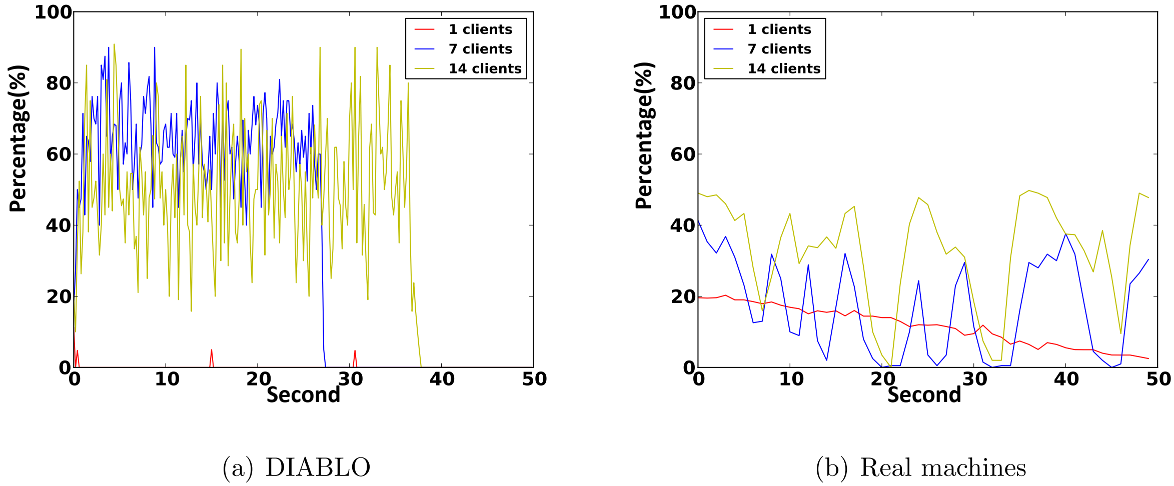
(b) 8-thread clients

Figure 5.14. Simulated memcahed server throughput vs. Measured throughput on real machines. Left is the results from DIABLO. Right is from real machines

while the simulated server CPU is closing to the 100%, as seen in Figure 5.15. The average client request latency degrades to over 1 millisecond under the 8-thread 14 clients setup.

Overall, our simulation results match observations in existing literatures that *memcached* is inefficient for not saturating network but limited by CPU performance on network packet processing [92]. To support our claim, we also plot the percentage of kernel CPU usage of *memcached* servers for 1, 7, and 14 clients in Figure 5.17. Both simulation and real machine results show that *memcached* spent a significant number of CPU cycles in the kernel.

Again, if the performance bottleneck appears to be in the application and operating system, reproducing the end-to-end application performance would be very difficult without modeling the computation of full software stack, especially when the software stack is constantly changing.



(a) DIABLO

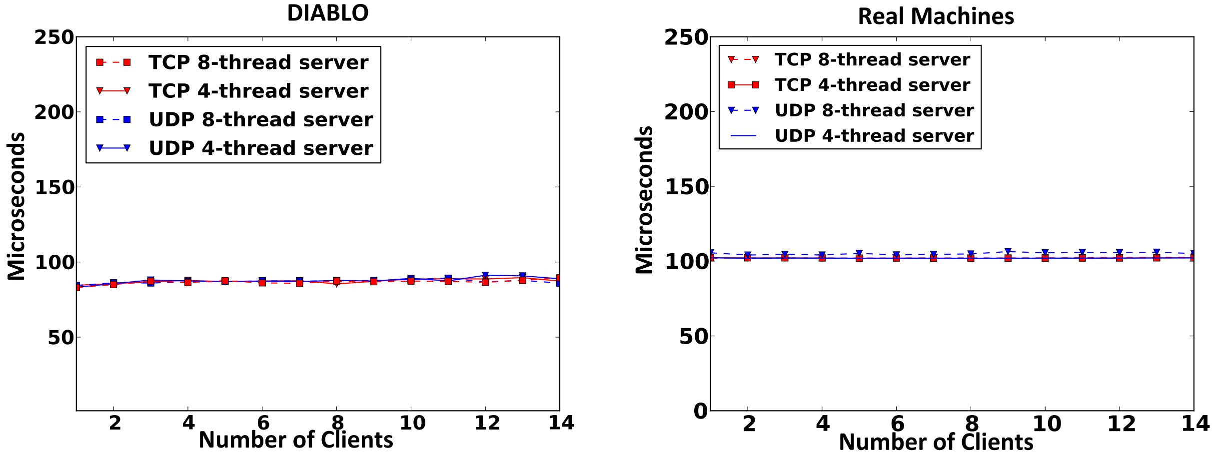
(b) Real machines

Figure 5.15. CPU utilization over time on simulated memcached servers and real machines

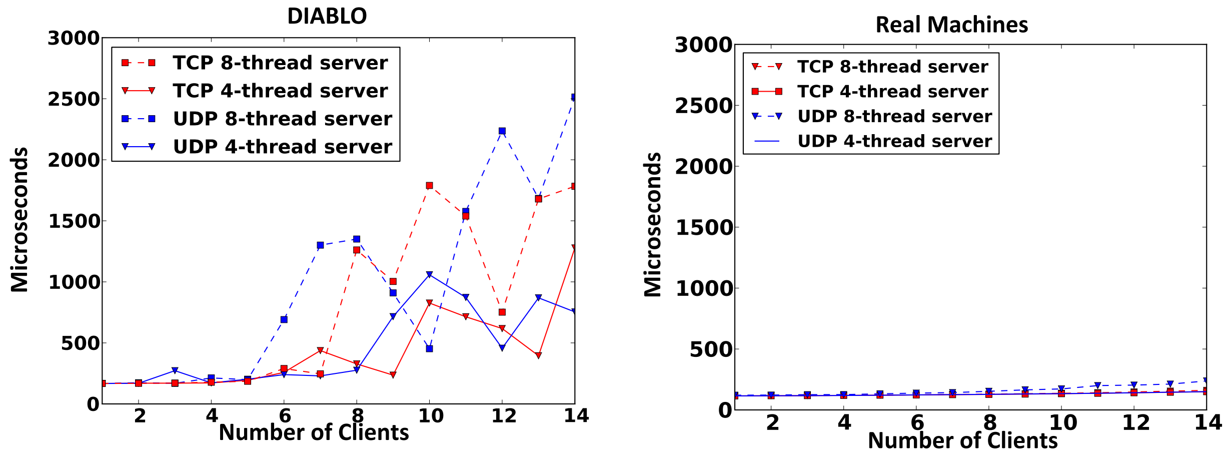
5.3.4 Large-scale Experiments

To show the scalability of DIABLO, we deploy the same *memcached* load test across all six BEE3 boards. We use four BEE3 boards to simulate up to 64 racks of 1,984 servers, connected with a three-level network described in Section 4.5. One BEE3 board is dedicated to array switch simulations, and the rest one to the datacenter switch. Each rack contains 31 servers with one rack switch. We use the 32nd port on the rack switch to connect to an array switch, creating a bandwidth over-subscription ratio of 31 to 1. Each array switch supports up to 16 inward facing links and one uplink to the datacenter switch, thus having a bandwidth over-subscription ratio of 16-to-1. To perform the scalability test, we simulate a 1 Gbps interconnect with one microsecond port-port latency switches as well as a 10 Gbps interconnect using switches with 100 nanosecond port-port latency. All switches share the same buffer configuration as those used in our rack-level experiments. From the rack-level experiments in the previous section, we know that *memcached* is a latency-bounded application. The goal of simulating a low-latency interconnect is to explore the impact of new switch hardware on latency sensitive applications at large scale.

We distributed 128 *memcached* servers evenly across all 64 racks to minimize potential hot spots in the network, and use the rest machines as clients. This creates a configuration of two *memcached* servers and 29 clients in a single rack. Each client sends 10K requests to a randomly selected sever from the 128-server pool. We perform our experiments at several scales: 496-node, 992-node, and 1984-node. We also proportionally scale down the number of servers when running at a smaller configuration. For instance, there are 32 servers for the 496-node configuration and 64 servers for the 992-node configuration. The 496-node setup uses only one 16-port array switch without a datacenter switch. Both 992-node and 1984-node experiments exercise all three levels of switches. We also perform our load tests using both TCP and UDP protocol. For convenience of representing simulation scales, we



(a) 1-thread clients



(b) 8-thread clients

Figure 5.16. Average client request latencies on DIABLO (left) vs. on real machines (right)

round up the exact number of nodes to 500, 1000 and 2000 respectively for the three setups in the following presentation.

Server statistics at scale

With our application-level random load-balancing, all servers are evenly loaded at moderate CPU utilization. Since the CPU utilization is very spiky, we use the median of all sampled CPU utilizations. Figure 5.18 illustrates the average of all median CPU utilizations from all servers under different configurations at various scales. We can see that doubling the servers effectively keeps the sever load low when servicing twice the number of clients. The

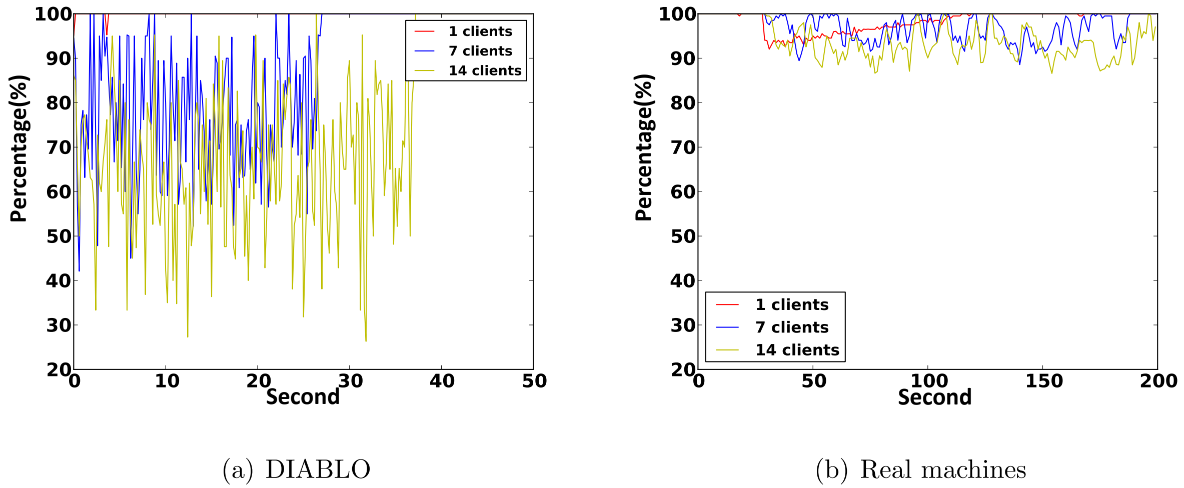


Figure 5.17. Percentage of kernel time of memcached servers

servers exhibit similar behavior with both TCP and UDP. For this particular application, using a faster 10 Gbps interconnect has a very small impact on the server CPU utilization under all scales we have tested regardless of the choice of network transport protocols.

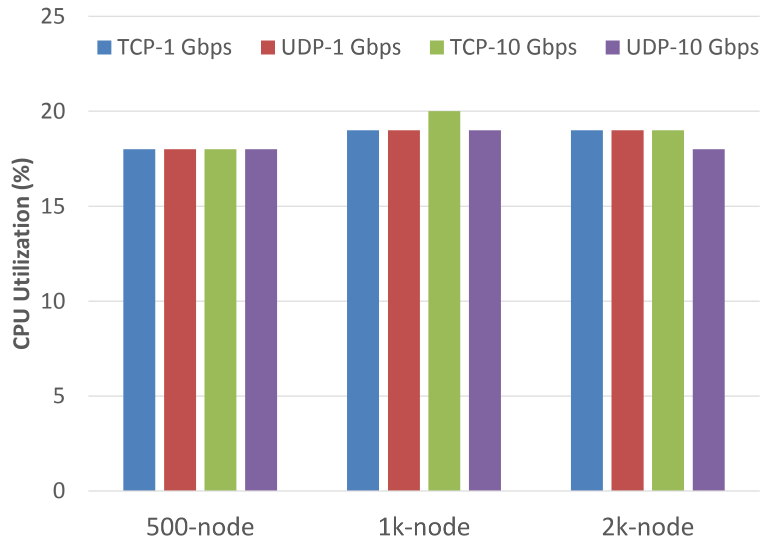


Figure 5.18. Average CPU utilization (median of all samples) per server at different scales

Figure 5.19 displays the average of the minimum free memory per server. Both TCP and UDP demonstrate similar server memory utilizations. As expected, the server consumes more memory when handling more clients. The average of minimum free memory drops from 82 MB to 75 MB when scaling from 500 to 1K nodes. However, we do not observe significant free memory drop when scaling to 2K nodes. Due to the limited number of nodes, the 500-node setup utilizes a two-level tree hierarchy as opposed to three levels. This suggests

that changing interconnect hierarchy could have potential impact on flow dynamics that eventually yields differences in server memory utilization.

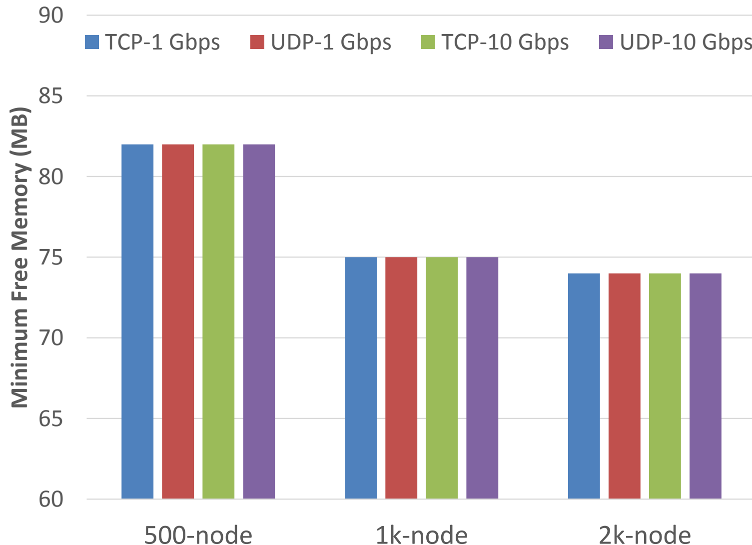


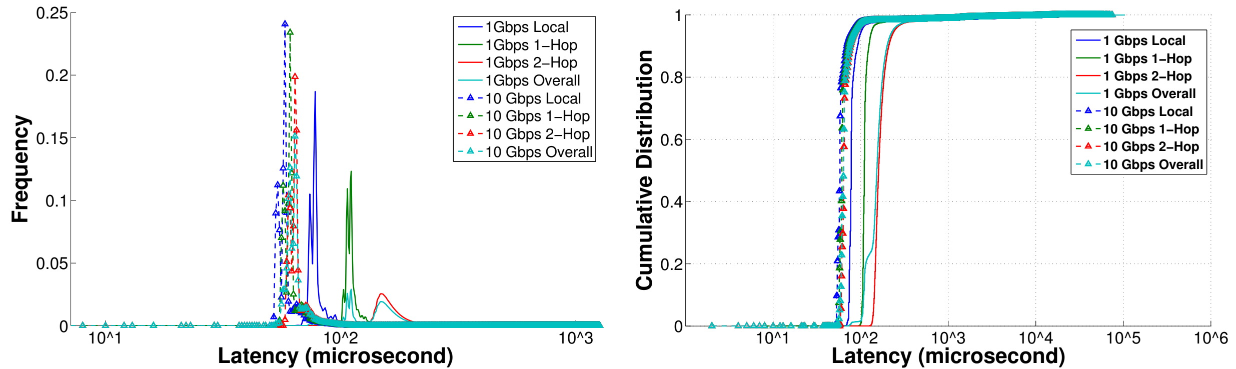
Figure 5.19. Average minimum free memory per server at different scales

Reproducing the request latency long tail

Under all configurations, we did not see any packet dropped due to congested buffers in either switches or NICs. One interesting thing to see is whether or not all client queries finish quickly at large scale at around one hundred microseconds like the single rack light-loaded case. As an illustration, Figure 5.20 plots the Probability Mass Function (PMF) and the Cumulative Distribution Function (CDF) of all client queries for our 2k-node setup using different interconnect running the UDP protocol. The shapes of both PMF and CDF are similar at the scale of 500-node and 1k-node. We also observed similar latency long tails using TCP.

We found the majority of requests finished in less than $100\mu s$, but there are a small number of requests that complete more than two orders of magnitude slower than the average, forming a long tail distribution. Such latency long tail behavior has been reported on real clusters [44]. Experiments have demonstrated such increased latencies can negatively impact user satisfaction leading to significant financial loss [85]. It has become an increasingly interesting topic among datacenter researchers.

To understand the long tail better, Figure 5.20 classifies all queries into three different categories based on the number of physical switches they traverse. The data series marked with *local* means that the request is made to the server physically located in the same rack. *1-hop* means a request that has to go through one array switch to reach a server in a remote rack. *2-hop* means a request that needs to traverse the datacenter switch to reach a remote server. From Figure 5.20, we know that all three types of requests exhibit a long



(a) Probability Mass Function (PMF)

(b) Cumulative Distribution Function (CDF)

Figure 5.20. PMF and CDF of client request latency at 2000-node using UDP

tail distribution. Moreover, 2-hop requests dominate the overall latency distribution at large scale. This conclusion is very intuitive for our simple random-select load balancing policy, as with more nodes the less likely a request hits the server that is in the same rack. Besides, we found that when a request traverses more switches in the system the more variations the latency has. This suggests that using a multi-hop network architecture could potentially making the latency long tail problem worse.

One interesting comparison between the simulated 10 Gbps and 1 Gbps interconnect is that low-latency high-bandwidth switch does help on the latency long tail issue. There are more requests that finish faster, at around 70 microseconds. In addition, the latency differences between all three types of traffic are small, shown as more clustered dash lines in Figure 5.20. Although the 10 Gbps interconnect employs switches with 100 nanosecond port-port latency that is 10 times better than the 1 Gbps interconnect, the low latency interconnect does not improve the application request latency by a factor of 10. This is largely because of the OS software processing overhead. Instead, the latency gain is less than 2×, which is reported to be seen on real-world clusters by Google [44].

Impact of system scale on request latency long tail

Another interesting thing to look at is the impact of the system scale on the request latency long tail. Figure 5.21 plots the zoom-in CDF curves between 0.96 and 1 of the cumulative distribution, focusing on the tail. We have tested many configuration combinations, but we only show the one with UDP protocol on 10 Gbps interconnect as an illustration, because other configurations show similar trends. From the graph, we can tell that the latency for most of requests does not increase significantly when scaling up. However, there are more requests falling into the tail. In order words, the latency long tail is more visible at a larger scale.

To demonstrate DIABLO's capability for design space exploration at large scale, we conduct a very simple experiment. It quantitatively analyzes which transport protocol (TCP

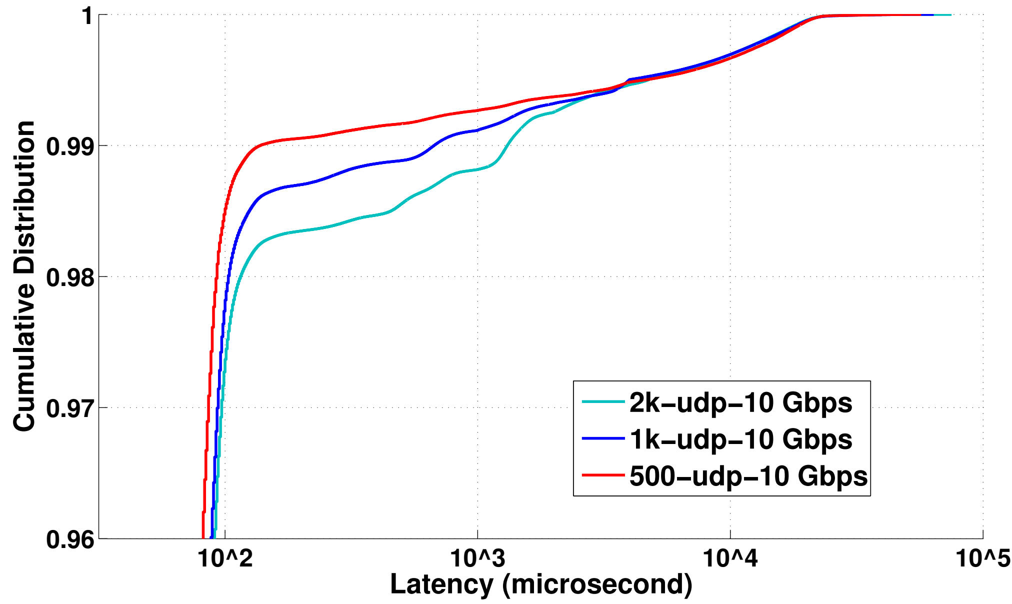
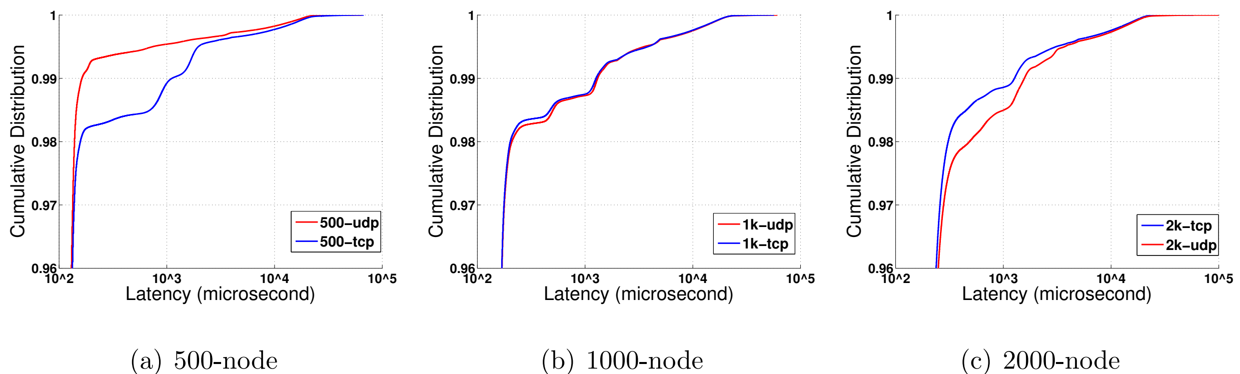


Figure 5.21. Impact of system scales on the latency long tail

or UDP) is better at large scales minimizing the long tail effect. Since scales of a few hundred nodes is a popular upper-bound for many academic research testbed, we would like to address one question: can we generalize the results at O(100) nodes to a larger scale at O(1000)?

To better understand this simple research question, we first simulated the *memcached* with the 1 Gbps interconnect. Figure 5.22 shows the cumulative tail distribution of using different protocols. At the 500-node scale, the UDP protocol is a clear win compared to TCP. However, the advantage of UDP drops when moving to 1000-node, whereas TCP slightly outperforms UDP. When we move to the 2000-node scale, it appears that TCP is a better protocol. The conclusion at 2000-node scale is completely reversed compared to that of 500-node.

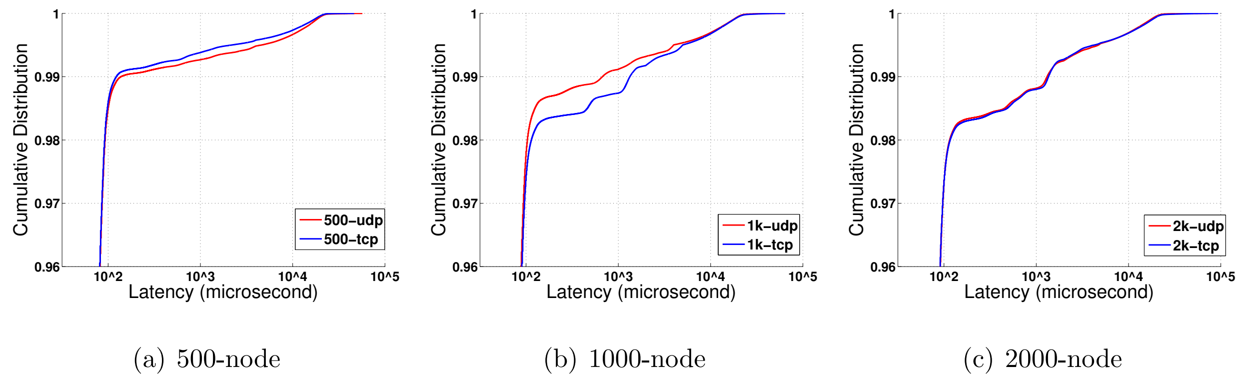


(a) 500-node (b) 1000-node (c) 2000-node

Figure 5.22. Comparing TCP vs UDP on cumulative tail distributions of client request latency at different scale with the 1-Gbps interconnect

Due to the limited availability and cost of new hardware, researchers typically employ

off-the-shelf hardware for software development, and assume the same conclusion holds on the new hardware as well. We emulate this usage model by simulating the same software configuration on the 10 Gbps interconnect in DIABLO. We plot the same CDF comparison graph as in Figure 5.23. The answers of which protocol is better are drastically different from those for the 1 Gbps interconnect. At the 500-node scale, TCP works slightly better. At the 1000-node scale, UDP is a better choice. When scaling to 2000-node, there is no significant difference between TCP and UDP at all. Our results show that the latency long tail is a very complicated issue with nonlinear behaviors. One cannot simply extrapolate results from a few hundred of nodes to a few thousand nodes. The same conclusion could also be different if the underlying hardware has been changed.



(a) 500-node (b) 1000-node (c) 2000-node

Figure 5.23. Comparing TCP vs UDP on cumulative tail distributions of client request latency at different scale with the 10-Gbps interconnect

Summary

At the scale 2,000 nodes and 69 switches, it is not practical to have a private-owned physical testbed and hold for a few hours to do any software and hardware design space explorations. For instance, if researchers propose a change to the kernel software to improve the *memcached* networking stack efficiency like those in [9], deploying these customized kernel to 2,000 machines would be a challenging task by itself. Therefore, to the best of our knowledge, we found no alternative solution to generate equivalent performance data in practice other than from DIABLO. We also demonstrate that at large scales the system could behave considerably different from systems at smaller scales. We also show that such phenomenon could be very unintuitive given the complexity of datacenter software stack.

5.4 Conclusion

In this chapter, we demonstrated three real-life use cases for DIABLO. DIABLO cannot only be used to model a novel network research proposal, but also perform hardware and

software co-tuning of existing hardware. The goal of DIABLO is not to model an existing datacenter with 100% accuracy, but rather unveil and discover the system performance scalability trends. From our validation results with real clusters, we show that at rack level DIABLO can successfully reproduce application performance behavior. We also found that DIABLO is an excellent number-crunching machine that generates data for rack-level experiments with a scale of 96 simulated server racks. Moreover, being able to simulate server computation and full operating systems imposes less evaluation constraints for better insights on application behavior from the full-system point of view. Researchers would be less likely limited to a few components, such as switch and network protocols. Our 2,000-node *memcached* experiment shows the great potentials of DIABLO targeting research at datacenter scale that none of existing evaluation platforms could possibly achieve.

Chapter 6

Experiences and Lessons Learned

In this chapter, we discuss experiences and lessons learned while building DIABLO. The purpose of the chapter is to provide insights for building future DIABLO-like FAME-7 simulators. We also point out directions for possible improvements in CAD tools that are essential to the productivity of FAME developers.

6.1 Core selection

In DIABLO, the choice of SPARC v8 as the ISA of our main processor core was not ad-hoc but a deliberate decision. There are several metrics we considered when picking the ISA and its implementation.

- *A standardized ISA that has an existing compiler and software infrastructure.* The goal of this testbed is to run full open-source datacenter software stacks, such as LAMP (Linux, Apache, Mysql, PHP). Leveraging existing GCC and Linux ports can greatly shorten the overall development time. Initially, we chose to use an FPGA-optimized soft-core from Xilinx called MicroBlaze. Unfortunately, the MicroBlaze compiler toolchain could not compile many off-the-shelf open-source software packages without significant modifications of makefiles and configuration scripts. Besides, the initial version of Microblaze did not have an MMU, so could only run shrunk down versions of Linux for embedded devices, such as uClinux.
- *A RISC ISA that can comfortably fit in a modern FPGA with reasonable resource consumptions and moderate implementation complexities.* Ideally, x86 is perfect to run any off-the-shelf datacenter software binary. However, both the implementation complexity and the resource consumption of an x86 FPGA implementation are significant.

In addition, most datacenter software is open source, and written in high-level languages. Thus, the ISA plays a less important role in functionality and performance. Initially, we considered acquiring a commercial RISC ISA with an existing open-source implementation. Unfortunately, the RTL implemenations we got from industry such as SPARC T1 and PowerPC 405 are derived from ASIC implementations, which map poorly to FPGAs. Moreover, even if we have the source code, they are not straightforward to understand. Further, it is very hard to modify the core according to our needs, for example, supporting network I/O for emulation.

- *An in-house open-source implementation with minimum third-party IP cores.* Because of less rigorous verification efforts, implementations of many commercial FPGA IP cores from FPGA vendors are poor quality, particularly low reliability. These IP cores work for basic applications with simple access patterns. Running server software on a real operating system could easily exercise corner cases in any implementation. If we built DIABLO from many third-party components, it is hard for us to pinpoint any hardware issue that causes a piece of complex software to fail without an extensive verification suite for every building block. On the other hand, having full control of the source code, we can easily modify an in-house implementation. This is extremely useful, when we need to change the hardware to have a better OS kernel and device driver support.

- *A complete verification suite.* Given the aforementioned constraints, there are not many off-the-shelf candidate cores that do not require significant modification. Therefore, we had to develop our own core. A complete ISA verification suite is crucial to verify the functional correctness of our customized implementation. As SPARC v8 is an open standard, we acquired the verification suite through a donation from SPARC international. The same verification suite is used as the SPARC certification test, which gave us great confidence while bringing up the processor core in the early design stages.

Although a 64-bit ISA is ideal for server applications to address more than 4 GB memory, 64-bit ISA implementations on FPGAs are intrinsically more complex than those of 32-bit ISAs and consume more FPGA resources. Besides, the number of 64-bit cores we can fit on the FPGA board we were using is very limited. Conflicting with our scale requirement of simulating enough datacenter servers. Owing to this practical hardware capacity constraint, we chose the 32-bit SPARC ISA as the proof of concept of DIABLO. Since FPGA capacity has been significantly increased with recent processor technology, supporting 64-bit processors emulating more server memory should be possible in future work.

6.2 Design languages and Impact of FPGA ECAD Tools

We implement the DIABLO hardware using the Systemverilog hardware description language (HDL), which is a major upgrade from the Verilog standard since 2001. Systemverilog was considered a new standardized hardware design and verification language at the time we started the project. Part of the DIABLO design, RAMP Gold, was also credited by industry FPGA logic synthesis developers as "one of the largest Systemverilog designs in the open-source community that exercises many advanced Systemverilog language constructs" [68]. However, from an early adopter's perspective, our experiences with the language were mixed.

Systemverilog introduces several high-level synthesizable language constructs for design, which make it closer to the C programming language compared to the original Verilog standard. Systemverilog remains a superset of the popular Verilog standard, but the small language feature changes make it more descriptive behaviorally. For example, the new *struct*, *union* and *library* keyword greatly reduce source code size. Although Systemverilog, like its Verilog predecessor, is not a strongly-typed language, we do not feel type-safety is a must-have feature building synthesizable high-level abstracted FAME models. On the other hand, the flexibility of implicit type-casting makes mapping high-level data structures to simple hardware structures more convenient. Therefore, we do not consider lack of strong typing a lethal drawback in the language. Most logic synthesis or simulation tools can check bit width and issue warnings for type mismatch. Carefully mining the compiler log would be sufficient for any users who are proficient in non-strong typing languages, such as C/C++.

Systemverilog introduce many new features for RTL verification. In practice, we find the support of assertion and coverage extremely useful for functional verification using traditional Verilog simulators. On average, every unit test in DIABLO includes 40-50 assertions to validate the functional and timing behavior of the unit. These assertions not only detect logic behavior at each instant in time but also over a period of time. Any potential functional bug could be easily captured by one or a collection of these assertions. Furthermore, measuring coverage provides quantitative confidence while running testbenches. This standard language feature offers more flexibility and better performance compared to similar features in Verilog simulator implementations.

Systemverilog also provides an object-oriented programming model for verifications, which is more closely related to Java than C++. However, the simulator support of many object-oriented features is still in its infancy, making the object-oriented features bells and whistles in reality. As a substitute, we implement all complex testbench logic in C++, which interact with our RTL design through the simple low-overhead Systemverilog Direct Programming Interface (DPI).

In order to address many intrinsic verification issues that Verilog has, such as race conditions, Systemverilog employs very complicated event-based simulation scheduling semantics

with 17 ordered regions. This makes it difficult to write the RTL code that simulates rapidly without significant effort.

Systemverilog is an open industry standard that is adopted by all major EDA vendors. It is therefore very convenient to run the same design through tools from different vendors to gain confidence about the correctness of results from a specific tool.

Overall, Systemverilog is a reasonable language to implement high-level FAME models. We find the small enhancements over traditional Verilog improve our productivity noticeably. However, due to its Verilog origin, Systemverilog still suffers from many problems that the Verilog language has in both simulation and synthesis. It still requires great effort to work on low-level individual wires at the cycle level in order to debug functionality of a large design, which is far more labor-intensive process compared to doing an equivalent software implementation in high-level languages.

According to our conversations with industry CAD developers, there is a strong incentive to standardize and unify design and verification language from the tool developer point of view. For user perspective, standardization is important for leveraging infrastructure investments of the semiconductor and CAD industries. On the other hand, we must be aware that some seemingly obvious standardization might not be desirable in practice. "Which design language to use?" is similarly to the classic debates between users of "Vi or Emacs", which exemplifies the fact of human nature that we become attached to seemingly irrelevant differences. We do not against the use of any non-standardized high-level design language. Besides, in general some decisions cannot be left to the whim of the individuals. As an individual graduate research project, we choose the most comfortable language. Taking a broader perspective, however, it may not be productive or reasonable for different design teams.

6.3 ECAD Issues

A much bigger problem than simulator design and debugging is the poor state of FPGA CAD tools, which are much worse than ASIC tools. During the first two years of development of DIABLO, we encountered 78 formally tracked bugs in four tools from two companies, ranging from logic synthesis and simulation to formal verification. The versions of these tools were all the latest production releases from year 2007 and 2012. During the development process, we cooperated with development teams from three major FPGA logic synthesis vendors. For a very long time, we had to rely on internal 1-off alpha builds to make progress. Although the turnaround time to get tool bugs fixed using our special connections to developers was relative short and the implementation quality of these tools keeps improving over the time, our overall negative impression of these tools remained the same throughout the years. We summarize our experiences as the following:

1. Most of the bugs affected the functionality of the tool, while only a few bugs affected the quality of result (QoR).

2. As a new language, support for new language constructs and advanced parameterizations are the most common issues for Systemverilog tools. According to our interactions with industry CAD developers, many of the problems are mainly caused by lack of test cases to verify the tool implementation.

3. It is very common that logic synthesis tools silently generate bad logic, in addition to obvious infamous compiler segmentation faults. Therefore, post-synthesis verification is a must. When building complex FAME models on FPGAs, we faced many similar verification challenges caused by design complexities to those seen by ASIC designers.

4. Ideally, formal verification tools are designed to verify the correctness of any CAD result. Unfortunately, they only work well comparing incremental changes in low-level gate net lists, which renders them useless comparing differences between a synthesized net list and a behavioral RTL implementation. One reason is that the formal verification tool usually shares a compiler frontend with the logic synthesis tool of the same vendor. If a bug occurs in the frontend of the synthesis tool, very likely the same bug will plague the formal verification tool as well. Besides, there are many false positives due to practical engineering issues using formal verification tools from different vendors, like signal naming conventions and advanced logic optimizations.

5. Although the newness of the Systemverilog language appears to be a large contributing factor to tool bugs, the backend tool that maps logic to FPGA primitives has never been free of bugs. Moreover, logic synthesis tools always have trouble taking advantage of newly introduced FPGA primitives in the first few years after a new device is announced.

6. Verilog simulator bugs are the number one issue that hampers our productivity when debugging RTL designs. Due to the complexity of the Verilog event-execution semantics, implementing a correct and fast Verilog simulator is very challenging. Although many Verilog simulators provide various performance optimizations that could speed up the simulation by $2-3$x, we always have to turn off all of these optimizations in practice just to avoid simulator bugs. In addition, debugging high-level functionality of the design using wave forms is tedious.

7. Regarding the quality of result, FPGA synthesis, place and route tools “just work” but do not work well. FPGA CAD vendors boast of many premium optimization features, such as placement-aware physical synthesis, which could cost users tens of thousands dollars extra per year. We found these features very hard to use, and the results we achieved seldom matched what vendors have claimed. When design constraints get more complicated, these fancy features break easily and have serious interoperability issues when using with tools from different vendors. We are constantly forced to run the CAD flow with the most basic setup, leaving behind great potential of improving the design performance by just employing better tools without changing the source code.

8. Modern FPGAs support designs with multiple clock domains. Large FPGAs have been supporting as many as 16-32 global single-cycle clock networks along with many

regional clocks for many generations. In general, multiple-clock designs are very common for large system designs. However, both FPGA logic synthesis and place and route tools have trouble parsing complex cross-clock domain timing constraints and performing even some very basic optimizations, such as retiming.

6.4 Building a Reliable Simulation Infrastructure

As a simulation platform, we designed DIABLO to run a few days reliably without any error. In DIABLO, there are many components at various levels that need reliability features built in. Some are at the higher system level, with others are at the circuit level.

6.4.1 Reliable Control Protocol

In DIABLO, the control software controls every FPGA through a Gigabit Ethernet connection, called the frontend link. Semantically, the frontend link does not allow dropping a single packet. The frontend link only sends moderate control traffic using less than 10 Mbps bandwidth on a Gigabit link. In the beginning, we assumed both the commercial Ethernet switch and our PC server are fast enough. Therefore, there should be no data loss on our control link. However, Ethernet is not a lossless protocol. Both the operating system software stack and the switch hardware could drop packets when there is a resource contention. Once we hooked more DIABLO nodes to a single frontend control computer, we started seeing packet loss regardless of the low aggregated bandwidth requirement on a fast link. Finally, we implemented a simple sequence-number based hardware retry mechanism to ensure a lossless transmission, which is similar to a TCP protocol with congestion windows size of one.

6.4.2 Reliable Inter-chip Links

DIABLO is a modularized design using high-speed serial transceivers to connect FPGAs. We partition simulated target components into different FPGAs, and they require exchanging data through these serial links in every simulation quanta without errors. Due to the nature of high-speed serial links, the physical medium is not error-free. Hence, a high-level reliable link layer protocol is necessary.

The physical transceivers on FPGAs support implementing multiple popular serial protocols, such as PCI express, SATA, XAUI etc. One straightforward implementation is to pick an off-the-shelf industry protocol with reliable transmission features such as PCI express and SATA. Unfortunately, these protocols are not designed for architecture simulations, and they have more overheads. For example, PCI express is a packet-based protocol. Not every field of the packet header is useful in our usage context. Similarly, the SATA protocol defines

more control commands than necessary. Additionally, some protocols include features to support bundling multiple physical channels into a single wide aggregated logical channel, such as the periodical channel alignment in XAUI, which are useless in the usage scenario of DIABLO, where transceivers are used independently.

Another issue we find in existing protocols is that some of the reliability features are really designed with specific target hardware platform in mind. For instance, the interval of sending clock recovery control sequences in protocols like XAUI assumes a link topology with typical delays in hypothetical line cards and backplanes. This extra complexity might be overkill given the simple point-to-point link topology in DIABLO.

An alternative is to choose a simpler protocol provided by FPGA vendors that has free implementations. FPGA vendors also provide tools to evaluate the *bit-error-rate (BER)* of their implementation to give users more confidence. These simple protocols do have some features to improve reliability, but the design reasons behind these features are seldom well explained. Some of the features are just clones from existing popular protocols. FPGA vendors also do not disclose the statistic models used for BER calculation. We found the BER numbers reported by closed-source vendor tools report tend to be "optimistic" (i.e. use less data bits to estimate) compared to those from common BER calculation models used in industry. In addition, the poor implementation quality of FPGA IP cores is also a big concern.

By designing our own link-layer protocol, we have a simpler design and improved reliability. The basics of frame format are similar to that of a point-to-point SATA protocol with minimum control command sets. We eliminate all unnecessary control overhead used in more complex scenarios such as multi-channel alignment. In addition to protecting each data frame with CRC checksums, we add hardware sequence numbers and a watch-dog timer based retransmit mechanism to ensure data transmissions are ordered and lossless. To support scalable multi-board simulations and easy power-up initializations, the link status detection and initialization scheme is taken from the 10 Gbps XAUI standard. Our own protocol also incorporates some advanced yet simple to implement features such as data scramble found in SATA and PCI Express 3.0 to improve analog performance of the link. Moreover, by implementing our protocol directly over FPGA transceivers without third-party cores, we can further reduce transceiver latencies.

6.4.3 Reliable DRAM Controller

The most difficult component to build reliably is the DRAM controller, which is an essential component to run everything in DIABLO. Intuitively, this is a very popular component required by many applications, aside from computer architecture simulations. Unfortunately, we had a hard time finding an existing FPGA DRAM controller that was reliable enough. At the beginning, we chose the memory controller implementation provided by Xilinx that is widely used, but we saw memory errors and found reliability bugs in their design. Then, we decided to develop our own memory controller focusing on reliability rather than fancy performance optimizations on top of a design from Microsoft Research by a well-regarded

Turing-award winner. Even so, our own design was plagued with memory errors for quite a long time at the beginning. It took years and running many real simulations to eventually improve the reliability. There are a few reasons why constructing a reliable DRAM controller on FPGA is difficult.

1. DDR memory modules run at a much higher clock speed (400-1000 MHz) compared to the FPGA logic (100-200 MHz). What is more, with each successive FPGA generation, the memory clock rates for each DDR architecture generation have increased at a faster pace than FPGA logic fabric performance, posing a great challenge to memory controller design. We need to carefully engineer the analog data path of the controller. It also requires simpler DRAM controller logic to keep up with the high clock rate requirements. Since DRAMs are fast on FPGAs, having a simpler controller logic will have less system performance impact.

2. The DDR memory controller data path is very wide, for instance 128-bit, and usually runs at clock frequencies close to the device limit. Although FPGA CAD tools already introduce non-deterministic slack for conservative timing analysis, the qualities of result we got from tools are not very consistent between runs. The resulting circuit is not completely reliable even if the tool reports zero timing errors. To mitigate this problem, some FPGA memory controllers from Xilinx use special hard-coded manual routing paths specific to a particular device to ensure the optimal routing on timing critical paths, which is at the cost of increased constraints complexity and reduced design portability. On the other hand, we find simple coarse-grained floor-planning and protecting the data path with ECC works pretty well in practice. Even if we do not use ECC DRAM modules on some cheaper development boards, the ECC circuitry does detect and correct occasional errors at runtime, caused by bad paths introduced during place and route.

3. Each FPGA has hundreds to thousands of small block memories that can be configured as asynchronous FIFOs to implement cross-clock-domain interfaces. These asynchronous FIFO primitives have built-in easy-to-use control logic. Usually, a typical FPGA memory controller design runs at a different clock frequency from the user logic, which interfaces with the controller through asynchronous FIFOs. Although FPGA designs run at lower hundreds megahertz, metastability does occur at this simple clock domain boundary and affects functionality, if not handled properly. This is a very common problem, which appears in both an off-the-shelf vendor-provided memory controller and early versions of our design. Metastability bugs are very hard to debug because of their non-deterministic nature, which may not necessarily happen under load but occasionally with specific workload patterns. In addition, current FPGA verification suites do not support such asynchronous corner cases, which renders reproducing the bugs in simulation impossible.

4. The popular DDR DRAM modules have over twenty timing limit requirements. Meeting these timing restrictions is very tricky, since some limits are used to prevent inter-rank conflicts on the data bus, and others are needed to meet the internal requirements

within the RAMs of a rank for the operations within an open bank. Supporting multiple DIMMs with multiple ranks and simultaneous opened banks are really challenging to get functionally correct. The base Microsoft controller on which we developed our own had been reported running reliably on real hardware for months. However, we still found it violates quite a few DRAM timing restrictions while running workloads generated by DIABLO.

6.4.4 Protecting Against Soft Errors

DIABLO is a large multiple-board multi-FPGA design targeting the state-of-the-art SRAM-based FPGAs, which usually employ the latest process technology. At 65nm and beyond, ASICs and ASSPs exhibit significant soft error rates. Today, relative to the traditional failure mechanisms, soft error rates dominate. The *single-event-upset (SEU)* problem is worse with each process node. The SEU has also steadily increased as the voltage drops and dimensions shrink. At 28nm, the stored charge is less than 1 femto-coulomb. Neutron reaction products can deposit up to 150 femto-coulomb, so upsets in the cell can be common if nothing is done in the design to protect the cell from upsetting [91]. Although careful IC design and layout techniques have decreased the soft error rate, each process technology generation offers twice the logic density, making the soft error rate of the whole chip remains the same. In addition, in order to simulate thousands of servers, the FPGA boards we used are loaded with high capacity DRAMs, which are known to have a high soft error rate.

To decide whether or not we need to protect and how to protect our design, we need to perform back-of-the-envelope calculations based on *soft-error-rate (SER)* data. Table 6.1 shows the unit SER data in DIABLO and the system error rate. From the table, clearly DRAM has the highest system SER followed by FPGA block rams. The FPGA configuration RAM is the most reliable component in the system, which is more reliable than the physical device package. The statistic *mean-time-between-failures (MTBF)* numbers tell us in the worst case we will see one DRAM error every 5 seconds and one FPGA Block RAMs error every three months. In DIABLO, both DRAM and FPGA Block RAMs have a utilization rate close to 100%, while the FPGA configuration RAM are usually half utilized. This suggests that DRAMs and FPGA Block RAMs are more likely dominating factors in system reliability. In addition, given the trend of FPGA technology, the SER of FPGA configuration is getting better in every generation, which makes it less of a reliability concern.

Having estimated the system error rate and knowing the potential vulnerability in our system, we really need answer two questions: 1) do we need to protect against errors 2) do we need to correct detected errors. For the first question, the answer is a clearly yes. The MTBF numbers are based on simple statistical assumptions. Neither does it mean the DRAM fail fast nor there is almost no failure in FPGA configuration memory. It gives us idea which components are more susceptible to soft errors. When we design a system, it is impossible that we have prior experiences of running it for a long time. However, it is very important to be able to detect errors and we could decide if we should do something further to correct the errors. To answer the second question, we can use the FPGA package SER

Component	Soft Error Rate Per unit	System Soft Error Rate
DRAM	25,000-75,000 FIT/Mb [113]	$2.6 * 10^{11} - 7.7 * 10^{11}$ FIT
FPGA Block RAM	692 FIT/Mb [79]	$4.2 * 10^{5}$ FIT
FPGA Configuration RAM	165 FIT/Mb [79]	$8.2 * 10^{4}$ FIT
FPGA package	1 FIT/pin [90]	$8.8 * 10^{4}$ FIT

Table 6.1. Soft error rate of individual components and system in DIABLO, assuming a 20 BEE3 boards with 64 GB DRAM each

number as a caliper. If the component is more reliable than a PCB solder joint, it is not worth the effort of error corrections. Detecting errors at the application level and restart all experiments would be sufficient to handle such extremely rare cases.

As a result, we use ECC DRAM memory on all FPGA boards. Based on data path widths, we protect all FPGA block RAMs with either parity or ECC, implementing with FPGA fabric or utilizing the built-in ECC feature on block RAMs. Regarding the configuration ram, we do not feel the need of employing advanced techniques, like *triple-modular-redundancy (TMR).* Instead, we leverage the FRAME_ECC feature on Xilinx FPGA devices, which detect and correct configuration ram errors using the JTAG chain on the device. If we detect an error in the configuration RAM, which is rare, we must power-cycle the device and restart the computation.

6.5 Pitfalls in DIABLO

Software is easier to modify than the FAME hardware

Although FAME-7 style simulators have significant performance and scale advantages that open up the space of possible experiment, the hardware design and verification efforts are not trivial. To minimize the simulator design efforts, conventional wisdoms suggest building simple hardware and pushing the design complexity to software. This design principle applies to building a system from the ground up with both custom software and hardware. For instance, when we port our research OS, on top of the RAMP Gold simulator, it is straightforward to rewrite part of the OS kernel targeting x86 machines initially to support the host cache and MMU architecture on our hardware.

However, the same hardware and software co-design principle does not work with a system that runs a large set of existing software, which has bugs. Even if we can access the source code, modifying existing software to tailor to requirements of the custom-built hardware is not a trivial process. In many cases, it took great effort just to pinpoint a software issue. One good example of this is porting a recent Linux kernel on DIABLO.

At the point we started porting, we grabbed the most recent stable version (i.e. 2.6.39.3)

from the Linux kernel source tree. The 32-bit SPARC port of the kernel is considered to be stable and has not changed for a decade. We still encountered several kernel bugs on proper cache flush support, triggered by running real dynamically linked programs.

We found fixing all software bugs is a suboptimal solution. First, it takes a long time to identify the real root cause of each bug we could only observe from the user application level. In addition to extensively probing the user and kernel source code, we built dedicated debugging hardware and heavily modified our functional simulator. We invested these significant debugging efforts just to help us to understand the end-to-end software execution path from launching the user application to where the execution diverges compared to an equivalent perfectly coherent system. Usually, when we see an application failure, the real problem could occur billions of cycles inside the kernel before we observed the issue from a user application. Second, the same software issue could occur in multiple places in the kernel. As an example, after we fixed the first kernel bug by porting some of the existing fixes in the more active ARM port, we kept seeing similar non-deterministic bugs while running real applications.

Our further analysis shows that a real traditional operating system functions more correctly on architectures with a fully coherent cache. Although the Linux kernel supports multiple processor architectures with different cache and memory architectures, the legacy shared memory model and performance optimizations makes the cache/TLB interface to the virtual memory subsystem very complicated. In order to support a new target architecture, there are more than twenty architecture dependent flush cache/TLB kernel functions, frequently called by many different kernel routines. Our experiences told us that there has never been a simple fix in one of these kernel functions that could solve all coherency issues. Even if we implement all of these functions correctly, we still could not guarantee whether these functions are used correctly by other parts of the kernel. To ensure the correctness, we find it is very common that many kernel developers just lazily flush everything at the cost of system performance to work around mysterious hardware coherency issues.

To summarize, many kernel issues we are facing are likely because none of the existing 32-bit SPARC processors has a non-coherent instruction and data cache. There was either no need for designers to consider supporting a non-coherent cache architecture, or all existing coherent SPARC processors mask potential bugs in the kernel. The ARM port of the kernel appears to be more correct, because there are many more commercial ARM chips with various cache architectures. As a result, we find adding coherent cache support in DIABLO hardware is a much easier approach. Even though this adds more complexity to the hardware, we show in early chapters that we could design a really simple coherent cache architecture on a special platform with faster DRAM that improves both correctness and performance.

It is easy to build a correct cycle-accurate software simulator

One of the commonly held beliefs about SAME style simulators is that they are easier to build and to get the desired timing behavior. Therefore, it requires less verification effort than FAME simulators, and working with a SAME simulator is more productive with less

development effort. Once a SAME model has been functionally verified, there is no need to change the software model.

However, we find this belief is totally false in practice. In order to assist functional verification of the DIABLO hardware, we build cycle-accurate software performance models for every component in our system. It is true that the C/C++ based software models are very easy to get correct functional-wise, but getting correct cycle-accurate timing behaviors for these software models is extremely hard. Note that since we are building FAME hardware we already have the detailed timing specifications, while developing the equivalent cycle-accurate software models. In many cases, we use FAME designs as a reference to debug timing bugs in corresponding SAME models. One big reason behind this is that the designer has to think and reason at the hardware cycle level due to the nature of FAME. Another important factor is that we are more aware of real structural hazards in the target hardware modeling with FAME.

One of the concrete examples is when debugging Linux kernel issues we spent majority of our time on fixing the software functional simulator to model interrupts and I/Os in a cycle-accurate deterministic manner, which on the other hand are easily modeled on FAME. To sum up, we believe both SAME model and FAME simulators are required in real design scenarios, where C/C++ based SAME focuses more on verifying the functional correctness and the RTL-based FAME is good at the cycle-accurate timing aspect.

Higher level FAME simulators (e.g. FAME-7) run at high clock frequencies with little efforts

FAME simulators especially high-level FAMEs written with abstracted RTL run on modern FPGAs, therefore many designers assumes that FAME should enjoy close to 100 MHz circuit performance without significant effort. Since FPGAs always use the state-of-the-art process technology, designers would assume a design on an older FPGA could receive performance improvement automatically by porting it to newer-generation FPGAs. On the contrary, in practice, we never get high clock frequency for free. Real FAME designs support multiple clocks and I/Os. If we do not carefully map FAMEs to FPGA efficiently, the raw clock frequency could drop quickly therefore losing an order of magnitude performance, as well as making the CAD place and route time unacceptably long.

When we implemented DIABLO, we always tried to leave logic and physical placement optimizations to the CAD tools. Though SRAM-based FPGAs enjoy riding the CMOS scaling curve, the performance of FPGA designs are dominated by routing delays of the switching fabric and clocking resource allocations on FPGAs. Besides, on any FPGA development board, we have to consider the physical I/O placement running in different clock domains, many FPGA structural constraints, such as global clock drivers, building reliable high fan-out synchronous reset network. Every time when pushing our designs through the CAD tools, there is a constant need to re-pipeline the design and other forms of physical optimizations to alleviate routing pressure, such as manually control fan-out of a net, resource

allocation and sharing. Adding fabric dependent synthesis attribute to the source code is also an essential part of the whole design. Besides, we have to reiterate several steps in the design and verification stage once pushing the design through CAD tools.

Since routing is really a critical resource on FPGAs, due to propriety switch box designs on FPGA devices, FPGA vendors give zero visibility to designers about routing hot spot in their designs. Designers must rely on indirect indications such as signal fan-out, estimated net delays output from the tools, and their own empirical design experiences to fine tune the designs. In other words, synthesizing FAME RTLs has never been a simple push-button task without knowledge of underlying FPGA fabric. This process has many iterations with non-trivial design efforts behind. Sometimes this process is even more complicated than designing directly for ASIC under a less constrained environment.

It is sufficient to verify FAME using Verilog simulator running randomized testbench and formal verification suite

FAME has a superior runtime performance over SAME. However, to debug the functionality of FAME the first step is to run testbenches on Verilog simulators, which are orders of magnitude slower than SAME. To speed up the RTL debugging process while maintaining sufficient visibility, in industry people use FAME-0-style hardware emulators, such as Cadence Palladium, and Mentor Graphics Veloce in place of software Verilog simulators. These emulators run at sub-10 MHz but fast enough to run real software, although taking hours to days to compile the RTL design. The ownership cost of these emulators, usually in the rage of millions of dollars, is substantial for academia users. Because of these practical constraints, designers in academia break large RTL designs into modules and stress test them with well-designed unit tests running on clusters of Verilog simulators.

This verification approach sounds reasonable in theory, but by no means has it guaranteed a design would run on hardware without any issues merely running cycle-accurate simulations in Verilog simulator. We argue that any form of functional verifications is not sufficient, unless the design runs on real hardware with real software. There are a few reasons.

First, unit tests heavily rely on the quality of testbenches to provide better coverage. When a testbench is developed, the designers always make usage assumptions for the unit being tested. The more varied usage assumptions the testbench developer made, the better the coverage of the testbench would have. Even a well-designed testbench could not provide 100% coverage. For instance, when we verify our processor design we used a verification suite donated by SPARC international. It is intended for the SPARC certification test, including various unit tests to verify any implementation of the ISA. However, these tests make some assumptions on the cache architecture and pipeline implementations. The verification suite is extremely helpful at the early design stage, but we kept adding more unit tests while finding corner cases with compiler generated code running with our research OS on FPGA hardware. Among the newly added tests, some are targeting at uncovering FPGA logic synthesis bugs.

Second, stressing independent modules could not uncover bugs introduced during module compositions. Different modules place different requirements on verification. The verification approach used for complex control flow logic may be different from that used for math units. Some blocks simply route data without changing it. Other blocks have visible effects that are not predictable without knowing the precise timings of interactions of transactions within them. In extreme cases with multiple clock domains, even a cycle-accurate model cannot predict the expected results [131].

Third, software developers do not follow hardware specifications but try whatever works on existing hardware implementations they have accesses to. For example, the Linux kernel uses a few hardware reserved bits to store some important kernel states. The verification suite we got from SPARC international does not cover this misuse. Such issues often lead to unpredictable nondeterministic behavior in the software. Though having the same reserved bits issue, our C-based functional simulator does not help to discover the real cause because of different interleaving and infinite fast simulated I/O. This problem would not exist if we develop our own kernel/libc software from scratch.

In conclusion, verifying DIABLO is a very challenging topic by itself, which requires extensive hands-on debugging directly on the FPGA prototype. Moreover, verifying complex system is not only an open topic for FPGAs, but also for ASIC designers, like comments from verification gurus "No matter what you do, the coverage is always zero" [132]. In order to improve the confidence level of verifications, designers should develop formal methods to prove the correctness of a design unit as much as possible in addition to running verification suites consisting of random and individual test cases. However, it is a time consuming process to develop formal methods at the full system-level. Designers of complex FPGA designs always have to face the trade-off between the longer verification time and a more reliable design.

6.6 Building a Low-Cost FPGA Board for DIABLO

The DIABLO prototype targeted at the BEE3 multi-FPGA board, which is the third generation FPGA emulation platform developed at Microsoft for computer system emulations. The build cost of BEE3 board is not cheap even for academic users, which is around $15K without FPGAs and DRAMs. A fully populated BEE3 board could cost university users up to $25K. During the development, we use cheaper single-FPGA alternative, the XUPv5 board, which costs only $750 but at the cost of fewer DRAM capacity and inferior build quality (we actually returned 40% XUP boards received because they are defective out-of-box). Although both boards are designed for academic research, they are far from ideal for DIABLO. We find there are mainly two reasons that account for the board cost and usability issues.

1. *High design complexity.* The BEE3 board is an 18-layer PCB design, while the XUPv5 board is a 13-layer design. One big reason behind this complexity is that they both

utilize many FPGA I/O pins for exotic I/Os that are never used by DIABLO. The BEE3 boards use most of the FPGAs I/O pins to provide a high-bandwidth ring interconnect between FPGAs on the same board, while the cheaper XUPv5 board loads the FPGA I/O pins with I/Os like LCD, audio codec and extension bus mostly useful in elementary digital design classes in schools.

2. *Poor I/O assignment*. When designing a PCB board, designers have less knowledge about future gateware that will run on the FPGA. Due to the complexity of supporting many different I/Os, it is impossible to have an optimal I/O assignments minimizing routing delays without prior knowledge about detailed gateware implementations. For instance, the I/O pins of the Gigabit Ethernet PHY are assigned to the opposite side of the die from where the hardened Ethernet MAC controller is located. In addition, the I/O bank and clock input assignment do not take advantage of regional clocking networks on FPGA to alleviate routing pressure on global clocking networks.

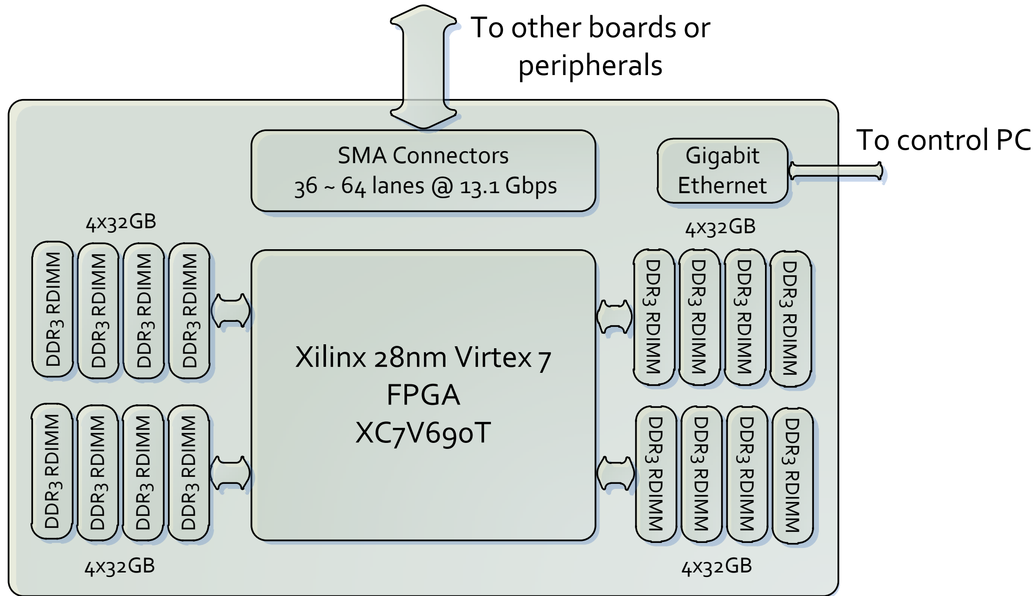


Figure 6.1. Prospective FPGA board optimized for DIABLO.

To address the above issues, we present a more affordable single-FPGA board design, showing in Figure 6.1, with an estimated cost around $5,000 including the FPGA. Hypothetically, the board is targeting the 28nm Xilinx Virtex 7 FPGA. Knowing the DIABLO design requirements, we equip the board with the following features:

1. *Single FPGA design*. For economical reason, we choose a mid-size Xilinx FPGA (XC7V690T), which has a cost of $600∼$800. We use the slowest speed grade FPGAs, because higher speed grade FPGAs only offer 10%∼15% performance gain but

at a substantial higher cost. DIABLO is a modularized single FPGA design. Compared to the FPGA on BEE3, the logic capacity of the FPGA in our new design is $> 4x$, while the BRAM capacity is $> 7x$.

2. *Minimum I/Os.* The only low speed I/Os is a Gigabit Ethernet interface, which is used as the frontend link in DIABLO to handle all control and console traffic.

3. *Maximum memory capacity.* Since one of the key limiting factors in emulation is the DRAM capacity, we use almost all FPGA I/Os to support more DRAM DIMMs. In our design, we support four independent channels with each having up to four DIMMs. If fully populated with DDR3 registered ECC DIMMs, every board can support up to 512 GB DRAM, which is 8x of that on the BEE3 board. Registered DIMMs are preferred because of higher capacity and better signal integrality. In addition, since the DRAM controller is almost the only gateware need to take care of during the I/O assignment and they are well understood, we can easily find the optimal I/O placement when designing the PCB.

4. *Simple inter-board and peripheral expansion.* We scale up DIABLO using only high-speed transceivers. Depends on the packaging of the FPGA, each board can have 36-64 13.1 Gbps transceivers, which translate to an aggregated off-board bandwidth of 471.6-838.4 Gbps. We do not bundle these transceivers into groups to fit into fancy connectors that require more expensive cables. Instead, we use the basic SMA connector to provide the maximum flexibility and reduce the cable cost. We could also build daughter cards connecting through the same high-speed serial transceivers that provide other I/O connectivity, such as storage.

Chapter 7

Future Work

In previous chapters, we demonstrated a prototype of DIABLO running unmodified production server software. The working set of all applications fits in the on-board DDR2 memory. The applications in our experiments exercise the networking stacks with little disk I/O except for loading dynamically linked libraries. In this chapter, we discuss several limitations of the current prototype. We also propose solutions to address them for future work. We classify future work into two categories: painkillers (must-have) and vitamins (nice-to-have).

7.1 Painkillers: Must-Have Features

7.1.1 Improving emulated target memory capacity

FAME-7 modeling techniques give us the capability of emulating larger target systems with fewer virtualized FPGA resources at the cost of longer simulation time. However, the target memory capacity is very hard to virtualize with limited physical DRAM storage per FPGA board. We should note that the memory capacity issue has not been an issue only for DIABLO, but also an open topic for any work that is trying to emulate a large datacenter with limited hardware resources.

Although a fully populated BEE3 supports up to 64 GB DRAM, each simulated node only supports 128 MB memory given the high-density design. According to our conversation with industry researchers [71], 128 MB memory is adequate to study novel datacenter network transport protocols, and server applications (e.g. memecached), whose performance is independent of total memory capacity. However, there are quite a few disk I/O intensive applications in the datacenter, such as those based on distributed file systems like Google

GFS [69] and Microsoft Dryad [80], which require gigabyte DRAM buffers to improve performance [138]. Without real physical storage, it is hard to model the dynamics of these applications. A newer board design proposed in Chapter 6 could improve the current memory capacity per node by factors of two to four. Besides, having real full-size DRAM main memories for thousands of nodes will still cost a significant amount of money. Given that future datacenter applications are more memory intensive, we still need new technology to support several gigabytes of memory per node.

There are two straightforward workarounds to support more simulated memory. One is to reduce the number of simulated nodes per FPGA. Nevertheless, it is still very hard to scale to several gigabytes of memory per simulated node without significantly affecting the simulation density. Thus, it is less practical in the context of massive-scale architectural simulations. Another approach is to apply software page sharing technologies used by virtual machines to avoid storing redundant copies in multiple nodes to improve memory utilizations. Although we know it is easy to share memory pages for applications and OS code, page-sharing is less effective for exascale applications with huge in-memory datasets with little redundancy. To sum up, we need to model large real physical target memories with a cheaper host memory technology.

Note that the overall simulation performance is three orders of magnitude slower than a real datacenter. This opens the opportunity of using FLASH memory to simulate server memory DIMMs. According to the ITRS memory technology roadmap in 2010 [19], FLASH memory costs a tenth of the same capacity DRAM but with a 1000× greater random access latency. Moreover, newer memory technologies like Resistive Random Access Memory (RRAM) and 3D-stacking may further improve cost-per-bit and density. Therefore, we propose to build a DIABLO with a hybrid memory hierarchy including both DRAM and FLASH memories to address the current memory capacity limitation. The basic idea is to use FLASH to emulate the target server DRAM, and to use the DRAM DIMMs on our FPGA boards as a "memory page cache". In theory, the memory hierarchy works similarly to OS virtual memory with demand paging and swap support. The DRAM page cache can be as large as 64 to 128 MB per node, which is large enough to cache the working set of many applications.

In terms of the physical implementation of the FLASH storage, there are two possible ways to connect them to the existing system. One is to connect FLASH memory through the high-speed transceivers on the FPGA board. In this way, we could use FLASH drives with the standard SATA disk interface. The benefit of using a standard disk interface is to easily employ commodity large Solid State Drives (SSDs), but the drawback is a more complicated interface with higher access latencies. Another approach is to directly plug in FLASH-chip-populated DIMMs into memory DIMMs on our FPGA board. Microsoft Research has already built SLC NAND flash DIMMs for BEE3 supporting up to 32 GB per DIMM with a simple low-latency interface [60]. By swapping the SLC chips with MLC chips, the overall capacity can be easily improved by factors of two to four. Micron is also developing a DDR4-compatible hybrid DRAM-NAND that could contain more than 256 GB of memory with a bandwidth over 300 GB/s shipping in 2016 [33].

When designing a FLASH-based memory system, we face two major challenges: bandwidth and latency. The NAND flash memory reads faster than it writes. The random read

latency around 40 to 50μs, while the write latency is around 100μs to 300μs [61]. The new 10nm Samsung MLC FLASH chip claims to have a bandwidth of 260 MB/s read and 50 MB/s write [30]. The 1000x slower access latency of FLASH could be hidden by DIABLO's simulation slow down. In addition, the host-multithreading feature of FAME-7 models can help to hide the access latencies of FLASH when there is a miss in the DRAM page cache. On the other hand, the bandwidth heavily depends on the FLASH controller and device architecture. The current BEE3 flash DIMM provides a 160 MB/s write bandwidth and 320 MB/s read bandwidth through SLC FLASH with a simple controller. If we use off-the-shelf SSD with a standard disk interface, for example, the 25nm Micron C400 SSD [12], we could get a comparable bandwidth of 180 to 500 MB/s read and 200 to 260 MB/s write with around 55μs access latency. In order to build a balanced simulation system with a 1000$\times$ slow down factor and a 20 GB/s peak target aggregated memory bandwidth per server in target, we need to connect at least one SSD or FLASH DIMM to every 10 simulated servers.

Although modern FPGAs have plenty of multi-gigabit transceivers to offer sufficient bandwidths for the FLASH interface, considering the simulation density per FPGA (hundreds of servers) in the future, we need to design a custom FLASH storage system to improve the bandwidth of existing cheap FLASH storage by 10x. This is plausible using multiple 64Gb 10nm Samsung modules as mentioned earlier in parallel together with a low-cost FPGA for the high-throughput interface mounted on a compact PCB board. To simplify the controller design and to optimize for DIABLO's usage scenarios, we do not need to support any industry protocol. Hence, the overall design efforts and manufacturing cost for this new storage system should be manageable.

7.1.2 Adding multicore and 64-bit CPU Support

Currently, we use only one hardware thread in each server model to model a single-CPU server. Applications will be increasingly optimized for multicore. Having a multicore server model will improve the accuracy of our server model.

In the future, we could easily extend the design to support more simulated cores per CPU by using more logic resources on newer-generation FPGAs. However, most of the work of adding a multi-core support will be in extending the existing Linux kernel. The current 32-bit SPARC port of the Linux kernel supports only up to 4-way SMP with many hardcoded tables, such as page tables and interrupt tables. This restriction is because there is no commercial 32-bit SPARC SMP system with more than four cores. This limitation with Linux is only due to our use of 32-bit SPARC ISA. The overall approach would work with other ISAs that had better multicore support.

Although the existing single-core 32-bit SPARCv8 configuration runs many off-the-shelf applications, there are several limitations of in our implementation, shown as the following:

1. *No support for 64-bit virtual address space*: This is an intrinsic limitation from a 32-bit ISA. 64-bit virtual address space would be useful, for supporting more than 4

GB virtual address spaces. It is also crucial to work with the hybrid FLASH/DRAM memory hierarchy proposed in the previous section.

2. *Non-IEEE 754 compliant FPU*: DIABLO focuses on network I/O-centric applications, which do not execute many floating point instructions. A full-fledged IEEE-754 compliant FPU consumes a large amount of FPGA resources even with the help of hard DSP blocks. Therefore, we do not support all rounding modes and precisions in our current FPU implementation.

3. *Low floating-point performance*: We only implement simple FPU operations, and use the Newton-Raphson method to emulate complex FPU operations like division and square root in software. This software implementation is similar to the hardware floating-point implementation in AMD K-7 processor [104] but with reduced precisions. Another performance limitation of SPARC v8 is that the ISA does not support transferring data between the integer and the floating-point register file directly.

In conclusion, many of these limitations are not caused by our FPGA modeling methodology, but due to trade-offs between FPGA resource consumption and engineering effort. Moreover, some come from limitations of the ISA itself.

7.2 Vitamins: Nice-To-Have Features

7.2.1 More flexible micro-code based NIC/Switch Models

In the existing DIABLO prototype, we directly build FAME-7 models for the NIC and switch hardware. We implement all important architecture features using pure FPGA hardware. These I/O models perform well on FPGAs. Although they consume considerable FPGA resources, they are not on the performance critical path. Because of sufficient parallelism in the hardware, in the worst case the host FPGA cycles consumed by DIABLO I/O models account for less than a third of those used for simulating server computations. In addition, the utilization of datacenter networking gear rarely reaches 100-percent utilization in practice. This suggests that our FAME-7 I/O models are over-optimized.

In order to simplify our model abstractions, we analyzed real-life usage of the simulated target hardware and concentrate on commonly used features. We further take advantage of the decoupled feature in FAME-7 models to build simpler FPGA hardware and use multiple host FPGA cycles to simulate high-performance hardware. Even with these abstractions, to achieve basic functionality we still have to implement many exotic target features in hardware, such as descriptor-based gather/scatter NIC DMA and switch virtual output queues. Moreover, to cope with the FPGA host memory subsystem, we have to build a distinct host-cache for every DIABLO model. Combined with host multithreading and decoupled design, we end up introducing more intermediate states that make the DIABLO FAME-7 model even harder to design and debug than the original target hardware. As shown

in Table 4.2, the FPGA resource consumption of a DIABLO NIC model is comparable to that of the server model. The downside of the extra complexity is that it is hard to add features to existing models using Verilog. For instance, in our prototype we omit some interesting TCP/IP checksum and send/receive offloading features, which could take many months to implement in hardware.

Therefore, we think in the future it is not necessary or desirable to implement every target hardware functional feature in FPGA hardware. Recent trends in datacenter switching also show that the static ASIC forwarding engine of a datacenter switch will likely become more programmable like a microcode engine [46]. Analyzing the design complexity of our existing model, we found that the majority of the complexity of our networking I/O models arises from moving memory data stored in some hardware-specific data structure rather than a heavy computation requirement. In addition, some advanced hardware features like TCP/IP segmentation offloading are also easier to model with a piece of software running on an I/O processor or microcode engine.

Based on these observations, we propose to build programmable microcode engines plus FPGA hardware accelerators to replace the current DIABLO I/O model. FPGA hardware accelerators are still necessary to model some heavily parallel hardware structures in the target system, such as the switch virtual queue scheduler. We can reuse the integer pipeline of the DIABLO server model as the microcode engines for everything else. There are many advantages of this approach:

1. It is easy to add and modify modeled architecture features by changing the code running on the microcode engine.
2. The SPARC v8 multithread microcode engines would be very easy to program with the existing software toolchain and compiler support.
3. The FPGA host memory system design can be greatly simplified. We no longer need different host caches for different DIABLO models, as all models will be more homogeneous.
4. The overall design and verification effort of DIABLO will be reduced because fewer distinct hardware model needs to be built.
5. The multithreaded microcode engine works like a programmable multicore network processor, such as the Intel IXP network processors. Running at around 100 MHz, it could provide enough horsepower to handle many network processing tasks even at the target line speed.
6. The microcode engine consumes very few resources on the FPGAs and we can have lots of these engines to improve the simulation performance when necessary.

7.2.2 Applying DIABLO to other I/O research

Currently, DIABLO is targeting datacenter interconnect simulations. We only build detailed architecture models for datacenter networking components. To make the whole system properly functional, we also implement functional models for other types of datacenter I/O, such as local disks and command consoles. However, these functional models are not optimized for emulation performance.

For instance, we forward all un-modeled I/O requests to a front-end PC from the FPGA through a narrow Gigabit Ethernet link. The front-end link works in a polling mode with the control appserver running on the PC initiating all transactions. Therefore, the raw performance of these functional I/O models suffer from the low bandwidth and long latency caused by the control software. Consequently, the console log dumping and functional disk performance have become the performance bottleneck during the Linux boot on DIABLO.

There is an incentive to improve the current functional I/O performance just to shorten the Linux boot time before running experiments. We could address this performance issue from both software and hardware perspective. A simple solution is to implement buffered I/O for both character and block kernel device drivers to improve the polling efficiencies. A more sophisticated approach is to build a dedicated DMA hardware for these I/Os in the FPGA instead of using software polling. In order to overcome the frontend link bandwidth limitation and reduce communication latencies, we can employ the spare multi-gigabit transceivers on FPGAs to directly connect the DIABLO hardware to the control PC or physical I/O devices, such as disk.

Moreover, datacenter disk I/O has been an active research topic for both software and hardware researchers in recent years, especially with the advent of new I/O storage technologies like SSD. It is plausible to build storage timing models on FPGAs with improved functional I/O hardware to extend DIABLO's emulation capability. As mentioned earlier many datacenter storage subsystems involve aggressive in-memory caching with complex software-managed control logic, emulating more target memory is almost a prerequisite for any storage research. Other than the memory capacity requirement, we found no technical difficulty that prevents the current DIABLO platform from applying to other I/O-related research.

7.2.3 Supporting more real world applications

Currently, we cross-build all C/C++ applications that run on DIABLO using a regular Linux/x86 machine. Unfortunately, many applications are not written with cross-compilation in mind. Hence, we always need to invest time to port config and make scripts. This work has become the main barrier to running more programs. One future direction for running more software is to bring up a full user-land Linux distribution, such as Debian Linux. If so, each compute node in DIABLO could be used as an independent Linux workstation, which runs standard GNU development tool chain natively on DIABLO or in our C function simulator. This improvements eliminates troubles of cross-compiling, therefore

we can easily support more sophisticated programs that require complex software package dependencies.

Expanding DIABLO's support of managed languages and scripting languages, such as Java/OpenJDK and Python, is also a good future direction. Although many proprietary datacenter storage and computation software frameworks are written with native languages like C/C++ for performance reasons, there are still quite a few popular open-source frameworks like Hadoop written in managed languages. With the help of full Debian Linux distribution, it is straightforward to support any popular managed language found in the most up-to-date Debian repository.

To sum up, we conclude some limitations of DIABLO and propose future improvements. Among all aforementioned limitations, the memory capacity issue is the dominating factor that prevents DIABLO from being applied to simulate a wider range of datacenter applications, particularly those of disk I/Os. However, the memory capacity has become a general open problem for any work that attempts to emulate large-scale datacenters with limited hardware resources. We also point out several future directions to build more flexible DIABLO models that significantly reduce design and verification efforts. Besides simulating datacenter networking architecture, we believe that DIABLO is promising for other datacenter I/O research.

Chapter 8

Conclusions

Simulation is the cornerstone of computer architecture research and development. Traditional software simulation techniques allow architects to explore a design space and validate their proposed implementations without building expensive hardware prototypes. On the other hand, the complexity of target systems has grown exponentially. Particularly in the context of ware-house scale computing, the scale has made any performance evaluation a challenging task. Moreover, data-center applications running at this enormous scale interact with the hardware system as well as the operating system in a tightly-coupled manner. Many application performance issues are not only merely software problems, but are consequences of complex interplays between hardware and software. From the computer system research prospective, we point out that researchers are facing an evaluation crisis. This crisis is not merely how fast the raw evaluation performance is but also the capability and availability of analyzing the whole target design at a more believable scale.

In order to overcome current evaluation limitations, people have proposed many improvements to the traditional event-based simulation and statistical analytical models. Many researchers are in favor of such cheap software-based evaluation platforms to avoid the need of building any hardware. Usually, such models only targeting manually-picked "point of interests" without looking at the whole hardware and software system. Though easy to build, in this dissertation we show that such methodology is seriously flawed by limiting the potential design space due to subjective intuitions of designers.

Unlike other modeling work, our philosophy of datacenter design evaluations is to treat datacenter hardware and software as a white-box system with plenty of architectural details. In this dissertation, we start off by analyzing general computer architecture evaluation methodologies. We introduced a novel performance modeling approach using FPGAs. To help further understand many existing efforts aiming to boost simulation performance, we developed the *FPGA Architecture Model Execution (FAME)* and *Software Architecture Model Execute* terminology. The four-level taxonomy of FAME levels help to systematically explain the cost-efficiency of DIABLO simulations.

The key insight of FAME is that instead of prototyping target architecture we implement multithreaded abstract performance models on FPGAs. Our models also contain many runtime configurable options that can be changed without reprogramming the FPGAs. This approach enables faster design space exploration without going through FPGA place and route for hours. FAME models are also capable of running full software stack at two orders of magnitude faster than state-of-the-art software alternatives.

As an illustration of the efficiency of high-level FAME, we built RAMP Gold a FAME-7 full-system multicore simulator on a $750 single FPGA board. RAMP Gold employs a decoupled functional and timing architecture. It supports standard 32-bit SPARC v8 ISA that runs the Linux operating system and unmodified datacenter software stack. In this dissertation, we also describe the detailed FPGA host implementation techniques building efficient high-level FAME simulators. Due to characteristics of the host FPGA platform, we found that the architecture simulator itself behaves very differently from the target machine it models. The design of tiny host functional cache and single line buffer based coherent architecture shows that the FAME architecture itself is a new design space. Furthermore, we discuss two ways of scaling the simulator itself:*strong scaling* using more parallel hardware and *weak scaling* using same hardware but pack more host threads.

Inspired by the success of RAMP Gold on multicore simulation, we applied the FAME-7 idea to model datacenter networking gears such as switches and network interface cards. We built DIABLO, a low-cost FPGA emulator for datacenter. As a proof of concept, we implement DIABLO in 24 Xilinx Virtex 5 FPGAs on six BEE3 boards. The prototype simulates up to near 3,000 nodes in 96 server racks with an Ethernet interconnect. Our prototype occupies only half of a standard server rack and consumes around 1.2 kwatt in total. The testbed was equipped with 384 GB DRAMs in 48 independent DRAM channels that has a peak memory bandwidth of 179 GB/s. All FPGAs in DIABLO are connected using high-speed point-to-point serial links at 2.5 Gbps, simulating 8.6 billion target instructions per second. Inheriting a good software support from RAMP Gold, DIABLO runs standard Linux with unmodified datacenter software stack. The overall hardware cost of our prototype is around $100K, and it costs little to maintain. We discussed the detailed architecture and implementation of DIABLO, and provided some future directions for building efficient FAME emulator and hardware platform.

As a proof of concept and validation, we conducted three experiments on real life datacenter hardware and software research problems. In the first example, we use DIABLO to faithfully model a novel datacenter circuit-switching architecture from Microsoft Research, running traffic patterns sampled from the Dryad Terasort application. We run the workload on simulated servers along with real device drivers. The results from DIABLO provided insight for improving future designs. In the second case, we reproduce the well-known datacenter TCP Incast throughput collapse problem on DIABLO. Moreover, we revisit this classic networking problem from the system prospective showing the impact of simulating computation and a full OS, including the device drivers. We also illustrate various performance scalability issues of TCP incast at a higher link speed. In the final example, we scale DIABLO to a 2,000-node system running unmodified *memcached* servers with clients generating traffic based on Facebook's production data. This example demonstrates that we can

apply DIABLO to study hardware and software interactions at scales that were previously difficult to achieve without significant hardware investment.

One of the key observations of DIABLO is the importance of modeling computation with both hardware and software architecture in detail. We show with the TCP Incast scalability test that the system performance bottleneck could shift to many places in the system. It is not limited to the network transport protocol and switch buffer designs. To better understand application scalability issues at warehouse-computing scale, people need to explore the design space by tweaking many knobs in both hardware and software designs. Historically, such design space sweeps are typical tasks of analytical models and event-based simulations, which we argue could not cope with fast datacenter software churns and increasing target complexities. We believe the FAME-based DIABLO simulator have the performance and flexibility to conduct such design space exploration at scale.

Our own usage experiences of DIABLO as a research platform are also positive. Not only can it produce results for massive number of simulated nodes that no practical alternative approach could achieve, but it also provides an enormous simulation bandwidth to speed up simulation at rack-level scales. Although the single node performance is three-orders of magnitude slower than real machines, the ability of simulating a large number of nodes at the same time overcomes the raw simulator performance deficit.

The design goal of DIABLO is neither to achieve the closest absolute numbers when model existing hardware nor to build a machine that runs fast enough to beat the target design. The motivation is to reproduce the relative system behavior at reasonable speed with affordable resources. We believe it is promising for datacenter-level experiments, helping to evaluate many novel hardware and software proposals at scale.

Bibliography

[1] Dinero iv cache simulator, online at:http://www.cs.wisc.edu/˜markhill/dineroiv.

[2] Effects of congestion on TCP transactions: comparison of Cisco Nexus 5010 and Arista 7124S network switches under incast conditions, hhttp://principledtechnologies.com/clients/reports/Cisco/24-port_switch_incast_comparison.pdf.

[3] Facebook Memcached, https://github.com/amanuel/facebook-memcached.

[4] Incisive Enterprise Palladium Series with Incisive XE Software, http://www.cadence.com/rl/Resources/datasheets/incisive_enterprise_palladium.pdf.

[5] libmemcached, http://libmemcached.org/libMemcached.html.

[6] R2D2: RAPID AND RELIABLE DATA DELIVERY IN DATA CENTERS, http://www.stanford.edu/ atikoglu/r2d2/.

[7] Twemcache, hhttps://twitter.com/twemcache.

[8] Yahoo! Reaches for the Stars with M45 Supercomputing Project, http://research.yahoo.com/node/1884, 2007.

[9] Scaling memcached at Facebook, https://www.facebook.com/note.php?note_id=39391378919, 2008.

[10] Glen Anderson, private communications, 2009.

[11] Leon3 Processor, http://www.gaisler.com, 2009.

[12] Micron C400 mSATA SSD, http://www.micron.com/products/solid-state-storage/client-ssd/c400-msata-ssd, 2009.

[13] Sun Datacenter InfiniBand Switch 648, http://www.sun.com/products/networking/infiniband.jsp, 2009.

[14] Switching Architectures for Cloud Network Designs, http://www.aristanetworks.com/en/SwitchingArchitecture_wp.pdf, 2009.

[15] The Convey HC-1: The Worlds First Hybrid-Core Computer, http://www.conveycomputers.com/, 2009.

[16] Cisco Nexus 5000 Series Architecture: The Building Blocks of the Unified Fabric . . http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-462176.html, 2010.

[17] Force10 S60 High-Performance 1/10 GbE Access Switch, http://www.force10networks.com/products/s60.asp, 2010.

[18] Hadoop, http://hadoop.apache.org/, 2010.

[19] ITRS Memory Technology Selection Process, http://www.itrs.net/Links/2010ITRS/2010Update/ToP 2010.

[20] Xilinx Virtex 7 Series FPGAs, http://www.xilinx.com/technology/roadmap/7-series-fpgas.htm, 2010.

[21] Enhancing the Scalability of Memcached, online at http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached/, 2012.

[22] Google G-Scale Network, online at http://www.eetimes.com/electronics-news/4371179/Google-describes-its-OpenFlow-network, 2012.

[23] How Facebook builts its Timeline feature, http://gigaom.com/cloud/how-facebook-built-its-timeline-feature, 2012.

[24] Intel Data Direct I/O technology, http://www.intel.com/content/www/us/en/io/direct-data-i-o.html, 2012.

[25] memcached: a distributed memory objec caching system, online at http://memcached.org, 2012.

[26] Microsoft Hyper-V virtualization platform, http://www.microsoft.com/en-us/server-cloud/windows-server/server-virtualization.aspx, 2012.

[27] Open Compute Project, online at http://opencompute.org, 2012.

[28] Oracle Virtualbox VM, http://www.virtualbox.org/, 2012.

[29] QEMU open source processor emulator, http://wiki.qemu.org, 2012.

[30] Samsung's new 10nm-process 64GB mobile flash memory chips are smaller, faster, better, http://www.engadget.com/2012/11/15/samsung-10nm-64gb-emmc-mobile-flash-memory, 2012.

[31] Virtual Distributed Ethernet, http://vde.sourceforge.net, 2012.

[32] VMware Virtual Server, http://www.vmware.com, 2012.

[33] Flash will ride DRAM bus in 2014, http://www.eetimes.com/electronics-news/4406570/Flash-will-ride-DRAM-bus-in-2014–says-Micron, 2013.

[34] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM '08*, pages 63–74, New York, NY, USA, 2008. ACM.

[35] A. R. Alameldeen and D. A. Wood. Addressing Workload Variability in Architectural Simulations. *IEEE Micro*, 23(6):94–98, 2003.

[36] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010.

[37] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of the 4th International Conference on Supercomputing*, pages 1–6, New York, NY, USA, 1990. ACM.

[38] K. Asanović et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, UC Berkeley, Dec 2006.

[39] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[40] T. Austin et al. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[41] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[42] K. Barr. *Summarizing Multiprocessor Program Execution with Versatile, Microarchitecture-Independent Snapshots*. PhD thesis, MIT, Sept 2006.

[43] L. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23:22–28, 2003.

[44] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages –, New York, NY, USA, 2011. ACM.

[45] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[46] A. Bechtolsheim. Moore's Law and Networking. In *The Linley Group Processor Conference*, San Jose, CA, USA, 2012.

[47] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT '08*, pages 72–81, New York, NY, USA, 2008. ACM.

[48] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[49] Y. Chen, R. Griffit, D. Zats, and R. H. Katz. Understanding tcp incast and its implications for big data workloads. Technical Report UCB/EECS-2012-40, EECS Department, University of California, Berkeley, Apr 2012.

[50] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN '09: Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82, New York, NY, USA, 2009. ACM.

[51] D. Chiou, H. Angepat, N. P. Patil, and D. Sunwoo. Accurate Functional-First Multicore Simulators. *Computer Architecture Letters*, 8(2), July 2009.

[52] D. Chiou et al. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *MICRO '07*, pages 249–261, Washington, DC, USA, 2007.

[53] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, pages 98–109, 2011.

[54] E. S. Chung et al. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):1–32, 2009.

[55] U. Cummings, D. Daly, R. Collins, V. Agarwal, F. Petrini, M. Perrone, and D. Pasetto. Fulcrum's FocalPoint FM4000: A Scalable, Low-Latency 10GigE Switch for High-Performance Data Centers. In *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects*, pages 42–51, Washington, DC, USA, 2009. IEEE Computer Society.

[56] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *SIGCOMM*, pages 254–265, 2011.

[57] N. Dave et al. Implementing a Functional/Timing Partitioned Microprocessor Simulator with an FPGA, Feb. 2006.

[58] N. Dave et al. Implementing a functional/timing partitioned microprocessor simulator with an FPGA. In *Proc. of the Workshop on Architecture Research using FPGA Platforms*, held at HPCA-12, Feb. 2006.

[59] J. Davis, C. Thacker, and C. Chang. BEE3: Revitalizing Computer Architecture Research. Technical Report MSR-TR-2009-45, Microsoft Research, Apr 2009.

[60] J. Davis and L. Zhang. FRP: a Nonvolatile Memory Research Platform Targeting NAND Flash. In *The First Workshop on Integrating Solid-state Memory into the Storage Hierarchy, Held in Conjunction with ASPLOS 2009*, Washington, DC, 2009. ACM.

[61] P. Desnoyers. Empirical Evaluation of NAND Flash Memory Performance sigops. In *Workshop on Hot Topics in Storage and File Systems (HotStorage09)*, Big Sky, MT, 2009. ACM.

[62] L. L. Dick, J.-T. Li, T. B. Huang, and S. K. C. Kenneth. US Patent 5425036 - Method and apparatus for debugging reconfigurable emulation systems, http://www.patentstorm.us/patents/5425036.html.

[63] M. Dobrescu, N. Egi, K. J. Argyraki, B.-G. Chun, K. R. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *SOSP*, pages 15–28, 2009.

[64] J. Emer et al. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[65] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: measured power, performance, and scaling. In *ASPLOS*, pages 319–332, 2011.

[66] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM '10*, pages 339–350, 2010.

[67] A. Ganesan, D. Lee, A. Leinwand, A. Shaikh, and M. Shaw. What is the impact of cloud computing on the data center interconnect? In *Hot Interconnects*, 2011.

[68] K. Garlapati and A. Sirasao. private communication, October 2010.

[69] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. SOSP '03, pages 29–43. ACM, 2003.

[70] G. Gibeling, A. Schultz, and K. Asanović. RAMP architecture and description language. In *2nd Workshop on Architecture Research using FPGA Platforms*, February 2006.

[71] A. Greenberg. private communication, March 2010.

[72] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2009.

[73] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM '09*, pages 51–62, New York, NY, USA, 2009. ACM.

[74] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM '09*, pages 63–74, New York, NY, USA, 2009. ACM.

[75] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM '08*, pages 75–86, New York, NY, USA, 2008. ACM.

[76] S. Han, K. Jang, K. Park, and S. B. Moon. Packetshader: a gpu-accelerated software router. In *SIGCOMM*, pages 195–206, 2010.

[77] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[78] L. R. Hsu, A. G. Saidi, N. L. Binkert, and S. K. Reinhardt. Sampling and stability in tcp/ip workloads. In *Proceedings of the First Annual Workshop on Modeling, Benchmarking, and Simulation*, MoBS '05, pages 68–77, 2005.

[79] J. Hussein and G. Swift. Mitigating Single-Event Upsets, WP395 (v1.0), http://www.xilinx.com/support/documentation/white_papers/wp395-Mitigating-SEUs.pdf, 2012.

[80] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[81] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM '08*, pages 51–62, New York, NY, USA, 2008. ACM.

[82] R. Katz. Tech titans building boom: The architecture of internet datacenters. *IEEE Spectrum*, February 2009.

[83] A. J. Kleinosowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(1):7, 2002.

[84] K. Klues et al. Processes and Resource Management in a Scalable Many-core OS. In *HotPar10*, Berkeley, CA, June 2010.

[85] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *Computer*, 40(9):103–105, 2007.

[86] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[87] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz. RAMP Blue: A Message-Passing Manycore System In FPGAs. In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 54–61, Amsterdam, The Netherlands, 2007.

[88] B. Kwan, P. Agarwal, and L. Ashvin. Flexible buffer allocation entities for traffic aggregate containment. US Patent 20090207848, August 2009.

[89] J. W. Lee et al. Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks. In *ISCA '08*, pages 89–100, Washington, DC, USA, 2008.

[90] A. Lesea. private communication, October 2007.

[91] A. Lesea. Soft Error Derating, or Architectural Vulnerability. In *Third Annual IEEE-SCV Soft Error Rate (SER) Workshop*, 2011.

[92] K. Lim, D. Meisner, A. Saidi, P. Ranganathan, and T. F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th annual international symposium on Computer architecture*, ISCA '13, 2013.

[93] R. Liu et al. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *HotPar09*, Berkeley, CA, March 2009.

[94] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35, 2002.

[95] M. M. K. Martin et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[96] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[97] J. E. Miller et al. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA-16*, January 2010.

[98] M. Mitzenmacher, A. Broder, A. Broder, M. Mitzenmacher, and M. Mitzenmacher. Using multiple hash functions to improve ip lookups. In *In Proceedings of IEEE INFOCOM*, pages 1454–1463, 2000.

[99] J. Mudigonda, P. Yalagandula, J. C. Mogul, B. Stiekes, and Y. Pouffary. Netlord: a scalable multi-tenant network architecture for virtualized datacenters. In *SIGCOMM*, pages 62–73, 2011.

[100] S. Mukherjee et al. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 8(4):12–20, 2000.

[101] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable router architecture for experimental research. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 1–7, New York, NY, USA, 2008. ACM.

[102] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM '09*, pages 39–50, New York, NY, USA, 2009. ACM.

[103] N. Njoroge, J. Casper, S. Wee, T. Yuriy, D. Ge, C. Kozyrakis, , and K. Olukotun. ATLAS: A Chip-Multiprocessor with Transactional Memory Support. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE), Nice, France, April 2007*, pages 1–6, 2007.

[104] S. F. Oberman. Floating point division and square root algorithms and implementation in the amd-k7 microprocessor. In *IEEE Symposium on Computer Arithmetic*, pages 106–115, 1999.

[105] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, pages 29–41, 2011.

[106] V. S. Pai et al. RSIM Reference Manual. Version 1.0. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, July 1997.

[107] M. Pellauer, M. Adler, D. Chiou, and J. Emer. Soft Connections: Addressing the Hardware-Design Modularity Problem. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 276–281, New York, NY, USA, 2009. ACM.

[108] M. Pellauer et al. Quick performance models quickly: Closely-coupled partitioned simulation on FPGAs. In *ISPASS*, 2008.

[109] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. *SIGMETRICS Perform. Eval. Rev.*, 21(1):48–60, 1993.

[110] M. Rosenblum et al. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, 1997.

[111] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proceedings of the 5th annual Linux Showcase & Conference - Volume 5*, ALS '01, pages 18–18, Berkeley, CA, USA, 2001. USENIX Association.

[112] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 480–491, Washington, DC, USA, 2006.

[113] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: a large-scale field study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[114] H. Shah. *Solving TCP Incast in Cluster Storage Systems*. Technical report (Information Networking Institute). Carnegie Mellon University. Information Networking Institute, 2009.

[115] J. Shalf. private communication, June 2009.

[116] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[117] G. E. Suh, C. W. O'Donnell, and S. Devadas. Aegis: A Single-Chip Secure Processor. *IEEE Design and Test of Computers*, 24(6):570–580, 2007.

[118] S. Swanson, A. Putnam, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, and S. J. Eggers. Area-Performance Trade-offs in Tiled Dataflow Architectures. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 314–326, Washington, DC, USA, 2006. IEEE Computer Society.

[119] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, D. Patterson, and K. Asanović. Ramp Gold: An FPGA-based architecture simulator for multiprocessors. In *4th Workshop on Architectural Research Prototyping (WARP-2009), at 36th International Symposium on Computer Architecture (ISCA-36)*, 2009.

[120] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A case for FAME: FPGA architecture model execution. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 290–301, New York, NY, USA, 2010. ACM.

[121] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the datacenter. In *HotNets*, 2009.

[122] C. Thacker. Rethinking data centers. October 2007.

[123] C. Thacker. private communication, May 2009.

[124] C. Thacker. A data center network using FPGAs, May 2010.

[125] C. Thacker. Beehive: A many-core computer for FPGAs, August 2010.

[126] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM '09*, pages 303–314, New York, NY, USA, 2009. ACM.

[127] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. P. Ryan. c-through: part-time optics in data centers. In *SIGCOMM*, pages 327–338, 2010.

[128] J. Wawrzynek et al. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2):46–57, 2007.

[129] M. Wehner, L. Oliker, and J. Shalf. Towards Ultra-High Resolution Models of Climate and Weather. *International Journal of High Performance Computing Applications*, 22(2):149–165, 2008.

[130] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26(4):18–31, 2006.

[131] D. Whipp. Experiences with RTL-Synchronized Transaction Reference Models. In *DesignCon East 2003, System-on-Chip and ASIC Design Conference*, Marlborough, MA, 2003.

[132] D. Whipp. private communication, June 2010.

[133] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *SIGMETRICS Perform. Eval. Rev.*, 24(1):68–79, 1996.

[134] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[135] H. Wu, Z. Feng, C. Guo, and Y. Zhang. Ictcp: Incast congestion control for tcp in data center networks. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 13:1–13:12, New York, NY, USA, 2010. ACM.

[136] W. Wu and M. Crawford. Potential performance bottleneck in linux tcp. *Int. J. Commun. Syst.*, 20(11):1263–1283, Nov. 2007.

[137] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *SIGCOMM*, pages 351–362, 2010.

[138] Y. Yuan. private communication, June 2010.

[139] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, 2010.

[140] J. Zhang, F. Ren, and C. Lin. Modeling and understanding tcp incast in data center networks. In *INFOCOM*, pages 1377–1385, 2011.