UC Santa Barbara
UC Santa Barbara Electronic Theses and Dissertations

Title
Smart Resource Sharing for Concurrency and Security

Permalink
https://escholarship.org/uc/item/4rs6n9s6

Author
Gao, Ying

Publication Date
2017

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
Santa Barbara

Smart Resource Sharing for Concurrency and Security

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

Doctor of Philosophy

in

Electrical and Computer Engineering

by

Ying Gao

Committee in Charge:

Professor Timothy P. Sherwood, Chair

Professor Frederic T. Chong

Professor Yuan Xie

Professor Tao Yang

January 2018

The Dissertation of
Ying Gao is approved:

Professor Frederic T. Chong

Professor Yuan Xie

Professor Tao Yang

Professor Timothy P. Sherwood, Committee Chairperson

December 2017

Smart Resource Sharing for Concurrency and Security

Copyright © 2018

by

Ying Gao

Acknowledgements

I am very grateful for the opportunity to obtain an academic school life that is primarily provided by my advisor, Timothy Sherwood. Because of his words-not-able-to-describe patience, open-mindedness and wisdom, I, once started as a very junior student from a different major other than computer science or engineering, have been well guided and nurtured through the wonderful computer world over the years. In both computer architecture and operating systems, that represents two of the most critical areas of hardware and software aspects of computing, I have since tasted the best of interesting knowledge to learn and the “worst” of intriguing and difficult problems to solve.

Every Ph.D.’s experience could be properly analogized to roller-coaster rides. The thrilling ups and downs are not for the faint of hearts. I thus deeply appreciate my parents’ support and faith in me, and their strong insight in pursuing the most advanced degree. I cherish the spirit learned from my mom, who is extremely disciplined and seldom gives up. I also want to thank my dad, who consistently shares his life experience to pull me out of the numerous down times.

I also owe my sincere thank you to my senior, Hassan Wassel, who made my first publication happen. His diligence and integrity, willingness to help, insights into research and most importantly, a kind heart, set the best model as a researcher and a mentor. Every time I thought of that fall when we were discussing on the NoC project, it would still boost my enthusiasm and keep me forward.

I also appreciate the two summer internships and the people I have met and worked with from industry. The project at Intel left me tremendous interest in concurrency problems that pave the way for my future career.

Mysteriously, I want to thank a VP from Google that I do not remember the name of. Thank you for pointing out the weaknesses in my research and for changing the way I look at research problems. Your silver hair and flowing pony tail now remains an unforgettable view.

Of course not the least, a deep thank you to all my collaborators and my committee who have given inspiring opinions and strong support. Also a thank you to my teachers, fellow lab mates and class mates, for I have learned so much from all of you. At last, I thank my close friends for everything.

Curriculum Vitæ

Ying Gao

Education

2017	Doctor of Philosophy, University of California, Santa Barbara
2015	Master of Science, University of California, Santa Barbara
2011	Bachelor of Science, Tianjin University

Awards

Micro Top Pick	**IEEE Micro Top Pick** from Computer Architecture Conferences, January - February, 2014

Professional Experience

Sept 2012 - 2017	Graduate Research Assistant, University of California, Santa Barbara
Jun - Sept 2015	Summer Intern, VMware Inc., Palo Alto
Jul - Sept 2013	Research Intern, Intel Corporation, Santa Clara

Publications

2017 **Ying Gao** and Timothy Sherwood. Cashmere: Application-Driven Computation Concurrency in a Mobile Operating System, *to submit.*

2017 **Ying Gao**, Hassan Wassel, Jason Oberg, Frederic Chong, and Timothy Sherwood. Provably Non-Interfering Architecture for Secure Networks-On-Chip, *to submit.*

2016 **Ying Gao** and Timothy Sherwood. Hardware-Assisted Context Management for Accelerator Virtualization - A Case Study with RSA, *Proceedings of the 29th International Conference on Architecture of Computing Systems* (**ARCS**) 2016.

2014 Hassan Wassel, **Ying Gao**, Jason Oberg, Ted Huffmire, Ryan Kastner, Frederic Chong, and Timothy Sherwood. "Networks on Chip with Provable Security Properties, IEEE Micro: Micro's Top Picks from Computer Architecture Conferences (IEEE **Micro** - top pick), May-June 2014.

2013 Hassan Wassel, **Ying Gao**, Jason Oberg, Ted Huffmire, Ryan Kastner, Frederic Chong, and Timothy Sherwood. SurfNoc: A Low Latency and Provably Non-Interfering Approach to Secure Networks-On-Chip, *Proceedings of the 40-th International Symposium on Computer Architecture* (**ISCA**) 2013.

Abstract

Smart Resource Sharing for Concurrency and Security

Ying Gao

Different layers of the computer system, from the low-level hardware accelerators and networks-on-chip (NoC) in multi-core systems, to the upper-level operating systems and software applications, rely on the sharing of hardware computing resources. Unfortunately such sharing, when not carefully managed, can introduce a host of protection problems and sources of information leakage. We describe a set of methods by which it is possible to systematically scale performance via hardware sharing without exacerbating security properties by being aware of the design and characteristics of individual layers and components. The key to this is efficiently dealing with security vulnerabilities introduced by sharing in terms of time and space through the creation of new security-conscious sharing interfaces. In a systematic way is to first define coordination techniques into more detailed patterns, and by bridging the gap of less efficient universal measures with provably more performant and secure patterns.

Specifically we demonstrate the usefulness of a sharing pattern for hardware and software systems where separation is of concern (interference and timing channel mitigation, etc). The most important insight is that in order to fully utilize

computing resources (to improve performance and availability), the entities that share these resources must coordinate in a pre-calculated way. More dynamic approaches to improve performance and concurrency are likely to introduce new interference in the system. While we show that certain static scheduling measures in lower level hardware such as networks-on-chip can provably eliminate timing channels, the dynamic nature of software systems makes covert channels harder to be confined. Besides, software systems also face other types of security problems beyond side channels. To improve concurrency and performance without exacerbating security requires a slightly different approach.

To study the obstacles that hinder software applications' scaling in a system because of security concerns, we delve into the Android operating system and its appification ecosystem structure. A prime avenue for attack is introduced because of its distributed sharing eco-pattern. We propose a centralized approach with a single reliable service as a method to enable computation reuse among applications. The proposed centralization technique favors well-protected application-to-system communications over vulnerable application-to-application communications. Thus not only computation concurrency is boosted but also the possibility of an app being attacked through the attack-prone Inter-Component Calls (ICCs) due to possible distributed computation sharing is eliminated. This approach further enables improvements to security with the addition of a novel

application-centric grouping for isolation. We show through a prototype on Android how our approach supports and protects inter-app resource sharing, while improving concurrency at scale.

Contents

List of Figures **xii**

1 Introduction **1**
1.1 Non-Interference and Domain Isolation 3
1.2 Thesis Statement and Dissertation Roadmap 5
1.2.1 Networks-on-Chip with Provable Security Properties . . . 5
1.2.2 Hardware-Assisted Accelerator Virtualization 6
1.2.3 Application-Centric Access Control and Computation Concurrency in Mobile Systems 8

2 Networks on Chip with Provable Security Properties **10**
2.1 Related Work 14
2.2 SurfNoC Architecture 18
2.2.1 A Motivating Example 18
2.2.2 SurfNoC Scheduling 21
2.3 SurfNoC Router Micro-architecture 24
2.3.1 Partitioning Virtual Channels 25
2.3.2 Allocators 25
2.3.3 Scheduler 27
2.3.4 Pipelining and separation discussion 28
2.3.5 RTL Implementation 30
2.4 Evaluation 37
2.4.1 Experimental setup 37
2.4.2 Impact on latency 38
2.4.3 Throughput 41
2.4.4 Area and power overhead 44
2.5 Verification of Non-interference 45
2.6 Conclusions 47

3 Hardware-Assisted Context Management for Accelerator Virtualization 65
3.1 Related Work . . . 68
3.2 Baseline RSA Accelerator Architecture . . . 70
3.2.1 Montgomery's Modular Multiplication and Exponentiation 70
3.2.2 Sharing an RSA Accelerator . . . 72
3.3 Tightly Integrated Virtual Accelerator Approaches . . . 73
3.3.1 Baseline Virtual RSA Accelerator Design Overview . . . 73
3.4 Optimized Solution . . . 74
3.5 Experimental Evaluation . . . 78
3.5.1 Relative Performance . . . 78
3.5.2 Area Cost and Power Consumption . . . 81
3.6 Conclusions . . . 83

4 Application-Centric Computation Concurrency 87
4.1 Background on Android Access Model, Security Challenges and Inter-Application Sharing . . . 93
4.2 Design: Application Driven Access Control . . . 96
4.2.1 Background on Android Audio Applications . . . 97
4.2.2 Speech Recognition Libraries . . . 98
4.3 Overview of Cashmere Architecture . . . 99
4.3.1 Library Paths Identification . . . 100
4.3.2 Computation Memorization and Management . . . 103
4.3.3 Application-Guided Grouping . . . 104
4.3.4 Concurrent I/O . . . 105
4.4 Implementation . . . 107
4.4.1 Two-Phase Grouping . . . 107
4.4.2 Inner-Group Isolation . . . 111
4.4.3 Inter-Group Isolation . . . 111
4.5 Evaluation . . . 114
4.5.1 Library Call Indirection Overhead . . . 115
4.5.2 Dictionary Model Influence in Latency and Accuracy . . . 115
4.5.3 Concurrent Apps Performance . . . 117
4.5.4 Grouping Overhead and Impact . . . 120
4.5.5 Security Analysis . . . 127
4.6 Conclusions . . . 129

5 Conclusions 131
5.1 Contributions . . . 132
5.2 Looking Forward . . . 135

Bibliography **138**

List of Figures

2.1 Time-division multiplexing scheduling in a 16-node 2D mesh (only one direction of channels is shown for illustration purposes). 50
2.2 Surf scheduling in a 16-node 2D mesh (only one direction of channels is shown for illustration purposes). 51
2.3 Surf scheduling in 16-node 2D mesh with three application domains (denoted by white, grey, and black) assuming a single-cycle routers for illustration purpose. The schedule runs as white, white, grey, and black and repeats, giving the white domain half the bandwidth. A packet (the white box under the node S) belongs to the white domain is sent from the node marked by S to the node marked by R. The figure contains six consecutive cycles. At $T = 1$, the packet is forwarded on the S port in the y-dimension (which is scheduled to forward white packets). It keeps moving in the y-dimension until $T = 3$ when it needs to move in the x-dimension on the W port. The packet waits 2 cycles (T=4 and T=5) until it is the white domain's turn on the W port and finally it is forwarded to its destination on $T = 6$. Another wait may happen again in the destination router (R) to forward the packet on the ejection port waiting for the white domain's turn. 52
2.4 Partitionable virtual channels 53
2.5 Virtual channel allocator: A 3x3 separable input-first VC allocator. In this example, we assume that VC0 and VC2 are assigned to domain 0 and VC1 is assigned to domain 1. Dashed lines shows signals that can never be 1 due to route computation restrictions. This example shows that we can reconstruct the allocator into smaller ones. 53
2.6 Crossbar with input speedup to eliminate contention on switch input port between VCs from different domains. 54
2.7 Scheduler: The scheduler output is used to mask requests to the switch output ports according to the surf schedule. 54

2.8 a) Buffer write operation from west input port: input flit arrives from domain-0s VC-0 (D0-VC0), control logic selects D0-VC0 buffer queue based on the Source ID and the Destination ID of the flit. b) Result of Buffer write operation: Input flit is queued in the buffer and the tail pointer increments from 0 to 1). 55
2.9 Description of credit-based flow. a) Initial state: credit count is 4. b)-f) router 1 sends out four flits H, B, B, and T to router 2; the credit count decrements to 0. g)-l) router 2 sends out credit signals to router 1; credit count of router 1 becomes 4 again. 55
2.10 RC unit takes 16-bit heads from the flits pointed by buffer queues head pointers and generates request for output VCs for each input VC. For example, the IVC0 signal stores the output VC ID requested by input virtual channel-0 of domain-0 (D0-VC0). 56
2.11 The I/O ports of the VC allocator unit (top) and the VC unit implementation (bottom). For OVC0, for instance, the 4th bit in the 20-bit vector is set if IVC4 requests for OVC0. The vectors are then rotated based on the previously granted request such that it receives the least priority. The priority allocation logic then grants each of the output VC to the first bit that is set in the respective vector. 57
2.12 The SurfNoC schedule (SA) logic masks all the requests that do not belong to the scheduled domain for the current clock cycle or do not satisfy the credit requirements. Thus, at most, only 2 output VCs can contend for the same output port. The round robin arbiter shown above allocates each of the five output ports to one of the two potential requests every clock cycle. The 'tail_flit_ack' signal is computed by making use of the buffer head information that is globally exposed to all the units by the buffer unit. For instance, the 4th bit is set if the 4th buffer/IVC4 is going to forward a tail flit on the output port. 58
2.13 Zero-load latency for different network size and different number of security domains (the two baselines are overlapped because zero-load latency does not depend on buffers and crossbar input speedup). 59
2.14 Zero-load latency against different network size with 16 domains (the two baselines are overlapped because zero-load latency does not depend on buffers and crossbar input speedup). 60
2.15 Average latency as a function of aggregate domains offered load for 2D mesh network of 64 Nodes: We can see that latency is stable below network saturation point. 61
2.16 Throughput as a function of offered load of one domain (only one domain is injecting) for 2D 64-nodes mesh using different number of domains. 62

2.17 Aggregate network throughput as a function of offered load of one domain (all domains are injecting packets) for 2D 64-nodes mesh using different number of domains. 63
2.18 Separation of uniformly distributed bandwidth. Throughput as a function of domain 0 offered load. We can see that, by using surf scheduling, domain 1 throughput is independent of domain 0 load (same trend was measured for domain 0 throughput while varying domain 1 load). 64

3.1 Traditional RSA accelerator block architecture 71
3.2 State diagram of the original RSA accelerator design. PRE/PRFC and POST/POFC are the preprocessing and the post-processing states for domain format and carry-save format conversions. MUL and SQR stand for modular multiplication and square operation respectively. . . 71
3.3 State diagram of an example transition case in the baseline architecture. When receiving active switch signal in SQR state, it will jump to WAIT_SW state to store intermediate results in local RAM. label denotes VM ID. If the current requesting VM was in PRE state during last switch out, next_state will be set to PRE. After numerous state transitions, the VM that was switched off during SQR state might request again, and have a chance to restore its state 75
3.4 The amount of local memory needed for storing intermediate results. The y-axis denotes the number of hardware registers and the x-axis denotes timeline measured by clock cycles during a single modular exponentiation operation 75
3.5 Design of optimized context switch enforcement in detail. SW_DFF represents register arrays including Sc_SW and Ss_SW storing intermediate results at previous SP 77
3.6 State diagram of an example transition case in optimized design. The abort signal calculated from time bound directs next_state when *switch* == 1. The current task is allowed to finish current square operation when *abort* == 0, sweet_state will be updated to its next square(SQR) or multiplication(MUL) operation judged by $E[i]$ for future state retrieval. 77
3.7 Relative performance under light (a), medium (b) and near-saturating (c) workload scenarios. V-2 and v-4 denotes the default maximum number of VMs allowed to concurrently occupy the device. 85
3.8 Comparison of area costs for v-1, v-2 and v-4 designs 86
3.9 Comparison of peak power consumption for v-1, v-2 and v-4 designs 86

4.1 The Cashmere Platform Architecture. Upon an application calling a lib function, the call will be directed to the modified audio library extended with binder IPC client interface to send lib call requests to Shared Lib Service. Shared Lib Service is a registered service served as a transition channel between client apps and the Cashmere server application. Library function table is the major component of the server application, it records previously computed lib calls and will reply to future identical calls directly, only un-computed fresh lib calls will be passed on to the original library. Note that the original audio SR lib, in our case libpocketsphinx.so will be renamed so applications cannot directly call it upon and instead the modified lib is named libpocketsphinx.so. 101
4.2 Cashmere as a registered service through Service Manager provided by Android platform. Cashmere is in Android user space. 102
4.3 Average WER to varied vocabulary dictionary sizes on PocketSphinx. 116
4.4 Average latency to varied vocabulary dictionary sizes. 117
4.5 Average latency of concurrently running audio apps natively and in Cashmere. 119
4.6 Average latency of running an app with small to medium vocabulary sizes without concurrent apps, running exclusively but concurrently with additional groups, and running non-exclusively but concurrently by joining one of the additional groups in Cashmere. 121
4.7 Average latency of running an app with large vocabulary sizes without concurrent apps, running exclusively but concurrently with additional groups, and running non-exclusively but concurrently by joining one of the additional groups in Cashmere. 122
4.8 Average latency of concurrently running different combinations of apps in Cashmere. 125
4.9 Average WER to varied vocabulary dictionary sizes. 127

Chapter 1

Introduction

The dissertation presents a step towards a more systematic design methodology by which one can construct scalable components sharing across hardware software boundaries. The need to bridging the gap between the diverse heterogeneous hardware landscape envisioned by many as the future, and the designed-for-single-core simple software operating systems is acute. From the hardware perspective, as we entered the multi-core and manycore era, communications and sharing efficiency among cores has become one of the biggest obstacles to efficiency. From the operating systems perspective, managing the emerging accelerator architectures and scheduling the complex software that multiplex their resources them becomes a significant new burden. The problem is especially acute in mobile platforms where

the computing resources are diverse, energy is limited, and the demand for new features is ever present.

While scheduling for performance alone is hard enough, when coupled with security concerns, the burden is even higher. All sharing comes with the risk of information leaking. If the shared state is not managed correctly it is possible for two different subsystems to learn about each others' actions. A malicious software could gain useful information about a victim when conflicts in the resources appear. Two more malicious subsystems might even purposefully communicate through such resource contention by forming a covert channel. The classic Denial-of-Service (DoS) attack is another example of how the system can be broken down by improper sharing schemes. Of course these specific scenarios are just a tip of the iceberg of the attacks enabled by systems engineered without a mind to secure sharing. Due to the creativity of attackers and the lack of established fail-proof designs, a large body of work exists focused on vulnerability and malware studies across all system layers.

Designs that attempt to balance security and performance in the context of the coordinating hardware/software subsystems is the topic of this thesis. It is difficult to solve performance and security problems at the same time, or at least to enhance one without hurting another. However, we show in this thesis a set of three patterns for managing the above attack scenarios while maintaining performance,

two in hardware-level (networks-on-chip and accelerators) and one in software system level (Android mobile OS). We show how careful interface design coupled with proposed coordination patterns can dramatically boost system performance and concurrency by efficient sharing without compromising the original system's security properties.

Before we dive into the details of either of the scenarios, we begin with some background on the growing demand for isolation inside hardware units and among applications in the operating systems.

1.1 Non-Interference and Domain Isolation

In high-assurance systems it is a common practice to break the system into a set of domains, which are to be kept separate. These domains should have no effect on one another. For example, the Mars Curiosity rover software runs on a RAD750 processor, a single-core radiation-hardened version of the Power architecture with a special-purpose separation kernel[1]. The kernel partitions the tasks, such as guidance, navigation and the various science packages from one another to help prevent cascading failures. Future space missions are looking to use multicore systems[2][84], which adds another layer of communication, but there are serious concerns about the introduction of opportunities for interference between system components[68]. The problem is that typical networks-on-chip have many internal

resources that are shared between packets from different domains, which we would otherwise wish to keep separate. Such resource contention introduces interference between these different domains, which can create a performance impact on some flows, pose a security threat by creating an opportunity for timing channels, and generally complicates the final verification and certification process of the system because all of the ways in which that interaction might occur must be accounted for. Non-interference means that injection of packets from one domain cannot affect in any way (including the timing of delivery of) packets from other domains.

Similarly, in operating systems such resource contention also exist among applications. The problem is even more acute in mobile operating systems where a lot of resources are from I/O devices (usually only one for each type) and often request real-time processing. To prevent such contention and to keep strict isolation, many mobile systems are designed to allow only one application to access the device at a time. However, this is in direct conflict with the current need of running concurrent applications both foreground and background. For example, a user might use voice command to order food by speech recognition (SR) featured restaurant apps meanwhile discussing with friends on social apps using audio, she/he might also require the help of a SR supported search assistant app as well. In the newest release Android Nougat (2016), the split screen feature is the first time Android will allow the convenience of running multiple applications in the

foreground. To bring isolation into the OS that requires application concurrency is not an easy task, under limited hardware computing resources and energy, it becomes even more difficult when we talk about scalability.

1.2 Thesis Statement and Dissertation Roadmap

I propose that under coordinated domain-aware sharing schemes, in almost all system layers, it is possible to increase concurrency and scalability while maintaining useful security properties (e.g. isolation). I demonstrate this observation through its application in low-level networks-on-chip, hardware accelerators, and higher-level operating systems and applications. The rest of this chapter provides an overview of our proposed steps towards a system that provides superior concurrency without exacerbating isolation properties.

1.2.1 Networks-on-Chip with Provable Security Properties

One simple way to ensure separation within a networks-on-chip is to simply time multiplex the network. Each "domain" gets it's turn across the entire network at a time. However, this approach introduces huge latencies and simple ways of relaxing TDMA introduce inter-domain interference. In replacing the non-scalable

TDMA approach, we propose a technique that assures multi-way non-interference in NoCs with low overhead on latency to allow for verification of high assurance systems such as those in aerospace and automotive systems. By carefully scheduling the network into waves that flow across the interconnect, data from different domains carried by these waves are strictly non-interfering while avoiding the significant overheads associated with cycle-by-cycle time multiplexing. The technique, named SurfNoC, significantly reduces the latency incurred by temporal partitioning. We describe the scheduling policy and router microarchitecture changes required, and evaluate the information-flow security of a synthesizable implementation through gate-level information flow analysis. When comparing our approach for varying numbers of domains and network sizes, we find that in many cases SurfNoC can dramatically reduce the latency overhead of implementing cycle-level non-interference.

1.2.2 Hardware-Assisted Accelerator Virtualization

In recent years, hardware accelerators are becoming first class citizens on chips because of their usefulness in improving performance and saving power in computation intensive tasks. However, unlike CPUs that can context switch effortlessly (without losing intermediate data) through registers and deep pipe-lining, the options for time multiplex sharing on accelerators are often limited – one either

drops the current computing task or the requesting app should wait in queue for the occupying task to finish. The former usually introduces wasted cycles and the latter degrades responsiveness.

In order to fix the designed-not-for-share accelerator architectures, we examine a set of hardware design approaches whereby the interface is split providing two virtual units. We build a public-key crypto accelerator virtualization and study the trade-off between sharing granularity and management overhead in time and space. Based on observations made during the design of several such systems, we propose a hybrid local-remote scheduling approach that promotes more intelligent decisions during hardware context switches and enables quick and safe state packaging. We find that performance can vary significantly among the examined approaches, and that our new design, with explicit accelerator support for state management and a modicum of scheduling flexibility, can allow highly contended resources to be efficiently shared with only moderate gains in area and power consumption. The statically designed separation in interfaces, switching points and fixed trackable memory are all in support for isolation. The work is also the result of cooperation within hardware architecture design and software scheduling.

1.2.3 Application-Centric Access Control and Computation Concurrency in Mobile Systems

While the previous two sections introduce the smart sharing schemes in hardware designs, when looking into the higher operating system level we find that sharing schemes can also be modified to enable application concurrency. A typical example is the Android system where applications would benefit from further communication and sharing. However, due to Android's appification [5] (decentralized) nature (each app has a Linux userID along with server client communication mechanism among apps) and arguably flawed permission model (permissions are mostly requested by app on installation and the components within an app share the same permission), there has been huge number of attacks [15][23] targeting these vulnerabilities. While research on Android has seen a tremendous amount of effort towards improving the security and privacy issues[96][94][24][31][88][27], very few have explored the problem with an eye towards enhancing computation concurrency and performance.

To discover the obstacles hindering the progress of application concurrency and performance in Android, we systematically studied the system designs and components that are both vulnerability-prone and performance-unfriendly. We make the key observation that the lack of centralization in Android could be hurting both security and performance. Inspired by the PeerReview[36] concept in distributed

systems community, we propose a central platform to guide interested apps in forming groups for sharing on different I/O libraries and computations. By refraining the communication paths among collaboration-specified apps, server app side security concerns can be relieved, thus promoting confidence and an improved willingness to share. Besides, the attack-prone inter-app communications are replaced by app-to-central service communications, the client app should be less worried in being attacked during sharing. For computation heavy library sharing, the central platform employs computation memorization to record library function call chains from concurrent running apps and dispatch computations with superior scalability.

Chapter 2

Networks on Chip with Provable Security Properties

Programmers are increasingly asked to manage a complex collection of computing elements including a variety of cores, accelerators, and special purpose functions. While these many-core architectures can be a boon for common case performance and power-efficiency, when an application demands a high degree of reliability or security the advantages becomes a little less clear. On one hand, the ability to spatially separate computations means that critical operations can be physically isolated from malicious or untrustworthy components. There are many advantages to providing physical separation which have been well explored in the literature [72, 92]. On the other hand, real systems are likely to use dif-

ferent subsets of cores and accelerators based on the needs of the application and thus will require a shared communication network. When a general purpose interconnect is used, analyzing all the ways in which an attacker might influence the system becomes far more complicated. The problem is hard enough if we restrict ourselves to considering only average case performance or packet ordering, but the difficulty of the problem increases even further if we attempt to *prevent even cycle-level variations*.

In high assurance systems it is common practice to break the system into a set of domains which are to be kept separate. These domains should have *no-effect* on one another. For example, the Mars Curiosity rover software runs on a RAD750 processor, a single-core radiation-hardened version of the Power architecture with a special purpose separation kernel [1]. The kernel partitions the tasks such as guidance, navigation and the various science packages from one another to help prevent cascading failures. Future space missions are looking to use multicore systems [84, 2] which adds another layer of communication, but there are serious concerns about the introduction of opportunities for interference between system components.

The problem is that typical networks-on-chip have many internal resources that are shared between packets from different domains which we would otherwise wish to keep separate. These resources include the buffers holding the pack-

ets, the crossbar switches, and the individual ports and channels. Such resource contention introduces "interference" between these different domains which can create a performance impact on some flows, pose a security threat by creating an opportunity for timing channels [89], and generally complicates the final verification and certification process of the system because *all* of the ways in which that interaction might occur must be accounted for.

These concerns are similar to, but distinct from, the problem of providing quality-of-service guarantees. While QoS can minimize the performance impact of sharing between domains by providing a minimum guaranteed level of service for each domain (or class) [32, 33, 50, 34], as shown by Wang and Suh, quality of service techniques will still allow timing variations and thus do not truly support non-interference [89]. The only way to be certain that the domains are non-interfering is to statically schedule the domains on the network over time. However, a straightforward application of time multiplexing leads to significant increases in latencies as each link in the network is now time multiplexed between many different domains.

The core idea behind our approach is that, if a strictly time multiplexed link is seen as an oscillating behavior, we can stagger the *phases* of these oscillations across the network such that a set of "waves" are created. As these waves traverse the network they provide an opportunity for packets of the corresponding domain

to travel unimpeded along with these waves (thus avoiding excessive latency) while still requiring no dynamic scheduling between domains (thus preventing timing corruption or information leakage). Channels in the same dimension and direction appear to "propagate" different domains such that after passing through the pipeline of the router, the channel is ready to forward a packet coming from the same dimension and domain without any additional wait (unless there is contention from packets of the same domain). In this way packets "surf" the waves in each dimension. We identify the many potential hazards non-interference faces in a modern network-on-chip, we discuss the details and ramifications of our surf scheduling methodology, and we argue that our approach truly does not allow even cycle-level cross-domain interference. Specifically in this chapter:

1. We present a link scheduling scheme and network router design which simultaneously supports both low-latency packet-switched operation and non-interference between domains.

2. We show that as the network grows in size, as the number of domains increases, and as the asymmetry between domains becomes larger, the benefit for a surf-scheduled network over TDMA continues to increase.

3. We evaluate the latency, throughput, area, and power consumption of these approaches through a detailed network simulation.

4. Finally, we argue that the technique is truly sound through an analysis of the router micro-architecture and with the help of formal verification via gate-level information flow analysis.

The rest of the chapter is organized as follows. We begin with a discussion of related work and how our proposed solution fits in the design space in Section 2.1. Next, in Section 2.2, we describe the core idea behind the SurfNoC schedule followed by a detailed router micro-architecture discussion in Section 2.3. Section 2.4 presents the evaluation of the system and explores the relationship between domains, partition asymmetry, and scheduling. Then, we provide a gate-level information-flow analysis in Section 2.5 Finally, Section 2.6 concludes the paper with our final thoughts and a discussion of future directions.

2.1 Related Work

Our proposed solution to non-interference in NoCs touches on many problems that has been proposed in the literature, such as timing channels in micro architecture, QoS in network-on-chips, fault-containment and composability in system-on-chips, and security in NoCs. In this section, we will try to review some of this related work and show how our work fits in the design space.

Timing Channels and Non-interference in Micro-architecture There has been an recent renewed interest in the analysis of timing channel attacks and mitigations through micro-architecture state such as cache interference [6, 90, 91] and branch predictors [7, 8]. One approach to these problems is a technique that can verify non-interference of hardware/software systems (including high performance features such as pipelining and caching) using gate-level information flow tracking [87, 85, 86]. More recently, a NoC timing channel protection scheme for a system with security lattices was been proposed [89]. This paper proposes a priority-based arbitration scheme to allow packets with LOW labels to always win arbitration (except when they reach a pre-specified quota during each system epoch to prevent denial-of-service attacks from the LOW domain). This ensures that information cannot flow from the domain with HIGH label to the domain with LOW label, but allows for information flow in the other direction. It can be extended to multiple security labels as long as they form a lattice. In this work, we propose a technique that assures multi-way non-interference in NoCs with low overhead on latency to allow for verification of high assurance systems such as those in aerospace and automotive systems.

QoS in Network-on-chips Techniques for achieving NoC quality-of-service guarantees have been proposed based on solutions to analogous problems in macro-

scale networks. These approaches for the most part attempt to limit the rates of each flow [32, 33, 50, 34]. However, quality-of-service guarantees are known to be not sufficient for timing channel protection [89]. Optimizations that allow flows to go over their designated rate when uncontended and the lack of fault containment is problematic for high assurance systems [72] because of the high cost of any unaccounted variation in such systems. The time division approach proposed here provides for both fault containment and timing channel elimination.

Security in NoCs Security in NoCs has been studied from several aspects that focus on specific attack mitigations such as defending against denial-of-service (DoS), battery-draining attack [26] and maintaining access control of specific memory region in shared memory systems [26, 69], and buffer overflow attacks [55, 56]. Gebotys and Zhang have focused on confidentiality by providing encryption techniques for data transmitted over the NoC in a SoC setting [28]. Availability is handled in the Tile64 iMesh networks by separating (and in fact physically separating) the network accessible by user applications from the network used by the OS and IO device traffic [92]. Our scheme can protect against DoS and bandwidth depletion attacks between domains because of the static time allocation to different domains.

Non-interference in NoCs Non-interference in network-on-chips has been studied in the system-on-chip domain to provide composibility and fault containment as well as time-predictability for real-time performance guarantees [38, 65]. Composibility means that the system can be analyzed as a set of independent components which allows for easier verification of the overall system without having to verify all possible interleavings of events in the system. This has been specially critical in high assurance systems that requires very high level of verification because of safety ramifications of the system. Æthearal proposed a time-division multiplexed (TDM) virtual circuit switching network to provide guaranteed services (GS) for performance critical applications with real-time deadlines and a packet switched best-effort (BE) network for applications with less requirements [29]. A lighter version that only provides GS was proposed to further simplify routers [79, 37]. More recently, Stefan and Goossens proposed a modification on Æthearal that enables multi-path routing both static and dynamic (based on a true random number generator) in order to enhance the security by using non-deterministic path instead of source routing used in Æthearal [80]. In addition, the need for real-time worst case execution time (WCET) analysis inspired a set of work, such as, the T-CREST project which tries to build a time-predictable multi-core for real time applications. They proposed a integer programming technique

to minimize the length of static schedule of all-to-all circuit switching connections in a TDM way [75].

Regarding packet switching networks, Avici TSR network [21] uses separate virtual channels for each destination in the network but packets destined to different locations share physical channels. Under saturation, physical channels are allocated fairly, but destinations can go over their fair share when the network is not saturated which can leak information by detecting the variation of bandwidth a certain node receives.

To the best of our knowledge, our scheme is the first to provide a packet-switched network that can guarantee two-way (or multi-way) non-interference and timing channel protection in a way that is both a) provable down to the gate-level implementation and b) provides low latency overhead.

2.2 SurfNoC Architecture

2.2.1 A Motivating Example

Consider the 16-node half mesh network (channels are drawn in one direction left-to-right and top-down for illustration purposes) in Figure 2.1, assuming that even nodes belong to domain 0 and odd nodes are part of domain 1. A straight forward way to support non-interference is by partitioning the virtual channels and

time-multiplexing the physical channels and crossbars between different domains such that channels are only allowed to propagate packets from domain 0 (black) on even cycles and packets from domain 1 (grey) on odd cycles (assuming a single cycle routers) as shown in Figures 2.1 and 2.1. This time-multiplexing scheme ensures that the latency and throughput of each domain is completely independent of the timing of the other domain's load. However, this baseline scheme means that packets will have to wait an extra cycles at each hop. Even worse, as we scale the number of partitions from 1 to D, assuming a single-cycle router each packet will have to wait $D-1$ cycles per hop. This is an expensive price to pay, and one that continues to get worse the further away you attempt to communicate. If we want to hold on to non-interference, we will still need these strict time-varying partitions, but by changing the phase of their oscillations we can dramatically reduce the latencies involved.

A better schedule for time-multiplexing will make sure that domains wash over the network as a wave, such that each dimension appears to be "propagating" one domain in a pipelined fashion. Figure 2.1 shows a simplified view of this point. Every link still rotates evenly through domain 0 and domain 1, but if we consider the top row in Figure 2.1, we can see alternating channels (grey, black, grey). In the next cycle (shown in Figure 2.1, the channels used to propagate packets from domain 0 (black) will carry packets from domain 1 (grey), and vice versa.

Before entering the network, the packet waits in the injection port until its domain's turn. The schedule ensures that when the packet is ready to egress the router that there will be no delay waiting its domain's turn at the downstream router. The only exceptions to this rule are when a packet needs to change dimensions (such as when the packet turns from traveling along the X dimension to the Y dimension) and when there is contention from packets in the *same* domain.

As an optimization, we constrain our schedule such that two directions of the router propagate packets from the same domain at the same time. For example, the top-left router in Figure 2.1 propagates packets from domain 0 (black) both to the right and down. In this case, any packet which is sent in a downward and/or rightward direction will only have to wait to enter the network and will have no additional waits during turns between dimensions (again, unless there is intra-domain contention). Of course this example is very simple as it has only two domains, even divisions, and does not consider the latency of the network routers. In the next section, we will show how to devise detailed strategy for k-ary n-cube meshes and tori networks and discuss how non-interference can be shown at the level of an implementation..

2.2.2 SurfNoC Scheduling

The most basic routing algorithm in meshes and tori is dimension-ordered routing. That is, a packet walks through a dimension until it cannot move further without going farther from the destination and then transfers to an other dimension. Thus, routing is linear in each dimension which provides an opportunity to reduce wait time between hops. This way packets will only have to wait when they enter the network from the injection channel and when they change dimensions. We will describe this idea in details in the rest of this section.

The straightforward way to support time-division multiplexing is to operate the whole network in time slices that are divided between application domains. That is a packet waits at each hop until the network is forwarding packets from that its domain. This approach leads to a zero-load latency L_0 that is proportional to the number of application domains D, pipeline depth P, and the number of hops H, as shown in Equation 2.1. This solution might work efficiently for a small number of domains such as 2 to 4 domains but in high assurance applications as many as tens of domains can be found [72].

$$L_0 = HP(D-1) \tag{2.1}$$

Building on the technique we developed in the motivating example, we propose SurfNoC scheduling in which different routers (and in fact different ports of the

same router) can forward packets from different domains at the same cycle. In this schedule, a packet waits until it can be forwarded in one dimension (i.e. its output channel is forwarding packets from its domain at this cycle) and then does not experience any wait at any downstream router in this dimension (assuming there is no contention from packets from the same domain) in a way similar to the schedule developed in the half-mesh example. After finishing the first dimension, the packet may experience another wait until it can be forwarded on the next dimension. We call this schedule Surf scheduling because a packet is like a surfer who waits to "ride" a wave until some location and then waits to "ride" another wave. In this analogy, waves are dimension pipelines. Equation 2.2 shows that maximum zero-load latency and clearly shows that the overhead is additive not multiplicative as in the straightforward way. The term $(n - 1 + 2)$ comes from $n - 1$ transitions between dimensions and 2 waits during injection and ejection. It is worth noting that this is the maximum wait not the typical one as the schedule may require less wait.

$$L_0 = HP + ((n - 1) + 2)(D - 1) \tag{2.2}$$

The way to implement these different "waves" is by scheduling different directions in a router independently; an idea inspired by dimension-slicing used in dimension-ordered routing in meshes and tori. We used what we call direction-

slicing of the pipelines, such that each direction has its own pipeline. This pipeline is a *virtual* one going through different routers (not in the same router). We will describe this idea in the case of a 2D mesh or torus.

In a 2D mesh or torus, each dimension has two directions (E and W for the x-dimension; N and S for the y-dimension). The pipelines of directions of the same dimension (i.e. N, S and E, W) are running in opposite ways as shown in Figure 2.3. In this technique, each port of a router is scheduled independently of all other ports in a pipelined way such that the downstream router in the same direction will forward packets from the same domain after P cycles where P is the pipeline depth of the router. These schedules are imposed on output channels of each router to avoid timing channels based on contention in the allocator (as detailed in the next section).

Figure 2.3 illustrates an example of 16-node 2D mesh schedule of 3 domains (colored white, grey, black). There are two waves south-east (SE) (as the one shown in Figure 2.2 and north-west (NW) running in the mesh. Each channel propagates packets according to the following schedule (white, white, gray, and black) and repeats. It is worth noting that using such a schedule results in half of the bandwidth being allocated to the white domain, whereas the black and grey domains guarantee only a quarter of the bandwidth for each of them. This illus-

trates the benefit of our schedule in statically assigning non-uniform bandwidth allocation to domains.

packet ordering and deadlock freedom

2.3 SurfNoC Router Micro-architecture

The micro-architecture of the SurfNoC router has two main goals:

1. Ensuring a timing channel free contention between packets, i.e. contention can occur between packets from the same domain and not between packets from different domains;

2. Scheduling the output channels of each routers in a way that maintains the surf schedule across the whole network;

In order to achieve these two goals, we used a dynamic number of virtual channels that are partitioned between domains independent of load (§2.3.1). We analyzed the VC allocator and switch allocators to make sure they are timing-channel free (§2.3.2). The scheduling of output channels is done through masking requests to the switch allocator from packets until its turn for the output channel arrives in the wave pipeline (§2.3.3).

2.3.1 Partitioning Virtual Channels

Spatial partitioning of the queues is not a new idea [89, 21]. Static partitioning of virtual channels is done through restricting the routing algorithm so that it generates output virtual channels in the range allowed for domain of the packet. This partitioning ensures non-interference between packets from different domains while they wait in the buffers before being forwarded, i.e. eliminating the head-of-line (HOL) problem between domains.

We added support for different queue length as well as different number of queues (or virtual channels) using the same amount of storage.

2.3.2 Allocators

The SurfNoC router has two allocators, VC allocator and SW allocator. We used a separable-allocator as the baseline allocator. These allocators use round robin arbiters. This may lead to timing channels if requests are allowed from different domains to the same resource. We will detail how we prevent that from happening for both allocators.

Virtual Channel Allocators The requesters of the VC allocator are packets requesting the upstream router virtual channels. The resources are virtual channels of the upstream routers. By restricting the routing circuit to only issue

requests for virtual channels that belongs to the corresponding domain, contention is guaranteed to be between packets from the same domain. Actually, we can use this property to reconstruct the VC allocator to be D VC allocators of size $v \times v$ where D is the number of domains and v is the number of virtual channels per domain (across ports not per port) instead of one large VC allocator of size $V \times V$ where $V = D.v$. This design can help save power by power-gating some of these allocators if the number of required domains is less than D for a certain application. Figure 2.5 depicts an example of 3×3 VC allocator and illustrates the rational behind the non-interference support in the VA stage as well as the optimization of separate D allocators. This also shows that we can use any arbiters or allocator design for VC allocation because it is intrinsically interference-free.

Switch Allocator The SW allocator assigns output ports to virtual channels. Since any virtual channel can request any port, we cannot apply the same technique we used for the VC allocator of dividing the allocator into separate smaller allocators. Another problem arises from the fact that switch ports are shared among virtual channels from different domains (as shown in Figure 2.3.2) which means that requests to the switch can be denied if two VCs (belonging to two different domains) on the same input port and requesting two legitimate (according to the surf schedule) output ports will contend on the crossbar input port

leading to one of them delayed, and thus a timing channel exists. We can solve this problem by using the input speedup parameter of the crossbar with value D, and hence no contention between domains on switch input ports. Figure 2.3.2 shows an example of such configuration.

mention the time/space trade-off in the discussion section in the end of the paper

By solving the input port request of the allocator, we can now design the switch allocator as a separable one of size $Dp \times p$ where p is the number of ports of the router. It is worth noting that it does not matter if the allocator is input-first or output-first because of two reasons. First, an input arbiter is responsible for one input to the crossbar that is shared between VCs from the same domain. Second, by using dimensional order routing and the surf scheduling, a VC can request only one output port. Requests to an output port are masked using the scheduler state so that only requests from the domain which owns the current time slot reaches the allocator (i.e. no contention between different domains can happen in the output arbiter).

2.3.3 Scheduler

The scheduler is a set of p tables each indexed by a counter, one for each router output port. The initial state of the counter is pre-determined at design time in

order to enforce the surf schedule. The number of slots in the tables is determined by the number of domains. The selected element from the array is used as input to a decoder. The decoder output is used to mask requests to the switch allocator as shown in Figure 2.7. If the number of domains D is greater than the pipeline depth (including channel traversal) P, the schedule table is initialized according to Equation 2.3 where S_{id} is the schedule of port i at index d, l is the location of the node in the dimension of port i, l' is the location of the node in the other dimension.

$$S_{id} = \begin{cases} ((D-P)(l+l')+d) \bmod D & \text{if } i \in \{0,2\} \\ (-(D-P)(l+l')+d) \bmod D & \text{if } i \in \{1,3\} \end{cases} \quad (2.3)$$

2.3.4 Pipelining and separation discussion

We have so far discussed separation regarding each pipeline stage separately but the question remains whether pipelining and pipeline stalls can cause interference or not. We will discuss each pipeline stage and the basic idea is to ensure that stalls do not induce interference between separate domains.

Buffer write and route computation (BW/RC) This stage is the first stage of the pipeline and because of credit-based flow control we are assuming, flits do not enter the router unless there is a guaranteed space in the buffer for it. Spatial

separation is ensured because VC allocation is done in the upstream router. Route computation can be done in parallel for all flits at the front of all virtual channels (waiting for RC). No interference can be caused in this stage.

Virtual channel allocation (VA) At this stage all flits send requests to the VC allocator. Using our design, interference can happen between virtual channels from the same domain but not between those from distinct domains. Stalled flits because of lack of free virtual channels (in the downstream router) prevent only flits from the same virtual channel from making progress. This can be insured by recording state in the pipeline for each virtual channel, i.e. stalls due to virtual channel allocation have to be per virtual channel (not per input port).

Switch allocation (SA) Switch allocation can fail, due to contending flits for switch ports (limited to virtual channels from the same domain), which causes stalls in the pipeline. We avoid stalling the whole port (which leads to interference between domains) by having a separate state in the pipeline stage for each virtual channel. Switch allocation can also be stalled because of lack of buffering in the downstream router, i.e. waiting for a credit. This stall effect is limited to a virtual channel and can be handled using the same way the failed SW allocation stall.

Parameters	Specifications
Architecture	4*4 Torus
Routing Algorithm	Dimension order deterministic routing
Domain count	2 (D0 and D1)
Virtual channels per domain	2 (VC0 and VC1)
Number of ports	5 (West, East, North, South, and Self)
Flit Size	144 bits (payload-128 bits, head-16 bits)
Flow control type	Credit-based flow
Input Buffer queue type	FIFO queue of depth 4
Arbiter type	Priority based round-robin

Table 2.1: Specifications of implemented design.

The key idea here is stalls can affect flits in the stalled stage and all previous stages only from the same virtual channel. Thus, we can guarantee separation because we statically assign virtual channels to domains.

2.3.5 RTL Implementation

Table 2.1 briefly describes the specifications of the implemented SurfNoc router design.

Buffer Write This stage is the first stage of the pipeline, and because we are assuming a credit-based flow control, flit do not enter the router unless there is a guaranteed space in the buffer for them. Buffers are implemented as circular FIFO queues where the incoming flit is queued when it arrives from one of the input ports. There is one buffer queue for each virtual channel. Buffer unit selects one of the virtual channel buffers to queue the flit. This decision is made entirely using

the source router ID, destination router ID, the current router ID, and the surf schedule. The flits wait in the buffers to be processed by the route computation (RC) unit. The flits are de-queued from the buffer when the crossbar sends out flits to the downstream routers.

In our implementation, we used five ports, two domains, and two virtual channels per domain. Hence, there were total of twenty virtual channels across all the ports. The domain of the flit was selected based on the surf schedule and the virtual channel was selected based on whether there was a need to take the wraparound link. Wraparound link is taken when the path to the destination router is shorter through the wraparound link as compared to the normal link. Virtual channel, VC0 was used when there was no need of wraparound links. However, if the router selected the wraparound link, VC1 was used for routing to avoid the deadlock condition and hence, the flit was queued into the VC1 of the selected port and domain.

Figure 2.8 gives an example of buffer write operation. The upstream router sends flit from domain 0s virtual channel-0 buffer. The current router receives the flit at its west port. There are four possible buffers corresponding to each domain and virtual channel. However, the control logic extracts information about the source ID and destination ID from the flit and puts the flit in the west ports

domain-0 VC-0 buffer (at location pointed by tail pointer). The tail pointer increments and points to next location in the queue.

Credit Table Since we are using a credit-based flow control, the buffer unit maintains a credit table that stores the number of buffer space available in the downstream routers for each virtual channel buffer. The router can de-queue the current routers buffer and send out the flit from one of the output ports only when there is at least one buffer space available in the downstream router. The router stalls if there are no credits available for the downstream routers virtual channel buffer. Credit table is not maintained for the port connected to the processor are the flits are sent to the processor if it is ready to accept the flit. A valid signal is kept that tells if the processor is ready to accept the flits. The router looks for this signal and sends out the flit only when the signal is set.

The credit count of the credit table is incremented and decremented based on the credit in and credit out signals. The current router sends out credit out signal to the upstream router when a flit is de-queued from the current routers virtual channel buffer to indicate that a space is emptied in the current router. The upstream router in turn increments the credit count upon receiving this signal. Similarly, when the downstream router sends out flit from one of its virtual channel buffer, the current router receives credit in signal from the downstream router and

it increments the credit count for that virtual channel buffer. Figure 2.9 illustrates the above description of credits flow and the result of flit transfer on the credit table count.

Route Computation As we implemented deterministic dimension order routing, a flit first travels in east/west dimension until it reaches the destination column (router); then, it changes dimension and travels in north/south dimension until it reaches the destination row (router). If the routing distance from current router to the destination router is equal from two paths, the deterministic routing always selects the path which does not has wraparound link in order to avoid any dependency that might arise if the decision is made on fly. RC unit processes the current flit (pointed by head pointer) in the buffer queue and if the flit type is head or head-tail, it computes the output port and output virtual channel for routing. In other words, RC unit computes the route on per packet basis (not per flit). Also, similar to the buffer unit, RC unit computes the route that is based entirely on the source router ID, destination router ID, and the current router ID. After selecting the output virtual channel for the route, RC unit sends this request to the VC allocator unit to actually allocate the output virtual channel for a packet Figure 2.10.

When the RC unit takes a flit from the buffer, it already knows the domain of the flit because there are separate buffers for each domains virtual channels. Because the output port and domain are known, the RC has only two choices of output virtual channels for the selected domain and the output port. In our implementation, because there are two virtual channels for a domain, there are only two choices of output virtual channels, VC0 and VC1. RC unit selects either VC0 or VC1 based on the need of taking the wraparound link. Wraparound link is taken when the path to the destination router is shorter through the wraparound link as compared to the normal link.

Virtual Channel Allocation The VC unit performs the function of arbitrating between the 20 input virtual channels for allocating the 20 output virtual channels. The inputs to the VC unit are 20 requests from the input virtual channels, each request holding the 5 bit ID of the desired output virtual channel, as computed by the RC unit. For each output virtual channel, a 20-bit vector is extracted from these inputs. In this vector, each bit is set if the corresponding input VC requests that output VC. This concept is shown in Figure 2.11.These vectors are then rotated based on a round-robin scheme that ensures fairness in arbitration. Here the zeroth bit of the vector gets the highest priority and the nineteenth bit gets the lowest priority. In every round of arbitration, original vector is rotated a

number of times such that the request that was granted in the previous round is pushed to the nineteenth bit. Thus the request that is granted the output virtual channel in the current round of arbitration gets the least priority in the next round of arbitration. After the rotation, a priority allocator is implemented that allocates each output virtual channel to the highest priority requester which is the first valid request starting from the zeroth bit. Once an output VC is allocated, its state is changed to 'BUSY'. The output VC is freed (status changed to 'FREE') when the switch allocation unit sends an acknowledgement that the outgoing flit is the tail flit of the current packet. Thus, an output VC is allocated when the head flit of a packet is encountered and is held by the same packet/input VC till it forwards all its flits. The outputs of the VC unit are the 5-bit input VC IDs that have been granted the 20 output virtual channels and a 20-bit 'VC_alloc_done' vector in which each bit is set if the corresponding output VC is busy.

Switch Allocation The primary role of the switch allocator is to allocate the five output ports among the 20 output virtual channels. Additionally, it also sends out the tail flit acknowledgement signal for the other units, whenever the flit that is going to be forwarded on an output port is a tail flit. The main difference between the virtual channel arbitration and switch port arbitration is that the former is on per-packet basis, whereas the latter is on per-flit basis. Thus, the SA

unit performs the arbitration for the output port on every clock cycle. In other words, it does it for each flit based on the SurfNoC schedule. At a given clock cycle, there can be contention between packets of same domain only, as per the SurfNoC schedule. So, for every output port, the SA unit checks the schedule and allows only valid requests to contend for the output ports. A request is realized as valid if there is a credit available in the corresponding downstream router and if the flit belongs to the right domain. Again, the output port is granted to one of the valid requests based on a round robin scheme. The inputs to this unit are the input VC IDs that have been granted the 20 output VCs and the credit table information from the buffer unit. The outputs are the 5-bit input VC IDs that have been granted each of the five output ports and the 20-bit tail flit acknowledgement signal (Figure 2.12).

Crossbar Switch or Buffer Read The SA unit provides the connection information in the form of 5-bit virtual channel IDs for each of the five output ports. This makes a connection from the input virtual channel buffers to the output ports. The buffer unit pops out flits from these input virtual channel buffers on the corresponding output ports. At the same time, buffer unit sends out credit out signals to the upstream routers to indicate that a space is emptied in the current routers buffer.

2.4 Evaluation

In this section, we evaluate the performance and separation features of our SurfNoC scheme. We also evaluate the area and power overhead compared to a mesh network without non-interference support.

2.4.1 Experimental setup

We implemented a model of the SurfNoC router in BookSim 2.0 [22], a cycle-level interconnection network simulator. The simulator is warmed up until steady state is reached and statistics are reset, then a sample of the packets is measured from the time it enters the source queue until it is received. For latency measurements, the simulation runs until all packets under measurement leave the network. Table 2.2 provides the simulation parameters used for different schemes. We evaluated four schemes, two which do not provide separation guarantees while the other two support strong separation. The non-separation baselines are an input-queued router with minimal resources which achieves almost 40% saturation throughput (*Baseline-small*), and a similar router but with much more resources (buffers and input-speedup in the crossbar switch) which we call *Baseline-fast*. We chose to use two baselines because the separation supporting router includes more resources and would achieve more throughput than a baseline with minimal area, which will hide the lost throughput due to the static scheduling. The non-interference

Parameter	Baseline-small	Baseline-fast	Surf and TDMA
VCs	12	32	See Table 2
Buffers per VC	4	4	See Table 2
Input Speedup	1	32	See Table 2
Flits per packet	1		
Router delay	4 cycles		
SW and VC Allocators	Seperable (input-first)		
Routing	DoR		

Table 2.2: Simulation Parameters.

Domains	1	2	4	8	16	32
Number of VCs per port	16	16	16	32	32	32
Number of flits per VCs	8	8	8	4	4	4
Input speedup	1	2	4	8	16	32

Table 2.3: Different configurations

supporting schemes are a straightforward (*TDMA*) (the whole network forwards packets from the same domain) and an input-queued router which enforces the surf schedule (*Surf*). Table 2.3 shows the different configurations used for different number of domains for Surf and TDMA.

2.4.2 Impact on latency

We first examine the impact of our non-interference support on latency with different number of domains and different number of nodes under the uniform random traffic pattern. In order to understand the effect of time-division multiplexing of channels, we measure zero-load latency (latency at offered load of 0.1%

of capacity for only one domain) and plot it for different number of domains in Figure 2.13. In this figure, we plot latency in cycles (y-axis) vs. number of domains on the x-axis for two network sizes of 64-nodes (Figure 2.4.2) and 256-nodes (Figure 2.4.2). We compare 4 configurations: baseline-small, baseline-fast, tdma and surf. It is clear that the latency overhead of surf scales much better than tdma for the same network size (for example, the overhead is reduced the overhead from 66 to 19 cycles by 71.3% for network sizes of 64 nodes with 16 domains. The savings is even greater (up to 84.7%) for a 256-node network. We can see that there is one exception to this reduction in latency which happens for 5 domains. It is a subtle case that happens only for 5 domains, because the packet leaves the router after 1 cycle of switch traversal (ST), spends 1 cycle for link traversal (LT) and after 2 cycles of buffer write (BW) and virtual channel allocation (VA) in the upstream router (total of 4 cycles during which the upstream router propagates packets form other domains), it becomes ready for switch allocation (SA) without any wait using tdma leading to same latency overhead of surf scheduling. One would also notice that the benefits are higher for larger networks because of the increased average number of hops. We can conclude that, in general, the savings of surf scheduling is more scalable with larger networks as well as higher number of domains.

In order to clearly understand how the overhead scales with network sizes or average number of hops, we re-plotted zero-load latency of 2D mesh networks of sizes varied from 16 to 256 nodes with 16-domains under the uniform random traffic pattern. It is clear that the latency of both baselines increases with network size due to higher average number of hops. We can see that the overhead of surf scheduling is almost independent of network size (average number of hops) leading to a parallel line to the baseline with constant overhead of 19 cycles (except for 16-nodes) because the packet wait-time depends only on the number of dimensions and number of domains. On the other hand, the larger the network, the higher the overhead for TDMA scheduling because a packet has to wait for its turn at each hop in the path to its destination. This clearly shows that our scheme is scalable with network size and proves our intuition of latency overhead independence of number of hops.

Zero-load latency is just one latency metric, thus, we now study latency as a function of network offered load. Figure 2.15 shows average latency measured after convergence as a function of offered load for a 2D mesh network of 64-nodes under uniform random and transpose traffic patterns. We vary aggregate network offered load on the x-axis, i.e. if we have D domains, all domains offered load is the value on the x-axis. We used 2 domains in this experiment. We can see that surf scheduling maintains its latency saving at all offered load values lower

that the saturation point of the network. We can also see that loss saturation bandwidth of the separation supporting networks is small compared to that of the baseline-fast configuration. We will examine individual domain throughput of the network in the next section.

2.4.3 Throughput

We want to understand the effect of non-interference on throughput from three perspectives: single domain throughput, aggregate network throughput and single domain throughput independence of other domains load. We checked these properties for a 2D mesh 64-nodes network with 2 and 8 domains.

Figure 2.16 shows the effect of supporting non-interference on single domain throughput for the two schemes: tdma and surf. We can observe that before the saturation point, the throughput of a single domain (only one domain is allowed to inject packets in the network regardless of the number of domains) is exactly the same as if the network is not partitioned. However, we can also see that one domain saturation throughput is inversely proportional to the number of domains. In fact, it is almost half(one eighth) of the saturation throughput of baseline configuration using the same resources (buffers and input speedup of the switch) as can be seen in Figure 2.4.3 (2.4.3) for two(eight) domains. This even

distribution of bandwidth is expected because of uniformly dividing the virtual channels among domains and time-division multiplexing of channels.

In order to understand the effect of separation on aggregate throughput of the whole network, we run an experiment varying offered load of all domains from 0 to 1 and measuring the aggregate network throughput (average number of packets received during a certain time slot) of all domains for all configurations. The results are plotted in Figure 2.17 for 2 and 8 domains. In this experiment "baseline-fast" uses the same buffer and input speedup values of the separation-supporting configurations(tdma and surf) in order to measure the performance loss due to non-interference support using the same set of resources. Although we can see that saturation throughput is reduced by around 11.7%, aggregate throughput loss is only limited to 5% and 4% for 2 and 8 domains, respectively. Figure 2.4.3 clearly shows that the network can operate when offered load is below saturation throughput without any performance loss. Non-interference configurations have higher saturation throughput than the small baseline because it uses more resources, and lower than the fast baseline that include same resources because of unused time slots due to schedule enforcement. Moreover, we can see in Figure 2.4.3 that if all domains are trying to inject packets at just 10% of the network capacity the network reach saturation leading to increased latencies. This can be

tackled by non-uniformly allocating the bandwidth according to application specific requirements.

In order to verify the benefits of assigning bandwidth non-uniformly, we performed an experiment on a 2D mesh network with 64 nodes and 3 domains. Bandwidth (VCs and time slots in the schedule) is assigned as follows: quarter of the bandwidth is assigned to domain-0 and domain-1, each; half of the bandwidth is assigned to domain-2. This non-uniform allocation is done by devising a schedule with 4 slots and assigning domain-3 time slots to domain-2. Saturation throughput as expected is 0.09 for both domain 0 and 1, while it is 0.21 for domain 2. Latency at 5% injection rate is 36(53) cycles for domain-2 and 39(53) cycles for domains 0 and 1 using surf scheduling(straightforward tdma). This shows that our scheme can have latency benefits as well as throughput benefit by designing a non-uniform surf schedule.

We examine the non-interference between domains by varying one domain's offered load while keeping the other domain offered load constant at maximum in a 2D 64-nodes mesh with 2 domains. We plot both domain's throughput for baseline-fast and domain 1 throughput for surf as a function of domain 0 offered load in Figure 2.18. We can see that domain 1 throughput is independent of domain 0 traffic if we use the surf scheduling but not for the baseline case.

2.4.4 Area and power overhead

Area The SurfNoC router requires modifications to the crossbar, more buffering and bigger switch allocator (due to bigger crossbar). For a D domain network, we added D input speedup in crossbars. Crossbars area scales linearly with the input speedup D because we increase only one dimension of the crossbar. We verified this trend using DSENT [83] (with 45 nm bulk LVT running at 1 GHz with 0.3 injection rate) and it scales linearly. For example, while a 5×5 crossbar occupies 1598.08 μm^2, a 20×5 consumes are of 7525.76 μm^2 which is almost a factor of 4.7 for input speedup 4.

Baseline-small uses 48 entries per input port, assuming 32-bit flits, DSENT estimates an area of 0.0125 mm^2. On the other hand, surf and baseline-fast uses 128 entries occupying 0.0327 mm^2, a factor of 2.62 overhead against the baseline-small.

We also added the scheduler which is mainly p copies of a a counter (where p is the number of output ports), D entries memory a $D \times 2^D$ decoder and $D.p$ AND gates (assuming that each domain requests one port regardless of the number of VCs per domain). We estimate the scheduler to be of negligible area compared to the router. For example, the storage requirement for a 16-domain 5-port router is just 324 bits.

Power Having seen area overhead, we now discuss power consumption overhead. Buffers power consumption increases from 11.9 mW for the baseline-small to 29.3 mW for the baseline-fast and surf schemes, an overhead of 146%. We couldn't model crossbars power using DSENT because it uses multiplexer-based crossbars which is not suitable for large crossbars (as in the case of 80×5 crossbar, a 5×5 crossbar with input speedup of 16). However, we estimate that crossbar power consumption with input speedup would scale linearly with the input speedup because dynamic power consumption is directly proportional to capacitance which is directly proportional to wire length which increases only linearly with input speedup without output speedup. DSENT estimates a 5×5 crossbar to consume 1.24 mW of power. If we scaled that by 16x, a crossbar with input speedup of 16 would consume a 19.9 mW.

2.5 Verification of Non-interference

In order to prove non-interference between domains of our arbitration scheme, we used Gate-level information-flow tracking (GLIFT) logic [87, 86]. GLIFT logic captures all digital information, including implicit and timing-channel flows, because all information flows represent themselves in decision-making circuit constructs such as multiplexers. For example, an arbitration operation leaks information if the control bits of the multiplexers depend on one of the two domains

but it will not leak information (or cause interference) if arbitration is based on a static schedule. GLIFT tracking logic can accurately capture this fact because it is precise (i.e. not conservative in the primitive shadow gates but is conservative in compositional shadow circuit). For example, a shadow-AND gate propagates a label of HIGH only if the output of the AND gate depends on the HIGH input(i.e. if one of a two-input AND gate is LOW zero, the output is guaranteed to be zero and thus does not depend on the HIGH input). GLIFT automatically generates conservative *shadow logic* that can be used to prove non-interference between domains for a given circuit. Shadow logic is a tracking logic used as a verification technique (and is not intended to be part of the final system, thus does not cost any area or power). We integrated the scheduler (§2.3.3) enforcing the surf schedule into a Verilog implementation of a switch allocator [46]. We used a two-domain allocator that allocates requests of different virtual channels to output ports. We modified the allocator to have a request per VC rather than per input port (as in the original design [46]). We synthesized the allocator using Synopsis Design compiler, then generated its shadow logic and verified the separation property using simulation of the resulting circuit. We assigned a LOW label for VC 0 requests and a HIGH label for VC 1. We tested inputs for VCs sharing the same input port requesting different and same output port. In all cases grants signals had the same label of its respective virtual channels which

proves that grants are independent of requests from the other domain. We also reversed labels of VC0 (HIGH) and VC1 (LOW) to verify that separation holds for the other way of information flow (Domain 0 to Domain 1). This proves that the crossbar arbitration, and thus physical channels, is timing-channel free which (in addition to static VC allocation) ensures network non-interference. Freedom of two-way information flow, or complete non-interference was verified.

2.6 Conclusions

Network-on-chips play an important role in integrating many components, whether they are accelerators, cores, or memories. Not only are they increasingly prevalent in consumer general purpose silicon, but they also seeing introduction in high assurance domains where security and verification accuracy are crucial in saving time, money, and potentially even lives. Separation is an important property that allows designers to reason about systems efficiently by defining sub-components that can be verified independently while limiting the design space.

While we believe this paper is an important step regarding gate-level separation in NoCs, there are many questions that merit further investigation. First, we make no use of application-level knowledge that might shed light on the expected communication patterns. Co-scheduling communicating tasks (with global traffic knowledge) [57] to be near to one another might introduce the oppor-

tunities for non-homogeneous yet non-interfering schedules across the network. Application-level knowledge of lattice-based information flow policies might be combined with this work to allow more flexibility in scheduling between comparable domains [89]. Second, our approach uses a dimension ordered routing that is oblivious to our time-division multiplexing scheme (surf-scheduling). As such, a packet will only change dimensions after it finishes traversing one dimension. A non-interference aware routing technique might minimize wait time by introducing more turns opportunistically. In general, the relationship between interference and more aggressive optimizations would be interesting to explore. Third, there are other topologies to consider, e.g. high-radix routers such as flattened butterflies [45]. Although flattened butterflies can use dimension-order routing and thus surf scheduling might directly be applied, non-minimal routing is usually required to improve throughput. Enforcing a surf-like schedule with adaptive routing might increase latency. However, all of these open questions require a foundation from which to build.

The foundation we propose here is SurfNoC, an efficient time-division-multiplexed packet-switched k-ary n-cube network. SurfNoC exploits the dimension-ordered routing algorithms in mesh networks by scheduling channels in each dimension in a pipelined fashion so that packets propagate in the dimension as if there is no domain restrictions on channels. Packets have to wait for their domain's turn only

when they enter, exit, and potentially in changing dimensions. We discuss our wave-based domain scheduled network and describe the implementation at the level of the router micro-architecture with respect to non-interference support. Importantly, while several works have discussed interference at a high level, we believe this is the first time that true cycle-level non-interference has been proven to hold at the gate-level. In addition to the formal gate-level analysis needed to demonstrate that, we show that our schedule latency overhead scales efficiently with number of separation domains compared to a straightforward synchronous TDMA scheme (saving up to 75% of latency overhead in the case of 64-node with 32 domains). Although each domain's throughput suffers as the network is partitioned (as would be expected), the aggregate network performance remains very close to no-separation baseline. More importantly, the latency overhead remains constant with respect to network size.

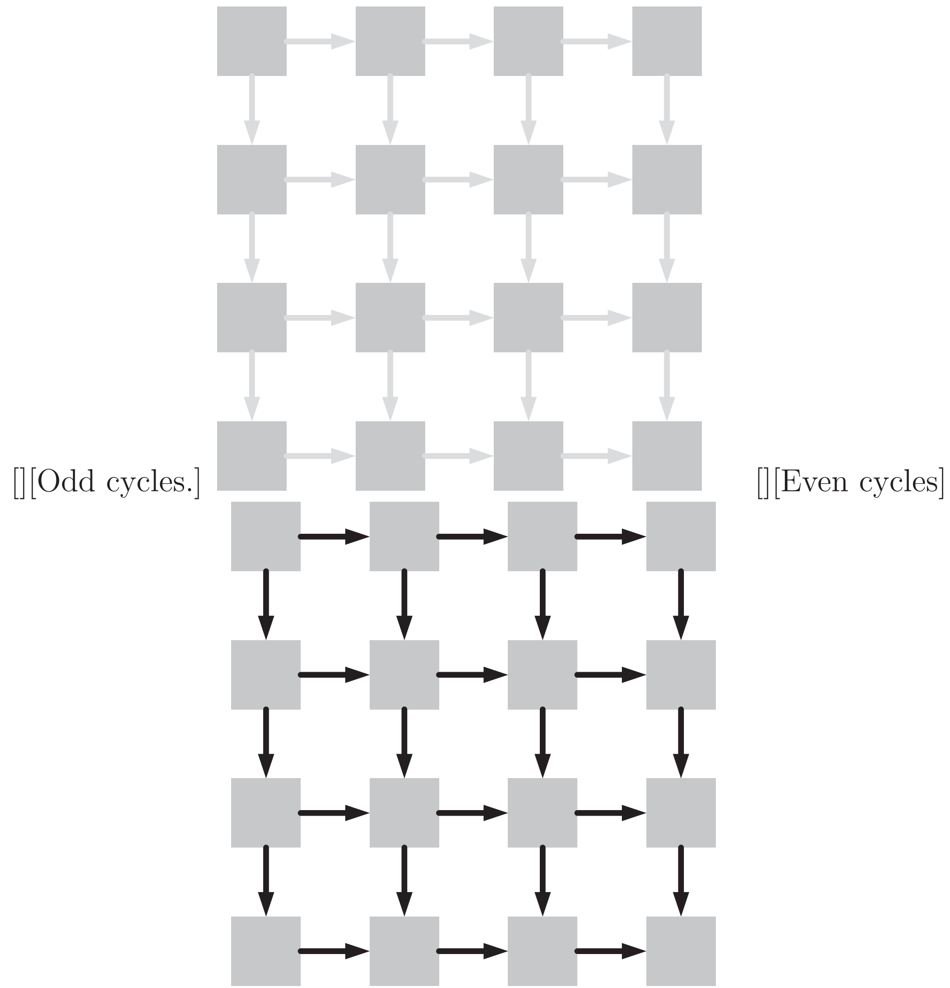


Figure 2.1: Time-division multiplexing scheduling in a 16-node 2D mesh (only one direction of channels is shown for illustration purposes).

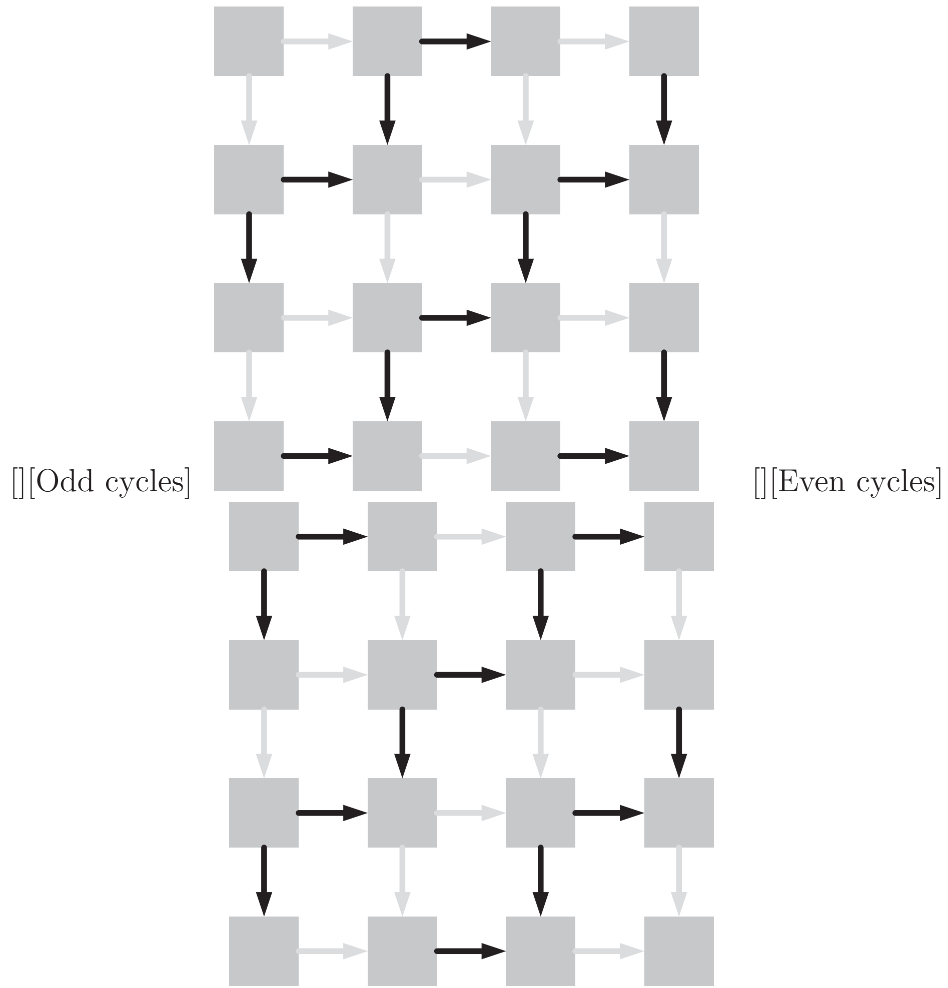


Figure 2.2: Surf scheduling in a 16-node 2D mesh (only one direction of channels is shown for illustration purposes).

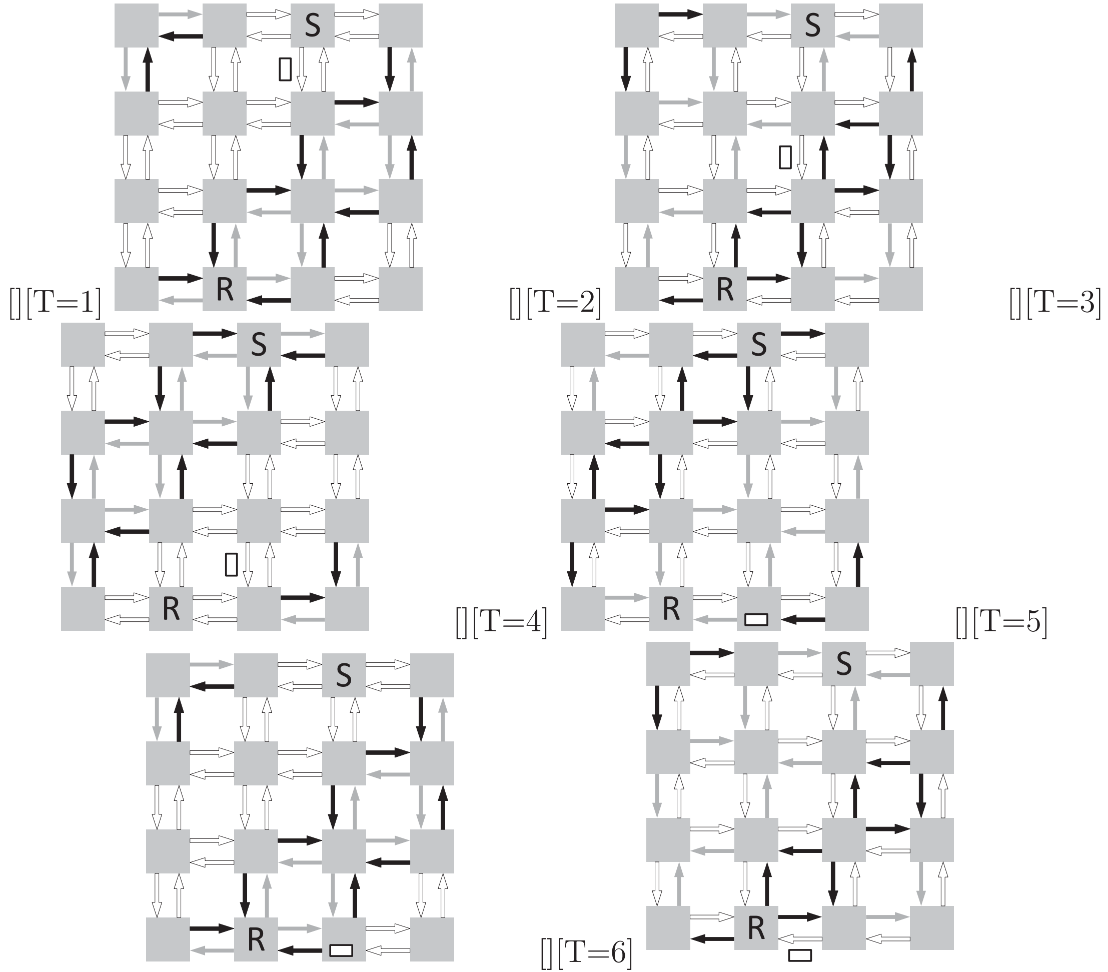


Figure 2.3: Surf scheduling in 16-node 2D mesh with three application domains (denoted by white, grey, and black) assuming a single-cycle routers for illustration purpose. The schedule runs as white, white, grey, and black and repeats, giving the white domain half the bandwidth. A packet (the white box under the node S) belongs to the white domain is sent from the node marked by S to the node marked by R. The figure contains six consecutive cycles. At $T = 1$, the packet is forwarded on the S port in the y-dimension (which is scheduled to forward white packets). It keeps moving in the y-dimension until $T = 3$ when it needs to move in the x-dimension on the W port. The packet waits 2 cycles (T=4 and T= 5) until it is the white domain's turn on the W port and finally it is forwarded to its destination on $T = 6$. Another wait may happen again in the destination router (R) to forward the packet on the ejection port waiting for the white domain's turn.

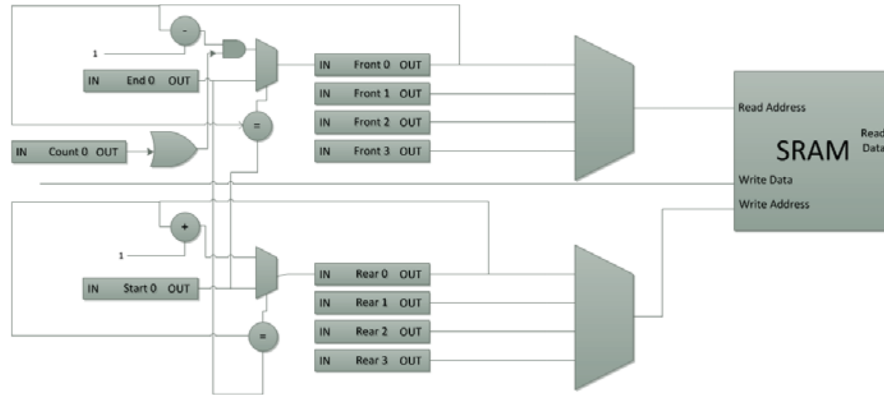


Figure 2.4: Partitionable virtual channels

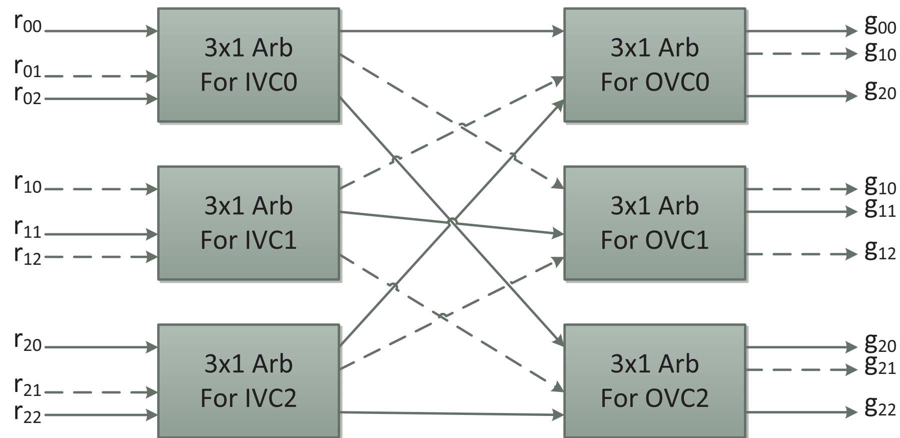
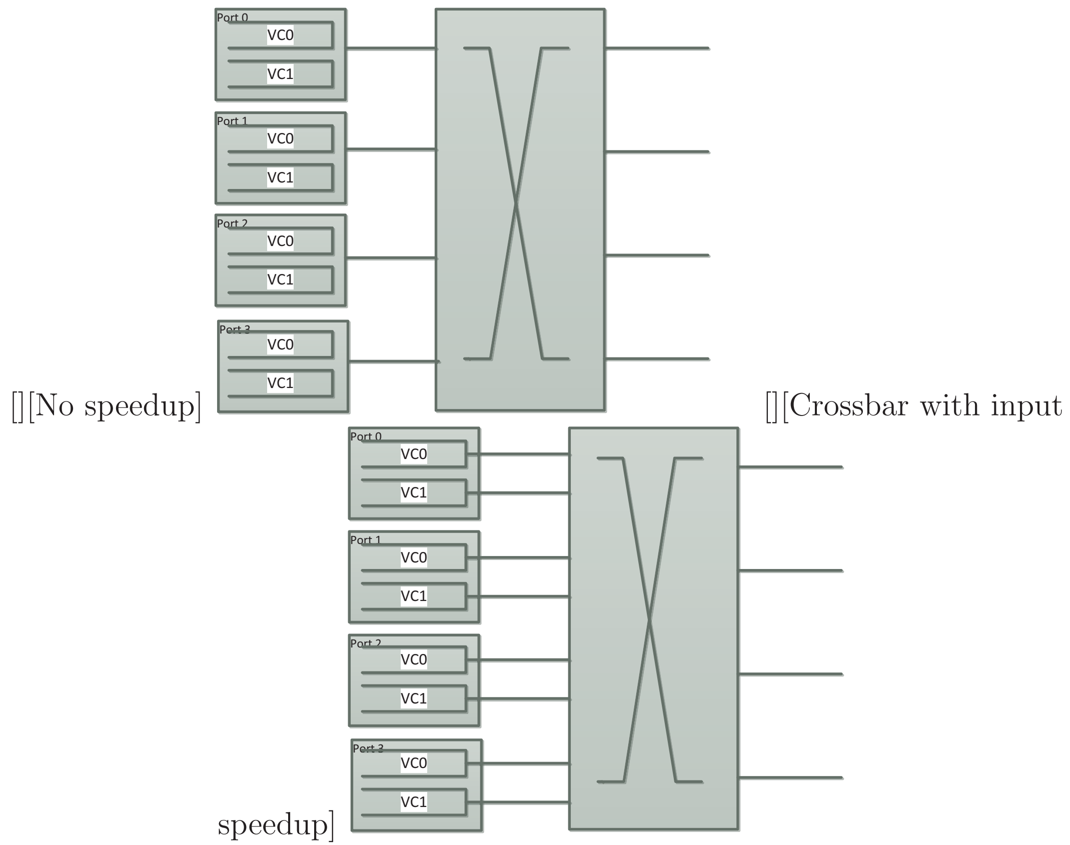


Figure 2.5: Virtual channel allocator: A 3x3 separable input-first VC allocator. In this example, we assume that VC0 and VC2 are assigned to domain 0 and VC1 is assigned to domain 1. Dashed lines shows signals that can never be 1 due to route computation restrictions. This example shows that we can reconstruct the allocator into smaller ones.



[][No speedup] [][Crossbar with input speedup]

Figure 2.6: Crossbar with input speedup to eliminate contention on switch input port between VCs from different domains.

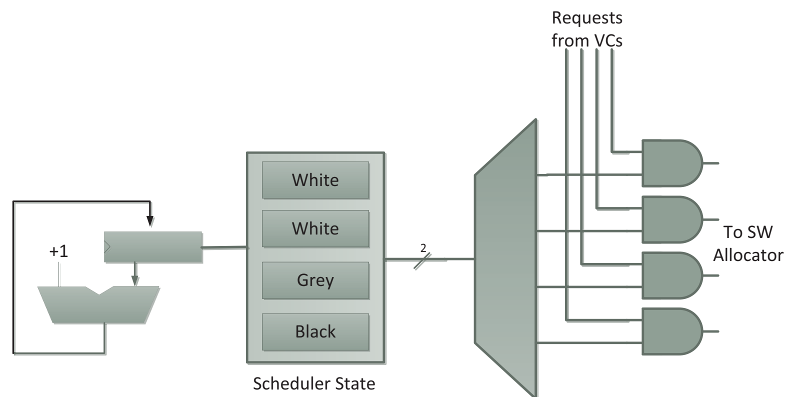


Figure 2.7: Scheduler: The scheduler output is used to mask requests to the switch output ports according to the surf schedule.

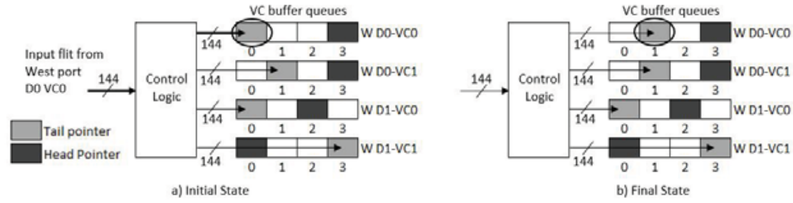


Figure 2.8: a) Buffer write operation from west input port: input flit arrives from domain-0s VC-0 (D0-VC0), control logic selects D0-VC0 buffer queue based on the Source ID and the Destination ID of the flit. b) Result of Buffer write operation: Input flit is queued in the buffer and the tail pointer increments from 0 to 1).

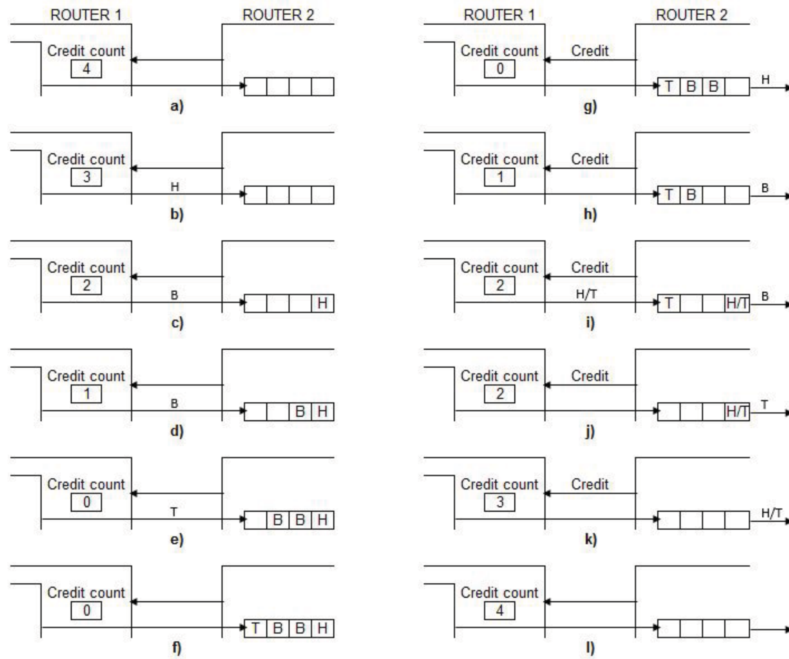


Figure 2.9: Description of credit-based flow. a) Initial state: credit count is 4. b)-f) router 1 sends out four flits H, B, B, and T to router 2; the credit count decrements to 0. g)-l) router 2 sends out credit signals to router 1; credit count of router 1 becomes 4 again.

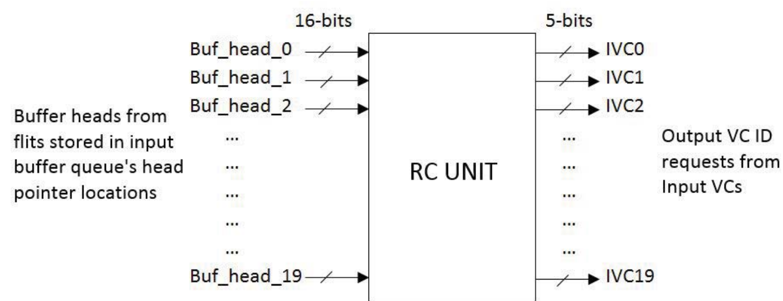


Figure 2.10: RC unit takes 16-bit heads from the flits pointed by buffer queues head pointers and generates request for output VCs for each input VC. For example, the IVC0 signal stores the output VC ID requested by input virtual channel-0 of domain-0 (D0-VC0).

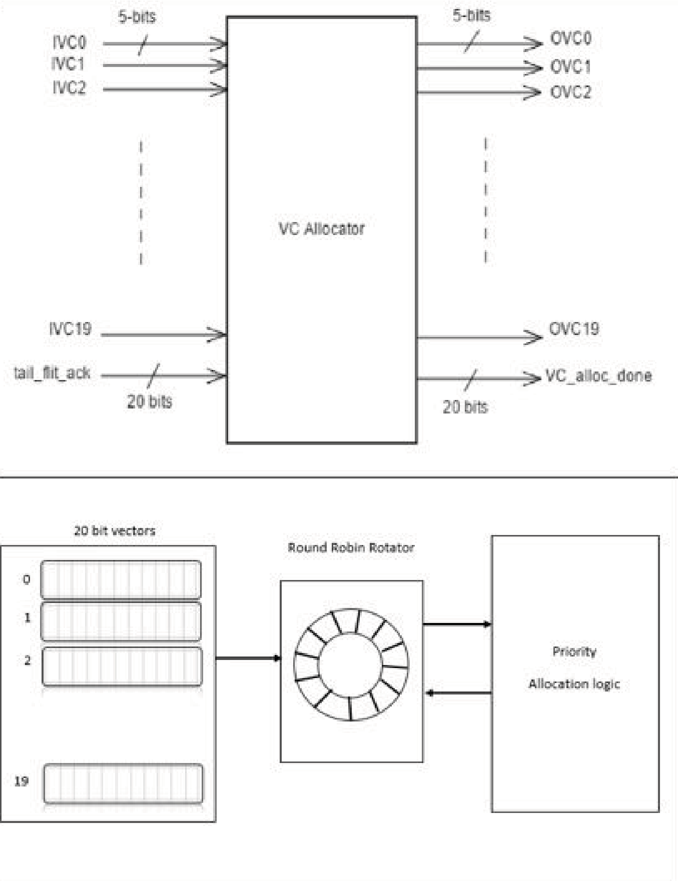


Figure 2.11: The I/O ports of the VC allocator unit (top) and the VC unit implementation (bottom). For OVC0, for instance, the 4th bit in the 20-bit vector is set if IVC4 requests for OVC0. The vectors are then rotated based on the previously granted request such that it receives the least priority. The priority allocation logic then grants each of the output VC to the first bit that is set in the respective vector.

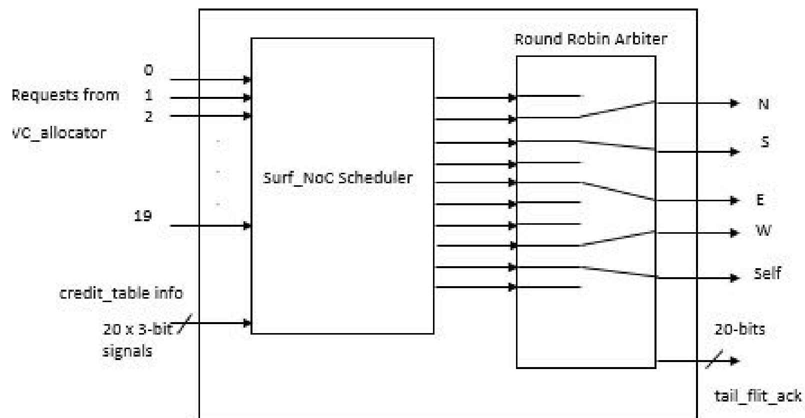


Figure 2.12: The SurfNoC schedule (SA) logic masks all the requests that do not belong to the scheduled domain for the current clock cycle or do not satisfy the credit requirements. Thus, at most, only 2 output VCs can contend for the same output port. The round robin arbiter shown above allocates each of the five output ports to one of the two potential requests every clock cycle. The 'tail_flit_ack' signal is computed by making use of the buffer head information that is globally exposed to all the units by the buffer unit. For instance, the 4th bit is set if the 4th buffer/IVC4 is going to forward a tail flit on the output port.

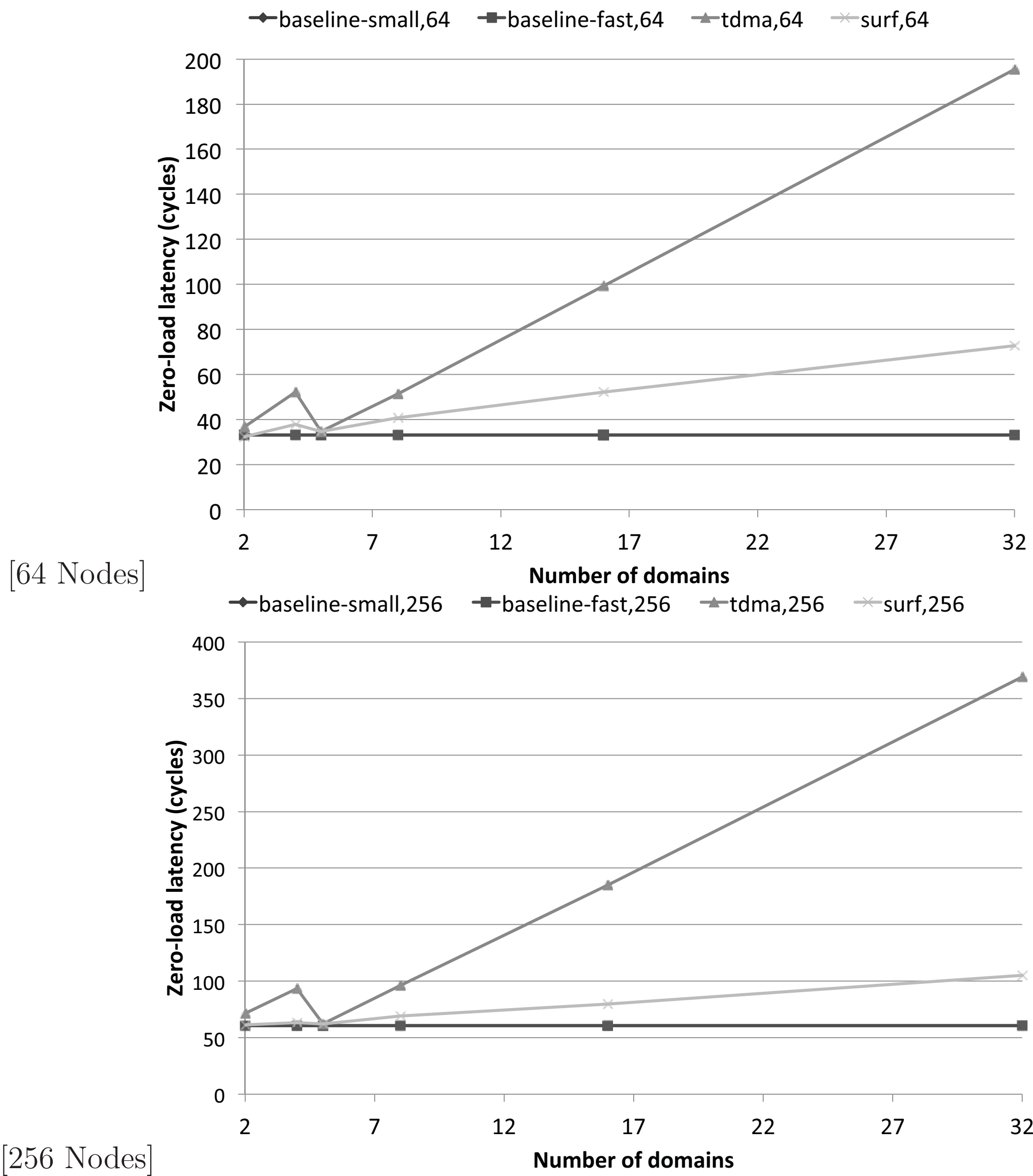


Figure 2.13: Zero-load latency for different network size and different number of security domains (the two baselines are overlapped because zero-load latency does not depend on buffers and crossbar input speedup).

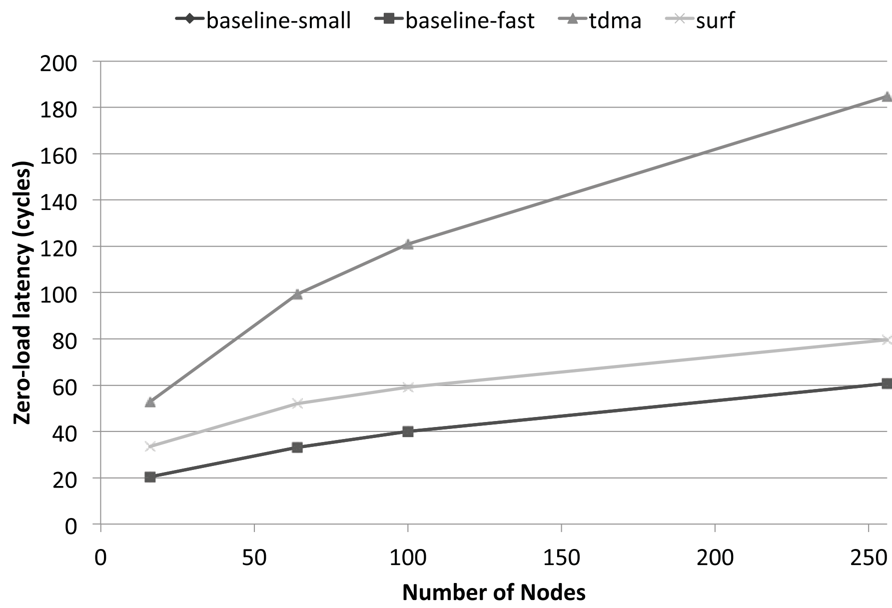
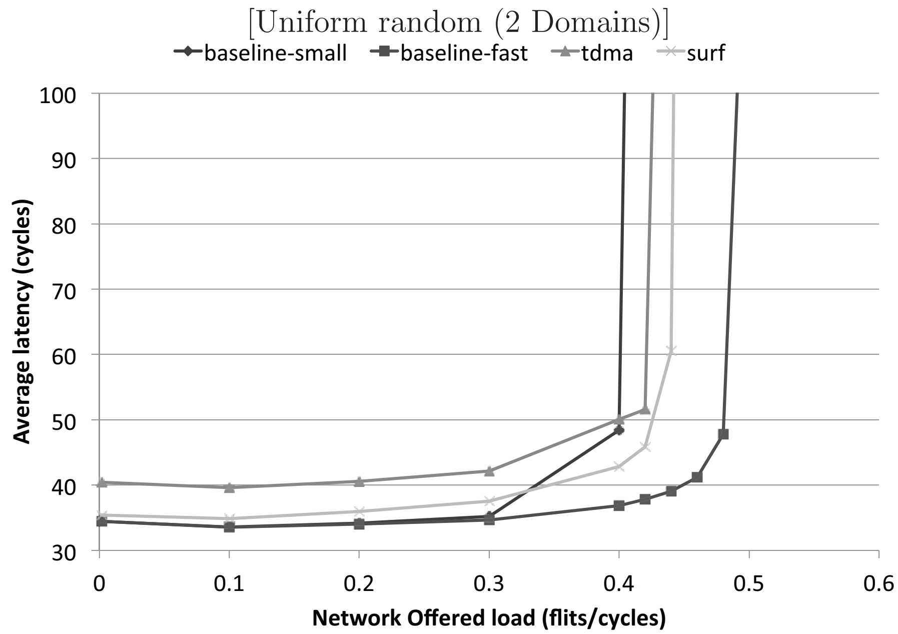


Figure 2.14: Zero-load latency against different network size with 16 domains (the two baselines are overlapped because zero-load latency does not depend on buffers and crossbar input speedup).



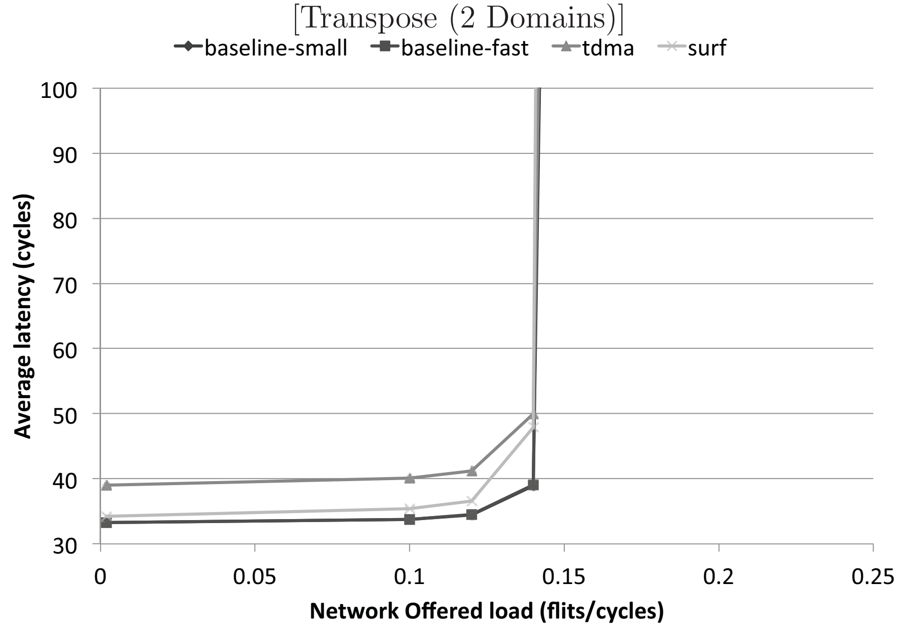


Figure 2.15: Average latency as a function of aggregate domains offered load for 2D mesh network of 64 Nodes: We can see that latency is stable below network saturation point.

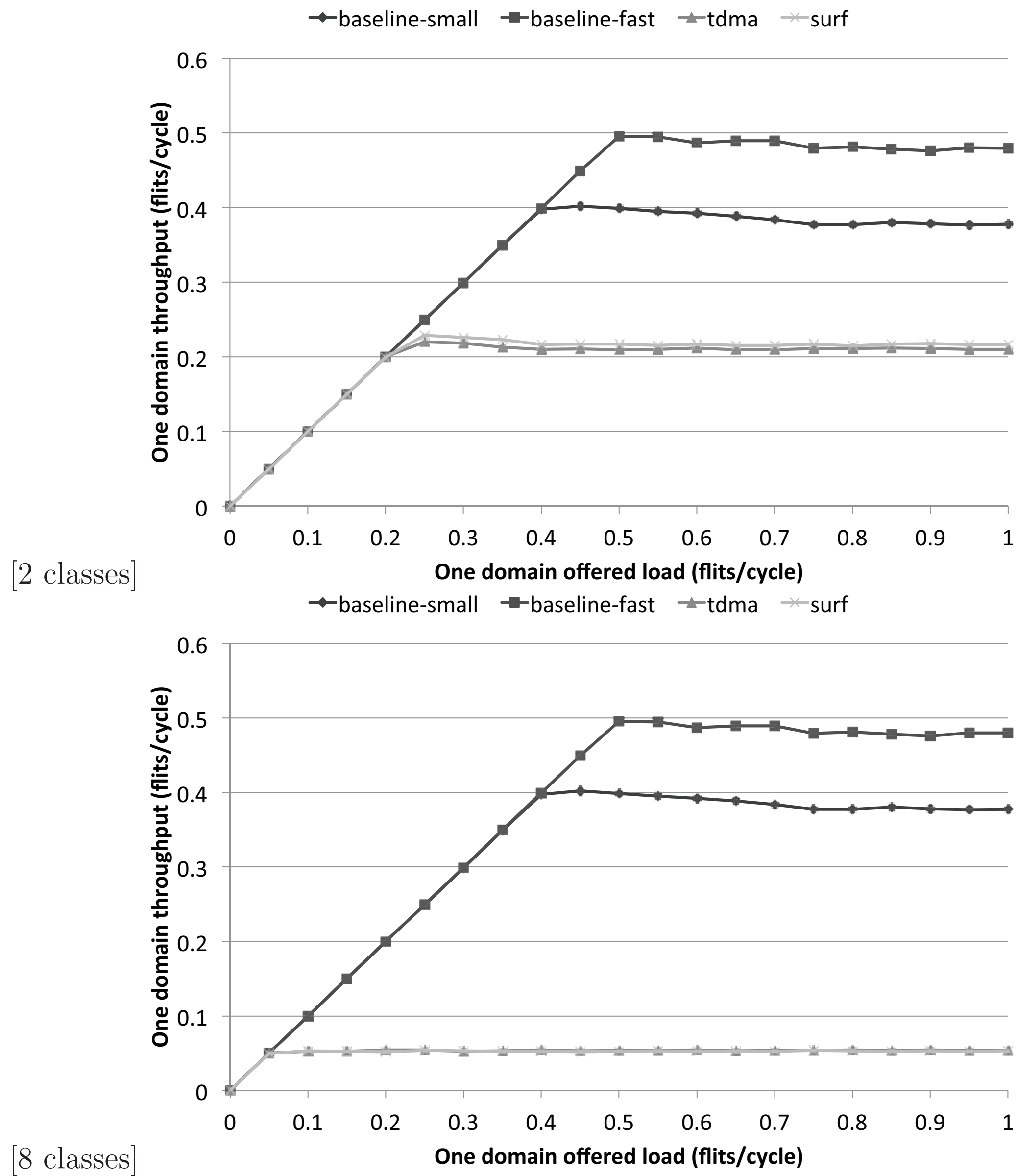


Figure 2.16: Throughput as a function of offered load of one domain (only one domain is injecting) for 2D 64-nodes mesh using different number of domains.

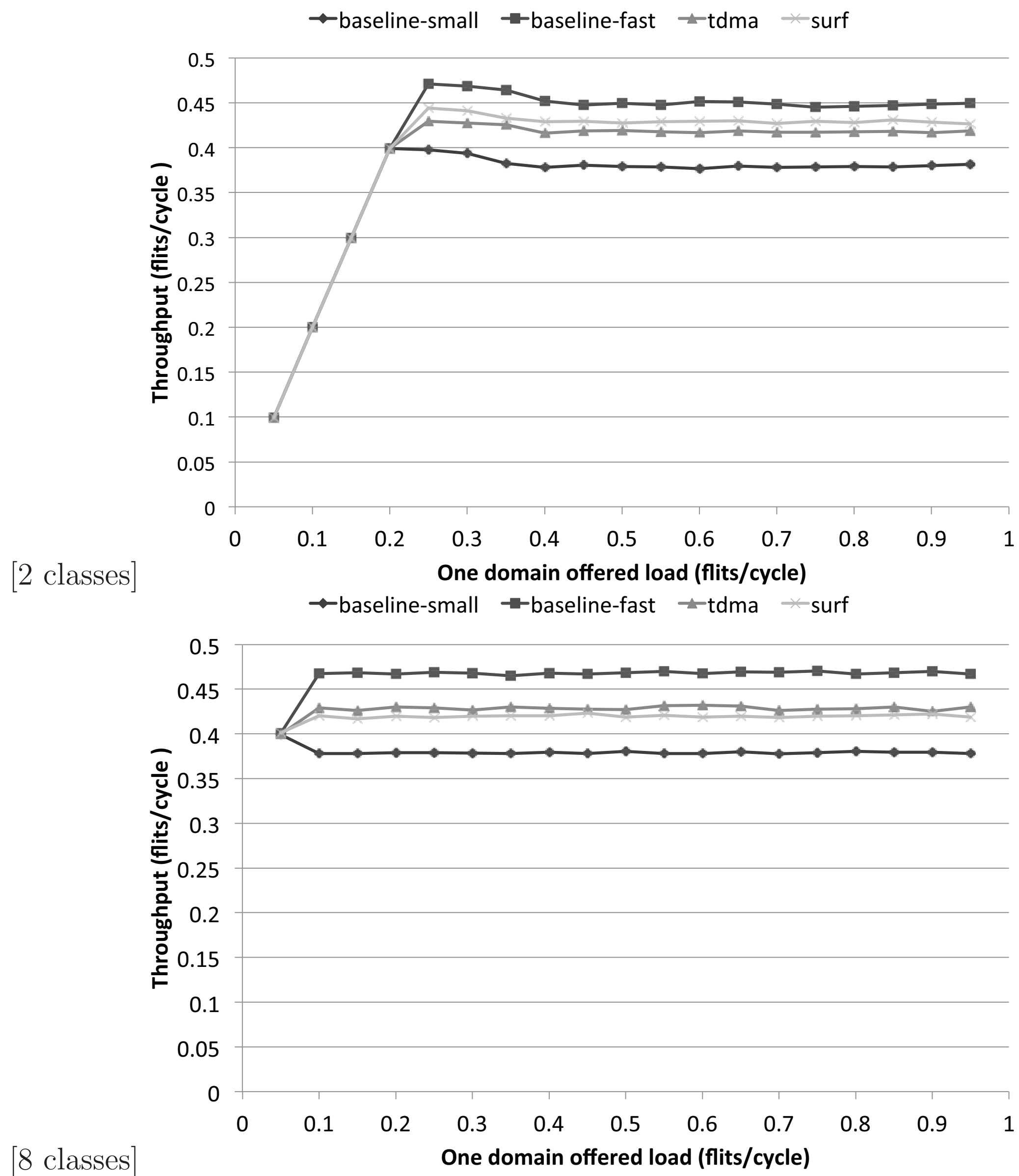


Figure 2.17: Aggregate network throughput as a function of offered load of one domain (all domains are injecting packets) for 2D 64-nodes mesh using different number of domains.

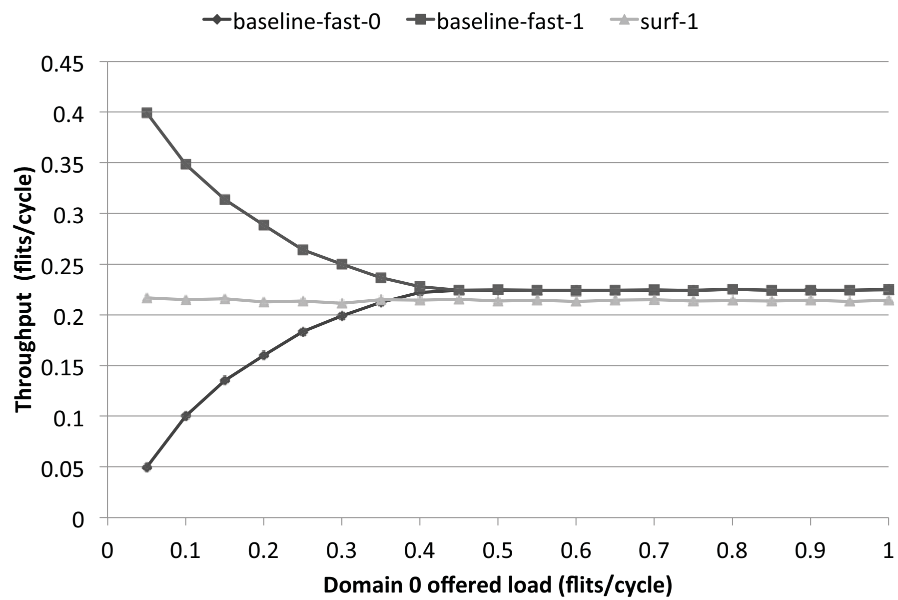


Figure 2.18: Separation of uniformly distributed bandwidth. Throughput as a function of domain 0 offered load. We can see that, by using surf scheduling, domain 1 throughput is independent of domain 0 load (same trend was measured for domain 0 throughput while varying domain 1 load).

Chapter 3

Hardware-Assisted Context Management for Accelerator Virtualization

Virtualization has emerged as a common means by which one may share and more optimally utilize underlying physical resources. As custom hardware accelerators are called upon to take significant portions of the workload from traditional CPUs, the state of computing tasks is increasingly spread across a set of highly heterogeneous devices. Effective virtualization of a system with such distributed and heterogeneous memory elements can be extremely complicated as both fine-grained scheduling and the safe management of the underlying hardware state

may be required [61] [54] [35]. For each distinct type of accelerator, the virtual machine monitor (VMM) must be aware of what subset of the machine state is critical to maintain correctness, which subset is potentially damaging if leaked to other VMs, and how critical parts of the hardware state can be managed and restored by the interface provided by that accelerator core.

This complexity also comes with a performance and system management cost, specifically in that it leads to an inability to coordinate the accelerators effectively. Switching the context for an accelerator can have a non-negligible cost (driver/OS and driver/device communication, cleanup, power management, etc) and that cost can be variable based on time. If the VMM is to coordinate the accelerators it must have an accurate view of what resources are free for scheduling and what the costs of scheduling might be. The VMM must either be able to estimate those costs from models, gather them through further communication with the accelerators (which may be then subject to delay due to resource contention), or give up the opportunity for efficient coordinated control.

There are several ways in which a designer may approach this problem. First, they might consider fixed pass-through (e.g. Intel VT-d [40]) where an accelerator is exclusively assigned to one VM, but this exclusive relationship limits sharing. A second approach is for the hypervisor to arbitrate between several VMs with one VM having access at a time, where the hypervisor halts operation of the

accelerator and restores it to a known state between guests [71]. This approach requires very little in the way of both additional memory and network communication, but carries a risk of significantly reduced throughput when interruptions cause the loss of interrupted but unfinished work. A third approach is to avoid dropping unfinished tasks, instead storing the intermediate results in memory for future retrieval. This method prevents wasting of allocated timing slots, but might incur heavy data communication [43] [47]. A fourth option is to involve the accelerator itself in the alleviation of context switch overhead. If the accelerator is granted some leeway in when the context switch occurs through a modicum of automation inside of a device, smarter switch timing might be possible saving both time and space. This might require an understanding of the computation and a careful re-architecting of the accelerator.

While performance is one important factor, the sharing of state also needs to be completed in a way that is secure. Given the importance of crypto operations, both in performance and security, they are a natural space in which to study accelerator design tradeoffs. To study the impact and suitability of different accelerator virtualization strategies and to provide optimizations for crypto devices, we implement a series of fast modular exponentiation engines. By making minimum changes to the device interface, we enable hardware assisted context management in such a way as to avoid exposing sensitive intermediate results to

the upper system and as to involve local scheduling to improve performance. Our experimental results suggest that above certain switching frequencies, the local context switch approaches achieve significantly higher throughput rate than more traditional schemes and thus enable a new level of fine-grain and fair scheduling. The additional area overhead for our baseline and optimized design to implicitly accommodate four VMs is only 36% and 15%.

3.1 Related Work

The management of accelerator-rich architectures is a very active topic of research, but much of the work is focused on application partitioning and fair scheduling, but less with VM-level sharing. HiPPAI [82] alleviates the overheads of system calls and memory access by using a uniform virtual memory addressing model based on IOMMU support and direct user mode access to accelerators. While it is efficient in limiting overheads at the user/kernel boundary, it lacks support in resource sharing. Traditional accelerator scheduling schemes still rely heavily on usage statistics collected from hardware. Pegasus [35] manages accelerators as first class schedulable entities and uses coordinated scheduling methods to align accelerator resource usage with platform-level management. Disengaged scheduling [61] advocates a strategy in which the kernel intercedes between applications and the accelerator on an infrequent basis, with overuse control that

guarantees fairness. Some work tackles the management problem by simplifying accelerator/application integration. VEAL [17] proposes a hybrid static-dynamic compilation approach to map a loop to a template for inner loop accelerators. DySER [30] utilizes program phase and integrates a configurable accelerator into specialized data-path to dynamically encode program regions into custom instructions. While these approaches are intelligent in software partitioning and mapping, they fail to take advantage of hardware assistance in resource managing. Some work starts to look into hardware device reusability: CHARM [20] and CAMEL [19] tackle the sharing and management problem mainly by automating composition of accelerator building blocks (ABBs), primarily stateless arithmetic units in ASICs.

Some projects favor managing hardware states implicitly. Task specific access structures(TSAS) [43] inserts a multiplexer as the input of each FF to select between updating its value from the combinational logic or from previously stored data, or simply remaining its value from the last cycle. This scheme takes the majority of the context switch workload within the device and enables fast switching, but at the sacrifice of non-negligible augmented logic and memory. Hardware checkpointing [47] where the hardware states of a device can be stored and be rolled back regarding checkpoint, hold the potential to minimize area overhead wisely. We recognize the value of hardware checkpointing - in fact we extend its

role in coordinated resource management: for accelerators like an RSA engine that implements real-time requests, hardware support in context management will be of great help to fast and fine-grained accelerator sharing.

3.2 Baseline RSA Accelerator Architecture

3.2.1 Montgomery's Modular Multiplication and Exponentiation

The core computation in an RSA crypto engine is modular exponentiation, consisting of a number of modular multiplications. Montgomery's modular multiplication algorithm [64] employs simply additions, subtractions, and shift operations to avoid expensive divisions. In this paper we work with an extension to this algorithm [77]. Three k-bit integers, the modulus N, the multiplicand A and the multiplier B are needed as inputs for computation.

Algorithm MM_UMS is defined as follows:

for i = 0 to i = k - 1 :

$$q = (S + A * B[i]) \; mod \; 2 \tag{3.1}$$

$$S = (S + A * B[i] + q * N) \; / \; 2 \tag{3.2}$$

S is restructured in carry-save form as (Sc, Ss) where Sc and Ss respectively denotes the carry and sum components of S. H-algorithm [16] transforms the computation of modular exponentiation into a sequence of squares and multiplications. Square operation could be performed when both multiplicand and multiplier are identical. The modular exponentiation algorithm, ME_UMS(M, E, N), iteratively applies a unified multiplication or square operation, where for each bit $E[i]$ in exponent E, both a single square operation and multiplication will be performed when $E[i] = 1$ while only a square operation will be performed otherwise.

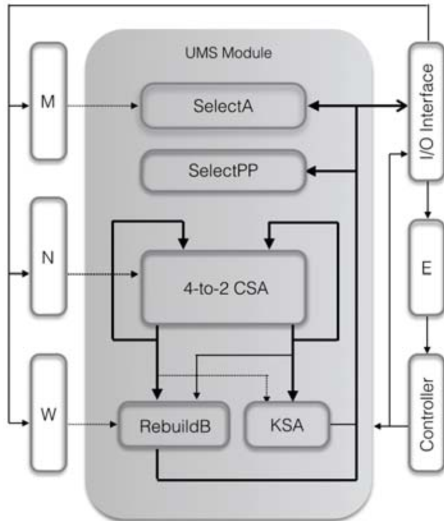


Figure 3.1: Traditional RSA accelerator block architecture

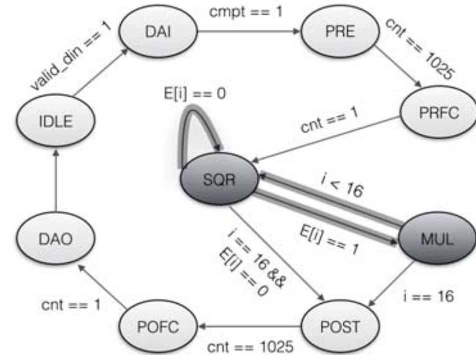


Figure 3.2: State diagram of the original RSA accelerator design. PRE/PRFC and POST/POFC are the preprocessing and the post-processing states for domain format and carry-save format conversions. MUL and SQR stand for modular multiplication and square operation respectively.

Figure 3.1 shows the baseline design. The unified modular multiplication/square module is highlighted in the shadowed region. The nine states in Figure 3.2 cap-

ture the major stages of the entire modular exponentiation process, as discussed in the algorithm ME_UMS.

3.2.2 Sharing an RSA Accelerator

One traditional method of device sharing is hard preemptive multitasking. The obvious drawback is that as the switching frequencies increase during heavy sharing, the throughput rate might suffer significant degradation.

To avoid the cost, two options are clear, either the OS relaxes its schedule to wait for the task to complete or the intermediate results from hardware have to be saved for future retrieval. The first option is becoming increasingly difficult since an application can occupy several accelerators simultaneously, thus a perfect point where all devices have just finished their current tasks can be extremely hard to identify or even exist. The latter option seems to comply well with software schedules, but the data movement required to store the intermediate results, coupled with the corresponding memory updates, making this option surprisingly tricky to execute well in practice. Moreover, exposing intermediate results to DMA are also risky due to DMA attacks [81]. A good solution should manage these burdens carefully and a new set of interfaces is needed to simplify the synchronization process.

3.3 Tightly Integrated Virtual Accelerator Approaches

The simplest tightly integrated design might store all local state in a set of D flip-flops sprinkled throughout the design. However, this approach is also prohibitively expensive. Simulation results suggest that regarding area (and power) efficiency, such virtualized accelerator can add up to a 78% area overhead.

So what can we do if we want to maintain the accelerator's capability of being fast switched without giving up almost nearly all of our efficiency? We describe two different solutions – the simplest being to replace the local and distributed storage elements with a set of RAMs.

3.3.1 Baseline Virtual RSA Accelerator Design Overview

In general, most sharing patterns fall into one of the four categories:

- Double Vacancy. No VM is occupying the device.
- Single occupancy. The accelerator is currently dominated by one VM while another VM requires input data streaming for starting a new task.
- Double occupancy. One VM is scheduled to resume a previous uncompleted computation while the other VM is in the process of computing.

- Single occupancy. One VM requires to resume a suspended task while the other VM is performing output data streaming.

Note that in scenario 3, two whole sets of states need to be stored. Based on this, we include 2KB of RAM alongside the core for temporary storage. We build a simple layer above the RSA accelerator to forward switching commands rather than changing the slave interface directly. We add a *switch* signal underneath the layer to help the controller determine the next state. In order to be able to interrupt a task in the middle of such computation, four more states are added to the FSM. We show the resulting state diagram in Fig. 3.3.

By enabling hardware preemption, the proposed accelerator virtualization approach successfully realizes the goal of abstracting away hardware details from software without abandoning tasks, at the sacrifice of increasing critical path delay by 16%.

3.4 Optimized Solution

In order to eliminate the increased critical path delay, we examine the registers that contain useful intermediate results along the entire process of a single modular exponentiation task. We measure the amount of memory needed to store these intermediate results against execution time cycle Fig. 3.4.

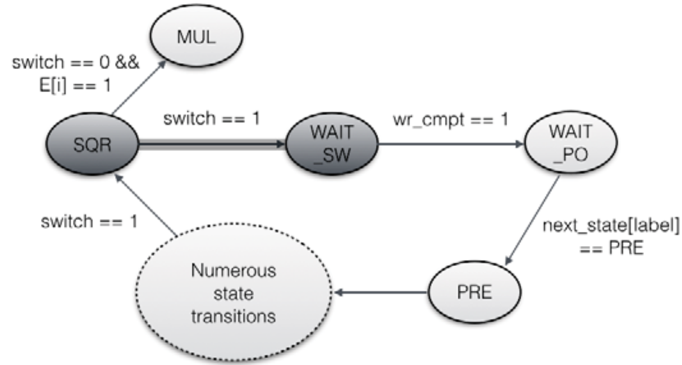


Figure 3.3: State diagram of an example transition case in the baseline architecture. When receiving active switch signal in SQR state, it will jump to WAIT_SW state to store intermediate results in local RAM. label denotes VM ID. If the current requesting VM was in PRE state during last switch out, next_state will be set to PRE. After numerous state transitions, the VM that was switched off during SQR state might request again, and have a chance to restore its state

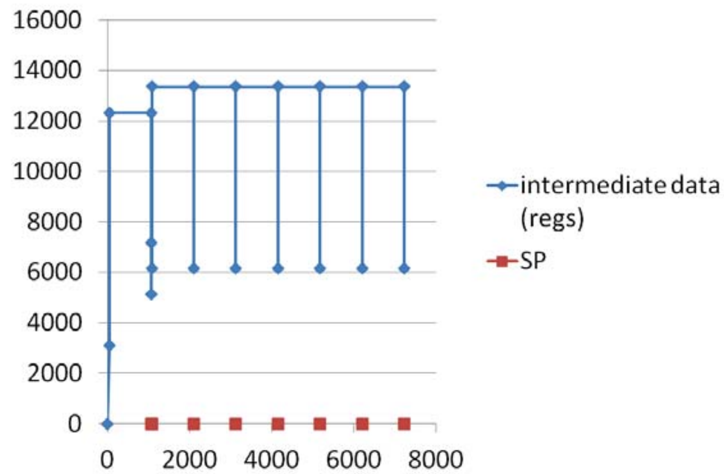


Figure 3.4: The amount of local memory needed for storing intermediate results. The y-axis denotes the number of hardware registers and the x-axis denotes timeline measured by clock cycles during a single modular exponentiation operation

At the completion of a modular multiplication or a square computation, only the value of Sc and Ss (1025-bit register arrays) are a must-save among all the large register arrays. These transition points, which we informally call SP (*sweetspots*), can be intuitively pinned from the FSM inside the device controller. If we can make sure all switching operations happen at these sweet spots, we can significantly reduce the RAM size required.

To achieve this goal, the device controller is slightly modified to ensure switching always happens at these spots. Upon each major state transition, the contents of Sc and Ss will be forwarded to two designated register arrays Sc_SW and Ss_SW. Note that the contents of these two registers will be refreshed every time an SP is identified and will be flushed during switching operation Fig. 3.5.

We also want to make sure that the OS gets control of the preemption delay so that it can make scheduling decisions easily when it needs to context switch among a number of concurrent applications. Upon receiving the switching command, the device will compare the time bound to its backward counter and make a decision about whether to reach the next SP or to simply fall back to the last stored one. If the time bound is equal to or smaller than the value of the counter, the current multiplication computation will be abandoned and contents of Sc_SW and Ss_SW will be stored to RAM. An extra state, $SWEET$, is added to allow data transfers between registers and RAMs during task switching. The state that leads

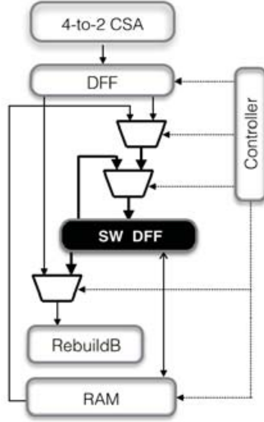


Figure 3.5: Design of optimized context switch enforcement in detail. SW_DFF represents register arrays including Sc_SW and Ss_SW storing intermediate results at previous SP

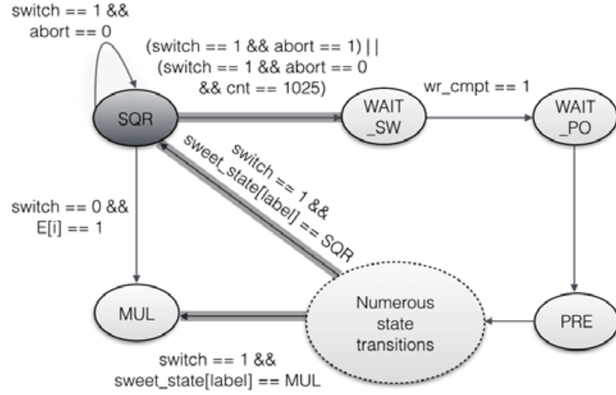


Figure 3.6: State diagram of an example transition case in optimized design. The abort signal calculated from time bound directs next_state when $switch == 1$. The current task is allowed to finish current square operation when $abort == 0$, sweet_state will be updated to its next square(SQR) or multiplication(MUL) operation judged by $E[i]$ for future state retrieval.

to $SWEET$ will be recorded. The relaxed timing bound can be very convenient for scheduling purposes, considering it is difficult for OS to decide the exact best timing to switch in a device. Granted with local scheduling power, the device can wisely help a task fully utilize its time slots. We show an example state transition scenario in Fig. 3.6 for illustration.

The design removes multiplexer arrays from the critical paths, significantly lowering area cost. Meanwhile responsiveness to interrupts or context switch commands is still guaranteed. Note that these modifications can be generally applied to public-key crypto accelerators. By simplifying the device interfaces, the VMM's scheduling becomes easier and more flexible. Tasks with higher priorities

can always be ensured a quick access to hardware acceleration. The hardware accelerator manages to secure itself in a blackbox, without exposing hardware information unnecessarily.

3.5 Experimental Evaluation

Our evaluations are based on RTL prototypes of accelerators with standard AHB I/O interfaces written in Verilog under the ModelSim [63] environment. We test through the encryption process and use a Verilog testbench with a public exponent 65537 and modulus generated from OpenSSL [67] for encryption. We synthesize all of the RTL designs using the Synopsys Design Compiler [18] with a 45nm library and collect critical path delays and executing clock frequencies. The area and peak power models for our embedded memories are based on CACTI 5.3 [4] and for logic and registers based on the results from Design Compiler.

3.5.1 Relative Performance

One important measure of performance is the total virtualized device throughput as measured by the numbers of encryptions per second. We compare the virtualized throughput rate from each of the designs to the upper bound of performance where each VM is given a completely independent copy of the device (i.e. no in-

terference at all). We simulate three scenarios representing light (concurrently running two VMs while requests from each VM fills 10% of its timeline), medium (two VMs with 20% requests) and near-saturating workloads (four VMs with 20% requests) respectively. The only contention for the crypto accelerator is from multiple VMs attempting to access the engine at the same time. Figs. 3.5.1, 3.5.1, and 3.5.1 depict the relative performance of virtualized devices under these loads respectively.

The y-axis of each of these plots is the relative performance of the different schemes (as compared to our ideal case). The x-axis is the time slice granularities under which the VMs are driving our accelerators. To simulate the fact that one does not switch between VMs instantaneously, a running task will not attempt to switch in a period smaller than a defined time slice. In real-time, latency sensitive, or reactive systems a design may be called upon to switch very quickly. To quantify the suitability of each of the previously described accelerator virtualization under various different switching speeds, we inject requests with the size of one task (for us, the crypto operation) but constrain the minimum window under which those switches might occur. The courser the minimum time between switches, the less we would expect to lose in wasted cycles as computation is abandoned on a switch, but more applications will have to wait to get their computation onto the accelerator. On each of the graphs there are 3 different bars labeled “Tra” for

“Traditional” which drops unfinished tasks on a switch. “Base” saves all of the hardware state as described in Section 3.3.1. Finally, “Opti” adds the hardware necessary to allow the accelerator to delay the switching under a fixed bound as described in Section **??**.

As can be seen in these graphs, when the request workload is comparatively low, the performance disparities among the three approaches are not as significant as those when task workload is heavier. However, the performance of the optimized design is consistently the highest throughout all the switching frequencies simulated. The base design has a slight advantage over traditional design when the time slice is smaller than the time for one encryption operation. The advantage more fully manifests when the amount of requests increases. The performance of all three approaches in all the scenarios reaches a peak around and slightly above 25 time slice. However, when we compare 25 to 100 granularities of the three figures, we can clearly see that the peak period tends to shrink as the workload increases. When reaching a comparatively coarse grain scenario around 200 the performance of all of the virtual devices suffer significantly. In these situations the bounds on switching time is large enough to cause a significant amount of idle time in the hardware. The optimized design outperforms the baseline consistently because the more restricted save points limit the hardware needed and the longer paths they cause.

One interesting observation is the non-monotonic performance of the traditional design. The throughput rate drops as the switching frequency rises until it reaches around 1/25. The reason behind this pattern is that when a task is switched off and dropped, the device is more likely to waste more computation cycles when the device is only allowed to be switched at a granularity slightly smaller than time of one operation. Provided that the v-4 optimized design shows at most a 3.6X performance improvement compared to the traditional design, in 20% workload scenario and reliably high efficiency throughout fine-grain granularities, the optimized design appears to be a clear choice in systems requiring very fine-grain switching when we consider performance alone.

3.5.2 Area Cost and Power Consumption

To model the area overhead and power consumption of the three virtual accelerators, we synthesized our RTL design in the TSMC 45nm technology. Results show that the original accelerator occupies $0.11mm^2$ with a peak power consumption of 54.7mW at 1.6GHz. Due to the lack of publicly available SRAM compilers in this technology, we use CACTI 5.3 [4] to estimate the area and power of RAMs.

The area cost is shown in Fig. 3.8. The y-axis of the area plot is the absolute area costs measured in mm^2 unit of the different schemes. The x-axis is the number bounds of VMs that are allowed to be running concurrently on the accelerator

(e.g. 2 corresponds to v-2 design). As we can see in the graph, the additional area overhead of the baseline design compared to the traditional design can increase area by up to 29% for v-2 and up to 36% for v-4. This extra price paid is primarily due to the additional arrays of multiplexers needed to switch between states and the additional RAMs needed to store the contents of all registers. Note that the optimized design scales better than than the baseline. The area overhead is merely 12% and 15% for v-2 and v-4 respectively.

Similar to the area costs trends, plots of the peak power consumption present an increasing pattern somewhat proportional to area costs. As we can see from Fig. 3.9, where the y-axis denotes the absolute peak power consumption measured in mW unit of the different devices, the optimized design scales better from v-1 to v-4 than the baseline design as the default bounds of running VMs increase. The traditional design stands out due to its more uniform power consumption.

An important conclusion is that the baseline design performs slightly worse than both the traditional and the optimized design regarding power consumption, whereas the traditional one suffers significantly reduced throughput/watt rate for near saturating workloads when the time slice is small. While baseline and optimized design already provide with most responsiveness, it is as well likely that energy consumption can be compensated from simplified software level synchro-

nization. Moreover, the internal memory read/write structure guarantees a quick and safe access to intermediate data without dealing with I/O hazards.

Due to its performance and power-friendly benefits, the optimized design improves the throughput/watt rate by at most 3.1X over traditional design when above switching frequency of 45 KHz magnitude and remains competitive to traditional design throughout all sharing granularity range under examination.

3.6 Conclusions

Growing heterogeneity in hardware devices continues to put easy and safe management in direct conflict with fine-grain scheduling and virtualization. Rather than take a top-down approach requiring that all accelerators be implemented in a particular style, we take a bottom up approach, looking at what it takes to manage the state of a device. In particular we found that there is a small but non-negligible penalty for adding in explicit access to the accelerator state both in terms of area and power. However, we also observe that there is an interesting and previously unexplored trade-off between the scheduling power one imbues the accelerator with and the efficiency with which the schedule can be managed to minimize the waste of timing slots.

With that said, under these limitations we presented comparisons of three different accelerator virtualization schemes working to manage a critical device - an

RSA accelerator. When a high degree of sharing and switching is required, the traditional task-dropping scheme can suffer significant performance degradation. If such conditions are expected, a hardware preemption scheme can be adopted, and with a bit of analysis, is able to alleviate the burden of resource scheduling and context management, and to prevent sensitive intermediate data exposure. Results show that our proposed approach manages to dramatically diminish the performance degradation of the traditional scheme and to compensate a naive TSAS in a low-overhead manner both in area and power. Although not yet verified by timing flow tracking tools, the optimized design provided decent isolation among concurrent tasks (sharing entities) by static virtual interfaces and fixed allocated memory. We also envision that with much wider granularity of high efficiency and concurrency this accelerator design can provide, different security favored pre-defined coordination can be more easily enforced.

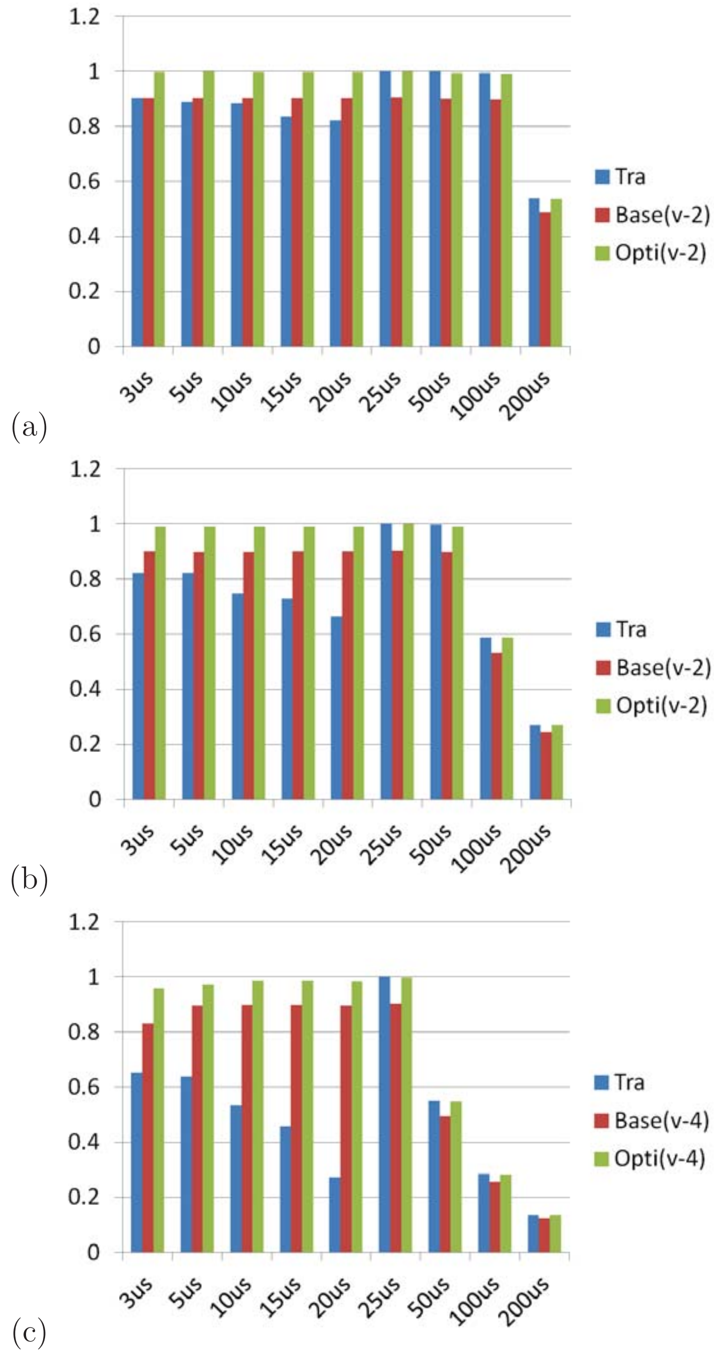


Figure 3.7: Relative performance under light (a), medium (b) and near-saturating (c) workload scenarios. V-2 and v-4 denotes the default maximum number of VMs allowed to concurrently occupy the device.

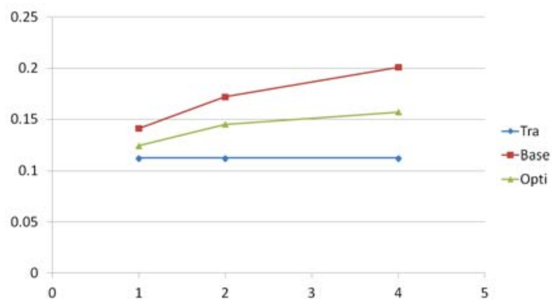


Figure 3.8: Comparison of area costs for v-1, v-2 and v-4 designs

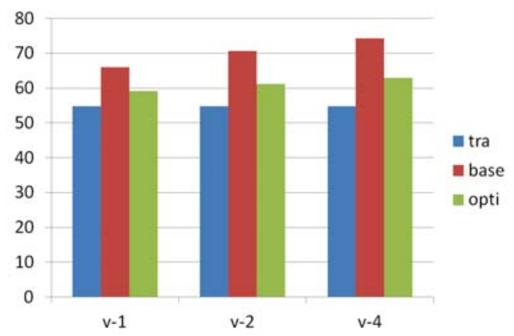


Figure 3.9: Comparison of peak power consumption for v-1, v-2 and v-4 designs

Chapter 4

Application-Centric Computation Concurrency

While the last two section focus on sharing that comes up in the context of cpu-to-cpu and accelerator-to-cpu communcation, we wanted to explore the full system stack up through to the operating system running on one cpu. Android is the most popular mobile operating system with the market share over 80 percent[3]. Its open source base and friendly libraries attract a large amount of contributions from the app development community. In 2012, the number of apps available in the market exceeded a million. These apps influence users' daily lives in communication, health, transportation, social networks, finance, entertainment, etc. Undoubtedly functionality, reliability and performance of these

apps and the underlying Android OS has crucial impact on users' life quality (daily experience). In 2016, with the release of Android Nougat featuring split screen for multiple foreground apps, the need of system support for application concurrency is unprecedented.

Within the many types of applications, human-computer interaction components including I/Os are often the focus of user experience enhancement. That usually includes audio and speech. For a user to truly manipulate multiple applications concurrently, speech is the optimal choice of input to advance beyond traditional keyboard typing. However, many successful speech interfaces Apple's Siri, Microsoft's Cortana are either OS specific or only targets system apps. Google also provides interface for voice searching [73], however, it has to be performed on a server. This becomes a major limitation of availability since mobile network connections are often slow or intermittent, and sometimes non-existant [53]. For a third party Android app to employ speech recognition feature, it is still inconvenient and inefficient. As speech recognition algorithms and functions are often computation exhaustive, currently there is no feasible way to support running multiple audio apps scalably in local environment. This fact is in direct conflict with the increasing concurrency need.

To resolve the conflict, one has to investigate deeper. By examining the speech recognition computations from different libraries, we find out that when applica-

tions use the same library, many of their speech recognition procedures adopt similar function calls and chains, meaning even from different apps, the computation paths might have considerable overlap. To take advantage of these computation overlaps, the computation paths should at some point have the same entry point (input). Yet another key observation is more exciting, that the speech computations are originally from the same device input (mic) and thus can have the same set of input data. We call these data seed data. Originating from seed data, identical subsequent computations can theoretically be shared among applications. The question is now how to share their computations safely and efficiently?

Fortunately on Android, there is one approach for applications to share – Inter-Component Calls (ICCs). An application can expose functionalities through APIs and intents. However, due to Android's appification [5] (decentralized) nature (each app has a Linux userID along with server client communication mechanism among apps) and arguably flawed permission model (permissions are mostly requested by app on installation and the components within an app share the same permission), there has been huge amount of various attacks [15][23] targeting these system design vulnerabilities. A large amount of research work has seen on security and privacy issues[96][94][24][31][88][27], yet the inter-app attacks have not been solved systematically. Meaning to simply share computations among applications on a pairing basis, is not only non-scalable (overwhelmed by ICC

communication overhead), but is also at great risk of compromised security and privacy.

We cannot help asking these questions: What are the security obstacles fundamentally introduced by the appification ecosystem layout (app isolation and decentralization)? Whether and how we can bypass these obstacles? In the meanwhile what are the feature and components in the Android OS that can positively influence the application sharing and concurrency?

To discover the obstacles hindering the progress of application concurrency and performance in Android, we analyzed the major security issues and categorized them based on the different system components involved.

Based on the study, we made the following discovery: The supposedly "performance promoting" functionality sharing mechanism through app APIs is both vulnerability-prone (suffer from privilege escalation attacks[15]) and sharing-unfriendly as not representing the server app's best interest in a lack of system procedure to confine the sharing parties in a fine-grain manner (the server app is not able to define the scope of sharing within specific apps). The mechanism carries the general decentralized system weaknesses but not the software-friendly business model. We realize the unfriendly model can be another challenge in promoting application sharing.

We further made the key observation that centralization is strongly lacking in Android where it is hurting both security and performance. Inspired by the PeerReview[36] concept in distributed systems community, we propose Cashmere – a central processing platform to guide interested apps in forming groups for sharing computation on libraries. Cashmere utilizes shared memory to cache library function call chains from concurrent running apps. By refraining the interactive paths among service-specified apps, seed app (the app feeds the original computation) side security concerns can be relieved, thus promoting confidence and willingness in sharing. The centralized scheme is well complementing the weakness of the decentralized Android base (in the pure decentralized scheme, an app is able to delay IPC communications, send incorrect results or even malicious calls/intents; On another note, pairing apps using IPC is not scalable). Thus the scheme boosts the client app side's confidence in sharing and can improve computation concurrency due to the provably increased number of sharing entities and the reduced/ramified IPC overhead. Note that without the confidence in sharing, an app might choose to compile its own local library and prohibit sharing radically. Thus Cashmere's support of fine-grained confinement is critical in promoting sharing and computation concurrency.

To summarize our contributions in this chapter:

- We analyze Android ecosystem structure and propose a central platform to support cooperative services to more securely improve application computation concurrency in the Android OS.
- We demonstrate its usefulness (performance wise) through a case study on a popular open-source speech recognition library.
- We initialize application-centric primary grouping to grant individual app's flexibility of action in determining security and performance trade-offs.
- We further solve the increased word error rate (WER) problem due to vocabulary dictionary model differences among speech recognition applications. We propose clustering-based sub-grouping on top of primary grouping, based on vocabulary similarities and overall WER friendliness.

The rest of the chapter is organized as follows. Section 4.2 provides background on Android system and discusses related work. Section 4.3 discusses the need for central access control and describes the procedure of speech recognition. Section 4.4 presents the architecture design of Cashmere – our application centric access concurrency platform. Section 4.5 details the prototype implementation of Cashmere and explains the sub-grouping algorithm. Section 4.6 evaluates Cashmere and discusses security impacts. Section 4.7 briefly describes future work and concluding marks.

4.1 Background on Android Access Model, Security Challenges and Inter-Application Sharing

Android's appification system has long been scolded for its lack of fail-safe defaults[5]. Although each app is represented by a unique Linux UserID, Android extends IPC to ICC where apps are able to communicate with each other beyond their own boxes. These ICCs are often the leak holes (through overt or covert channels) that break the boundary of inter-app data isolation and privacy protection that Android is aimed to achieve in the first place with its appification structure.

In the contrary, there has been attacks targeting different layers of Android system. RiskRanker[31] categorized apps threat severity to high-risk (root exploits), medium risk (privilege escalation) and low risk (non-critical data stealing) based on malware potential. High risk apps, although may cause severe damage and loss, of whom malicious components are obvious to detect. Medium risk apps might have the culprit since most of their attacks directly take advantage of Android's ICC mechanism rather than exploiting careless system bugs. Once succeeded, the malicious app can potentially gain dangerous permissions and sensitive data. The study also provided interesting findings that some of the reputable apps also

exhibit inappropriate behaviours or minor violations that make its benign title hard to judge.

We are inspired by this observation that we can further conjecture that these borderline apps may be categorized by other apps as dangerous or non-dangerous apps based on the judging apps' criteria. Currently Android only allows an app choose to share with either system apps or all apps[5], thus an individual app's interest and security requirement is not protected in a fine-grain manner during sharing. When an app clearly knows (by analyzing the market and its business models, offline checking tools and anti-virus software) the interested apps to share and certain apps to isolate, the system will not be able to grant such client app-specific access and communication paths. Meanwhile an app is provably not able to defend signature obfuscation techniques [11] without system intervention. It is worth mentioning that since current Android is strongly lacking support for third-party anti-virus apps/software [5], offline anti-virus check with reconfigured permissions (grant anti-virus app root privilege) can be a much more effective approach. To off-load this huge real-time/real-world scanning work on Google Play or other app market vendors (not to mention some of them are not reliable/benign) is impossible, it is more practical for an individual app development company to run the process within its interested sharing parties. We discuss this in detail in section 6.

On the optimistic side of Android's appification structure, Android is designed for apps to share functionalities. However, it is almost impossible to pair apps and share concurrently through inter-app ICCs in a scalable way. Based on the incredible growth in the number of apps, increased memory capacity on hardware, and the newest Android's split screen feature, we can only foresee that the demand/requirement for scalability will be significantly increased. Concurrent app sharing has been a rather new topic and there has been research work on the context analysis for coordinated scheduling on sensor devices [44]. Similar frameworks have been seen on a larger sensor-rich platform [52]. Although these works advance system support for sharing, there is almost no measure taken at the system level in preventing or mitigating certain vulnerabilities like covert channels [74]. Also none of these works extend to third-party libraries that have been heavily used, decentralized in nature and most attack-prone. Tackling with third-party library security and privacy issues has been among the most challenging in the Android security community [5] [76] due to Android's lack of privilege separation policy within an app.

Many tricky and unsolved security issues in Android should be the results of a lack of centralized system support, meanwhile the performance enhancing work is relying on system centralization. It is intuitive to suggest that in order to

improve application concurrency and performance without compromising security, wise system centralization mechanism is the key.

4.2 Design: Application Driven Access Control

Android system has been recently seen a shift to the dynamic and fine-grained permission control. It is already possible for developers to define custom permissions that can grant access to their app's functionality to other apps written by the same developer, system apps, or all apps installed on the device. However, although this mechanism improves the sharing realm among apps, it is still holding two factors that hinder the deeper and cleaner sharing.

First, applications require the same functionalities are more likely to be in direct competition. It is misleading and naive to assume that an app is willing to share features or computing results to all other installed apps (basically all apps). To preserve this competition edge, an app may choose to share only among its sister apps (written by the same developer or company) rather than sharing towards a much unbounded base of apps. In this case, the benefit of sharing is much more limited. Even if an app wants to share to a grander but limited group of apps beyond its own relatives, there currently exists no means for a third-party app to identify another app's true identity to enforce the desired permission grant.

Second, from the receiver client app point of view, the service given by other third party apps is ultimately decentralized - that unavoidably carries the (undeniable) security defect - malicious intentions including not conforming to the communication protocol or aiming at privilege escalation attacks [23] through interfaces/IPC (it is not easy to avoid these attacks since all components in an app enjoys the same permission), or providing incorrect results or unwanted data (tamper). Note that current Android OS does not provide central witness or regulating.

To promote clean, safe and at-will sharing among applications, we propose signature-based grouping (permission control). In this section, we discuss system implications and opportunities in the case study of audio/ speech recognition (SR) applications and libraries.

4.2.1 Background on Android Audio Applications

Audio recording and processing is among the most primary functions of a mobile device. With the growing popularity for user voice control, speech recognition applications have been developed and improved for fast computing needs and low latency. Typical audio applications read audio files or streams from the audio service of the device and use libraries of speech recognition algorithms to decode and extract important information about the content.

Several technical challenges have hindered the deployment of such applications on mobile devices. The most difficult of these is the computational requirements of continuous speech recognition for a medium to large vocabulary scenario. The need to minimize the size and power consumption for these devices leads to compromises in their hardware and operating system software that further restrict their capabilities below what one might assume from their raw CPU speed [41]. Moreover, memory, storage capacity and bandwidth on mobile devices are also very limited.

4.2.2 Speech Recognition Libraries

Automatic speech recognition (ASR) is broadly defined as the translation of spoken words, or an acoustic waveform containing speech, into a string of words [59]. In modern ASR speech is modelled as a mixture of acoustic and language properties [9]. The acoustic models attempt to move from audio samples of speech to potential phonemes being spoken, and from these phonemes to possible words.

The language model provides the probability of a specific sequence of words occurring given a particular form of speech such as news, lectures or conversation. This is used in conjunction with the output by the acoustic model to identify the spoken phrases which are most likely to be correct [59]. Constraining the vocabulary of the model is used to increase the identification rate for systems

like telephone interfaces and controlling subsystems in a car [48]. Larger, yet still constrained, models are used for tasks such as Internet search and calendar management [78]. The advantages of such a system are that with a tightly defined language model recognition rates can be greatly increased. Constraining the vocabulary can improve the accuracy between 50% and 80% [14].

Platform speed directly affected our choice of a speech recognition system for our work. The SPHINX speech recognizer of CMU [51] provides the acoustic as well as the language models used for recognition. It is based on the Hidden Markov Models (HMM). Though all the members of the SPHINX recognizer family have well-developed programming interfaces, and are actively used by researchers in fields such as spoken dialog systems and computer-assisted learning, we chose the PocketSphinx [41] as our speech decoder which is particularly meant for embedded platforms. It is a version of open-source Sphinx2 speech recognizer which is faster than any other SR system.

4.3 Overview of Cashmere Architecture

We present a software platform design called Cashmere. Cashmere enables Android system to concurrently run multiple audio/speech applications with improved overall system efficiency, reducing computational overhead and major malicious concerns. Cashmere's key innovation in providing this central layer be-

tween applications and computation libraries, is of significance in transferring vulnerability-prone app-to-app communications into safer app-to-system communications.

4.3.1 Library Paths Identification

For continuous audio applications, the microphone service streams audio frames from the mic to the application as they become available. This stream becomes the seed data for repetitively computed audio processing operations within an application. Initial-level audio processing operations, such as configuration initialization and sampling, create derivative objects from the seed data. Other processing operations then generate successive derivatives when decode sample data, mark progress, get hypothesis or more feature-based operations.

Since the computing process basically is a library function call chain, we call these call chains library paths. Comparing a set of audio and SR applications, we observe the initial-level library paths are usually the same. This key observation promotes the first steps in computation sharing and concurrency. Since each application's library path can only differ from another application at the beginning of a library call, we define these bifurcations as branch points.

To provide structured sharing of paths, Cashmere exerts a layer between a library call and the actual target library. This layer is a central platform that

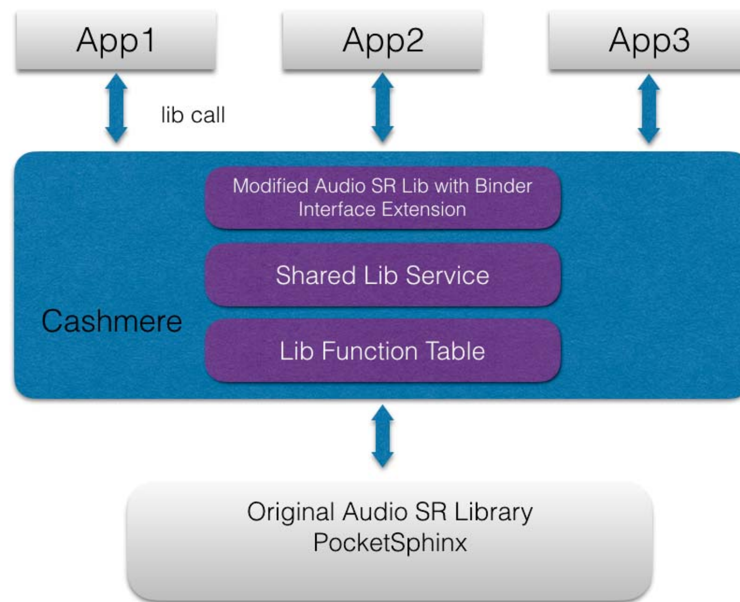


Figure 4.1: The Cashmere Platform Architecture. Upon an application calling a lib function, the call will be directed to the modified audio library extended with binder IPC client interface to send lib call requests to Shared Lib Service. Shared Lib Service is a registered service served as a transition channel between client apps and the Cashmere server application. Library function table is the major component of the server application, it records previously computed lib calls and will reply to future identical calls directly, only un-computed fresh lib calls will be passed on to the original library. Note that the original audio SR lib, in our case libpocketsphinx.so will be renamed so applications cannot directly call it upon and instead the modified lib is named libpocketsphinx.so.

receives library call requests from all running applications and dispatches according function returns. Cashmere matches library calls with identical parameters, reusing computation results to reduce computational redundancy. Cashmere and applications establish server client relations. Leveraged on Android's binder mechanism, a library call will be directed to Cashmere through binder IPC. The according replies will also feed through binder handles. The architecture of Cashmere is illustrated in figure 4.1.

Cashmere itself is designed as a registered binder service within Android Platform but outside of Android kernel space. Figure 4.2 shows Cashmere's relations to the Android system.

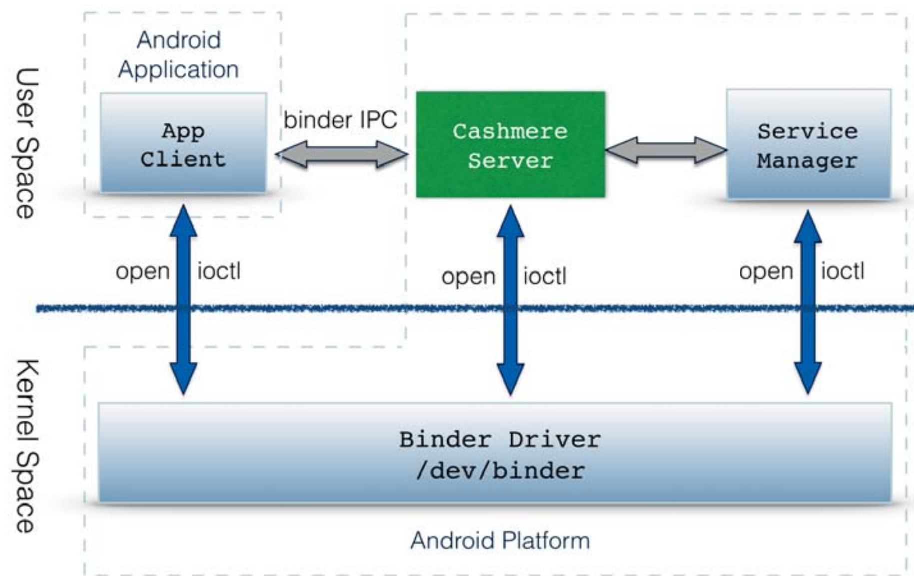


Figure 4.2: Cashmere as a registered service through Service Manager provided by Android platform. Cashmere is in Android user space.

4.3.2 Computation Memorization and Management

Although automatic memorization [60] is not a new topic, there is still difficulty and room for optimization. For instance, only pure functions, those whose output is determined solely by their input arguments, can be memorized.

Besides, in our scenario, we also need to enable general access to the memorized library call computations from different apps. To allow this, an efficient common memory region is required.

We design a hash table to store the functions and computations. To correctly maintain the matching table, procedures of write to and read from the shared region must be carefully designed. Since there is no need to use a sorted map while *unordered_map* in C++ does not allow complex key format such as pair or vector, we assign the function name as the entry key, while storing the value as a vector of pairs with each pair containing a Boolean variable *ready* (set true if the computation of the entry is completed) and a pair containing a vector of input parameters and a vector of the according output parameters.

Upon receiving a function call, look up is performed by comparing the function name first, then compare the input parameters of the function against each pair in the value entry. If a match is found, a read will be issued by simply returning the output of the entry if the Boolean variable *ready* is true.

Otherwise a new pair with the input parameters will be appended to the vector (*ready* marked false), and the function call is forwarded to the original library. Once the execution is finished, the output part of the pair will be updated from the function computation and *ready* set true. Note that the above only works for flat function calls (whose parameters do not contain returns from other functions). To allow memorization for nested calls, we introduce void pointers to represent all parameter vectors. Such that a function return can be equally seen as normal parameters by retrieving the region the pointer points to.

4.3.3 Application-Guided Grouping

Applications tend to be professionals in sharing users' privacy, however, they are not able to use their expertise where they can more wisely and ethically collaborate - I/O devices and computation results. Our study of tracing general interactions among the most used mobile applications has found out that resource contentions have been a major kill in performance and energy-efficiency.

We propose application-guided grouping, supervised by Cashmere within Android OS. The central platform is equipped with the partition ability based on each application's collaboration interest by initializing and updating group matching table entries during an application's first invocation of the library. Note that

apps can be in the same group only when in direct or indirect mutual agreement (trust), the details will be discussed in section 5.

4.3.4 Concurrent I/O

Conventional sharing is discussed on the basis of context switch, while in our terms we define it as an altruistic behavior. A lot of work has been discussing improving concurrent I/O application performance [39][42][13]. However, their focus is on paralleling the I/O access within a single application while do not see the missed opportunity of more concurrency among multiple applications.

Our intuition is that if improving parallelism and concurrency within a single application can improve its performance, it will be similarly beneficial to the overall system performance if we improve the concurrency by solving resource contention across application boundaries. Although the idea seems promising, it exerts a difficult problem as secure isolation standards among applications are usually significantly higher than within one application itself (among its components). The key insight behind our proposed technique is that the vast majority of applications are not malicious (The signature signing party performs checks and Google Play as well normally will remove malicious apps within a short period of time) and they can be scanned and reversed engineered offline for anomaly detection. Also considering there has been established trust among many appli-

cations for functionality sharing through APIs, it is reasonable to form groups for mutually trusted applications, thus extending the per-application boundary to group-based boundary.

One could argue that such mechanism will create unfairness by benefiting more to the applications in a large group and create disadvantage towards more isolated applications. However, the evaluation results indicate quite the contrary. Even if there is unfairness in computations without system intervention, we can solve this by strategically promoting the groups containing foreground running app(s) (be it multi-app group or single app-group), that is, for inter-group applications, we use non-cooperative method, the groups of applications that seem more important to the user will be promoted. Thus it is a reasonable strategy because it is user-guided and user-friendly. Note that the absolute fairness perceived by the system may not be demanded by users.

Also if an application is isolated, even if it cannot benefit from obvious performance gains, it may either purposely chooses to sacrifice performance advantage for security reasons or be offered such luxury of lower leakage risks without compromising performance.

Note that once the group formation is determined, it will be memorized by the system. No further negotiations will be needed among applications in the same group should the user starts running a subset of the same group again.

To illustrate of the usefulness of concurrency through cooperation, consider a scenario where a user wants to voice order food through restaurant apps and share this information during audio chatting with friends. He is probably running several restaurant software in the front while he wants to share these apps' information in real-time to his friends and family who by the chance are using three different apps to communicate with him (which isn't rare considering the amount of different social network apps we have). Rather than rotating the usage of the CPU and audio I/O among these software, if the applications are in the same group, they can share I/O data and subsequent computations. This greatly eliminates context switch cost and computing resources, and both the front and background software can get real-time audio information constantly (responsiveness is important in communications). This greatly increases the convenience to the background apps that are usually less favored and thus only get occasional audio data or even no data in the original Android environment.

4.4 Implementation

4.4.1 Two-Phase Grouping

Out of both security and performance concerns, we design a two-phase grouping. Phase One defines the primary boundaries among non-trusted parties. We

require each application to provide grouping information, e.g. the list of trusted application IDs. Since every Android app has a unique application ID, this ID can uniquely identify the app on the device and in Google Play. However, considering the large base of application collections, we also provide options for only specifying non-trusted applications, or delegating grouping obligation to a subset of other applications (meaning the app will use the same list as its representative) or simply use system default. Note that based on the fact that most malicious apps in Google Play can be cleaned up in a short period [31], we can design system default based on popular anomaly detection tool (RiskRanker [31]) and the existence length of an app (the longer it has been in Google Play and since its last update, the less the risk.). In this paper, we will not extend the discussion on anomaly detection and system default.

Phase Two further divides group members based on vocabulary dictionary similarities. As we mentioned, in speech recognition services, constraining the vocabulary of the model is used to increase the identification rate for systems. Thus one application's vocabulary corpus can be dramatically different from one another and using uniform speech models will kill precision and accuracy, and might incur significant delays due to the potentially enlarged model. However, if each application uses their own models there will be limited room for computation sharing. It is crucial for Cashmere system to define the boundaries of similarity,

and to rebuild models for each subgroup. Note that all members within the same group still share the audio record resources and pre-processing procedures regardless of subgroup boundaries.

To accurately categorize subgroup interests, we use information retrieval techniques to rank the closeness to a subgroup should a new app be added to the group. The primary information we use is the text corpus collection of both the subgroups and the new member app. During our evaluation, we find out that the word error rate is beyond linear to the size of the dictionary. It suggests that groups with smaller vocabulary should benefit more from grouping by sacrificing less in accuracy. The indication creates very subtle trade-offs in grouping since it basically implies that when looking at the overall platform performance and accuracy, it is more advantageous to add the new app into a group with smaller vocabulary size than into the ones with larger sizes when the vocabulary increase caused by the new app is similar in both cases.

Additionally, since the number of applications and groups are not fixed from the beginning of sub-grouping and is primarily restrained by the platform computation capacity, the straight forward k-means clustering algorithm [49] of simply repeatedly computing new centroids of k groups will not work properly and will be very inefficient. Especially in the scenario when only a small number of applications needs to run concurrently, the expansion to a comparatively large number k

is unnecessary. On the other hand, although increasing k (the number of groups) may be friendly towards recognition accuracy, each increase of a new group might result in significantly increased latency as well.

Due to the dynamic nature of k in our case, we have to adapt the clustering algorithm based on the platform computation performance we collect and thus determine the maximum number of groups allowed and the appropriate point of regrouping or generating new groups. The subgrouping algorithm is illustrated in Algorithm 1.

The present standard index for ASR system assessment is WER, which is defined as the proportion of word errors to words processed. Let N, S, D and I denote the total number of reference words, substitutions, deletions and insertions (see Equation.1). In Connected Speech Recognition, WER is defined [12] as

$$WER = \frac{S + D + I}{N} \tag{4.1}$$

We compute the overall temporarily increased WER upon app joining for each sub group, so the lowest of all will be selected as the candidate group. We use the temporarily updated WER of the candidate group to guide regrouping. If the updated WER is below the default WER limit, then the candidate group is accepted for the app to join. If the WER is above the limit but the maximum number of subgroups is below the count limit, we can rely on k-means clustering, in

the hope that total regrouping will help decreasing overall WER given k (subgroup count) is increased by one.

We compare the corpus file of the new app to the corpus file of each sub group. Upon joining the subgroup, the subgroup's corpus file will be updated combining the new app's corpus information. The updated corpus file will then be fed into Sphinx lmtool [70] to generate a new dictionary model accordingly. For this reason, we require that each app provides their original corpus file along with the app, either locally or though web access (remained as future work).

4.4.2 Inner-Group Isolation

Due to performance and user interests concerns, we assign the foreground app as the leader app of the group. The leader app will always have its own extra copy of data inaccessible by other members, thus that even a malicious member will not be able to manipulate the leader's original data. And if there exists a malicious app modifying the shared data for damage purposes, the group always has the original copy to reset with.

4.4.3 Inter-Group Isolation

We are very conservative in designing inter-group isolation for the purpose of keeping data leak risk at the lowest among applications that have no foundation of

Algorithm 1: Sub-grouping algorithm.

Given: Groups G, WER(dictSize), App A

foreach $g \in G$ **do**

  g.tmpWER $\leftarrow$ WER(combineDict(g, A))

  g.ΔWER $\leftarrow$ g.numOfApps $\times$ (g.tmpWER - g.WER) + g.tmpWER - A.WER

end

sort g $\in$ G with g.ΔWER in ascending order;

foreach $g \in G$ **do**

  if *g.tmpWER* $\leq$ *WERLimit* **then**

    return g

  end

end

if *G.cnt ¡ groupCntLimit* **then**

  run KMeansClustering(G.cnt+1, G, A)

else

  return first g $\in$ G

end

trust. There is no sharing of computing results nor even the seed data. Since Android adopts Linux's UID permission system and there has been extensive research in refraining or sandboxing untrusted third- party applications' aggressive/sneaky access to other apps or the user's data [95][93], we assume applications of different groups have no knowledge or execution power of other groups' data given there is no direct inter group communications and the OS is not corrupted.

In our Cashmere source code, we initialize separate function lookup tables to the exact amount of maximum groups allowed rather than initializing a single big table using the groupID as the first-layer entry key.

Since the maximum concurrent ASR apps in normal scenario is below hundreds on a device, for safety concerns we design separate vectors storing appIDs for each group rather than using the appID as a hash key. Upon receiving a function call request, the appID will be compared against each element in each vector of the groups. If there is a match, then the app is allowed access to the memorized function layer where the group's data is stored. The search time is still constant due to the minimum number of apps.

If there exists no groupID for the app, Cashmere starts group forming based on the grouping information the app provides and the group's collaboration preference. Thus there is strong isolation among different group's data and computation.

4.5 Evaluation

We evaluate Cashmere implementation in an Android emulator with instance setting as Lollipop version 5.4, hardware Nexus 5x hexa-core Cortex-A57. In addition to our case study utility, we design a set of micro benchmark apps to characterize the actual performance of our platform. As we focus on the performance side among applications, we extensively examine the varied latencies and show how our approach bypass the bottlenecks. Note that since Lollipop does not support split screen (multiple foreground apps), we design most benchmark apps as background services in order to create and characterize concurrent running scenarios.

Our evaluation addresses the following research questions:

RQ1 How does Cashmere compare to commercial mobile environment in terms of latency in concurrent application computation?

RQ2 How do various optimizations and groupings affect performance and security of applications running on Cashmere?

RQ3 Can sub-grouping lower recognition WER in the presence of multiple speech applications of different functionalities and to what degree?

4.5.1 Library Call Indirection Overhead

We microbenchmark a major ps_decode() library call on a one sentence audio record using the original vocabulary model. The computationally intensive decode library call takes 1.71 s, exhibiting the potential benefits of caching expensive library calls. We do a single-call analysis on our microbenchmarks by averaging 10 instances of library calls. Cashmere introduces a minimal overhead (memory allocation and redirection) of as much as 12.8 ms per call, yielding overhead to 0.75% for expensive calls.

4.5.2 Dictionary Model Influence in Latency and Accuracy

We also proceed the experiment with a more constrained dictionary model (with the minimum vocabulary corpus needed for the scenario of 10 words), and it takes 350 ms. It shows the drastic impacts different vocabulary models can exert. However, even in such low latency calls, Cashmere only adds 3.7% overhead. Note that the recognition accuracy is 100% under this extremely tailored scenario. To characterize the the relation of WER to the vocabulary dictionary size, we increase the dictionary size on varying granularities and repeat the experiment. Figure 4.3 and 4.4 show the relationship of WER to dictionary size and latency to dictionary

size respectively. WER increases nearly linearly (less than exponential) to the word counts in a vocabulary dictionary. When the word count is smaller than 20000, WER is below 2%, which is highly acceptable. However, WER tends to increase faster when word count exceeds 80000. The reason behind is that the more word counts, the less difference between recognition candidates and the more error-prone combinations of words. In contrast, average latency is nearly linear to word count while scales even better. This is due to the audio pre-processing time and other application system communication cost.

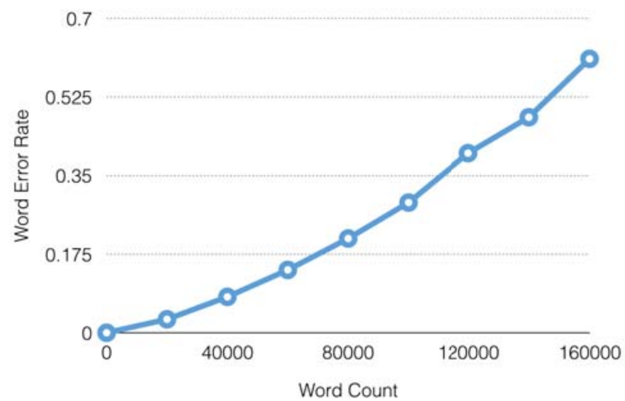


Figure 4.3: Average WER to varied vocabulary dictionary sizes on PocketSphinx.

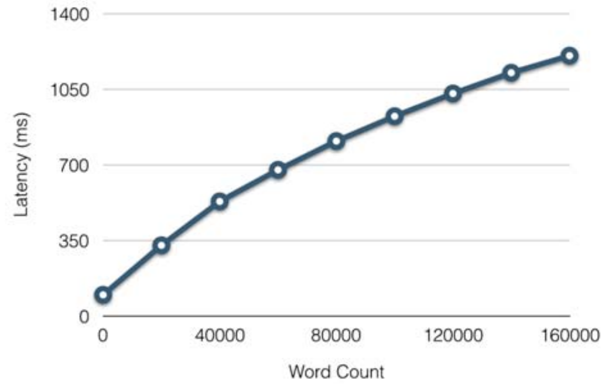


Figure 4.4: Average latency to varied vocabulary dictionary sizes.

4.5.3 Concurrent Apps Performance

We next analyze the ability of Cashmere to service concurrent Apps. We employ fundamental audio tasks as benchmarks to examine the benefits and overhead of Cashmere on Apps.

We use audio tasks of Voice Recording and Speech Recognition. Apps employ these audio tasks to perform a diverse set of duties. These tasks comprehensively cover the PocketSphinx library, and serve as a basis for many audio applications that can potentially run concurrently. In our evaluation, each of our apps receives the audio stream through the Cashmere platform distribution. An app sends the stream through each benchmark audio task, which begins by recording, and read it to samples. We include these pre-processing steps in our evaluation results. Speech recognition algorithms decode and identify speech contents in a

Table 4.1: Applications with Dictionary Word Counts

Applications	
Application	DictWordCount (Average)
Restaurant	216
Airline	21350
Mini Health Assistant	1158
Advanced Health Assistant	69812
Direction	370
Search	31756
E-Commerce	14620

voice record, useful for applications such as social apps, gaming and vehicle GPS. Speech recognition consists of two components: decode and classification. The Vertibi Search algorithm compares a computed distance against a cascade of pre-trained classifiers. Speech classification identifies contents by matching against a vocabulary dictionary and voice dataset. The nearest neighbor is determined to be the content word.

Our benchmarks consist of two combinations of apps running audio tasks. In a potential scenario, a mobile device would run simultaneous background applications running multiple audio tasks, e.g., logging social interactions with speech recognition, recording speeches, directing locations. We selected and coded a set of speech recognition tasks with a variety of duties (as shown in Table 4.1). The vocabulary of each task is formed with either the guidance of the widely referred speech dataset website Linguistic Data Consortium [66] and VoxForge [58] or open sourced software.

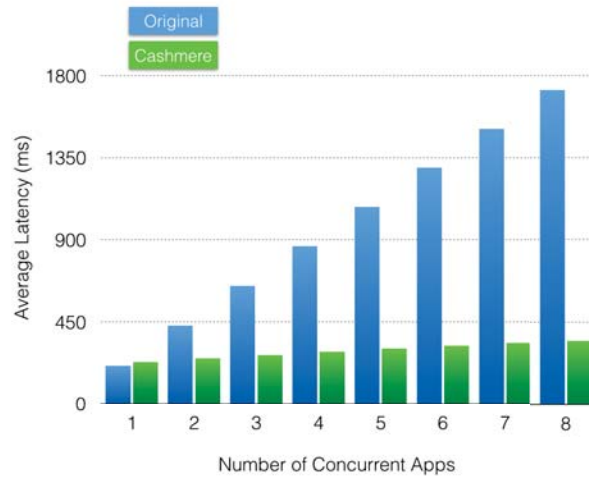


Figure 4.5: Average latency of concurrently running audio apps natively and in Cashmere.

As as can be seen from Figure 4.5, Cashmeres efficiency benefits become apparent when running multiple identical speech recognition audio tasks. While native processing time is doubled when running two tasks on the original platform, Cashmere manages to maintain the increased processing time within 19.0% of the original frame processing time. This reduces the per-frame processing time of running two audio task instances by 41.9%. The advantage is even more substantial when running more than 5 instances where the latency of the original platform has strongly exceeded the limit of human tolerable latency of 1s [62] while Cashmere stands firmly within. Combinations of applications with different categories also benefit from Cashmere efficiency. Through sharing the decode() library call on the input frame, Cashmere decreases the processing time of the combination of voice recording and speech recognition by 11.2%.

Thus, Cashmere provides app scalability, as many apps can run with minimally impacted frame rate performance.

4.5.4 Grouping Overhead and Impact

Grouping happens exactly the very first time an app calls the library. We measure grouping overhead by averaging 2 to 15 apps grouping overhead. The overhead is 0.033 ms, as an one time overhead per app joining, is minimum compared to speech processing and recognition time.

Impact of Primary-grouping on Timing Channels

We consider the scenario when an app requires more security protection (e.g. less induced timing channel), so it chooses to form a group owned only by itself. From section 6.3 we know that if there are concurrently running groups (apps), the latency will slow down for each concurrent entity. However, what kind of apps (defined and mostly influenced by dictionary size) will be influenced to what degree under different group formations is not clear. Since timing channels can be quantified by latency variations [10], we run apps with small to medium (300 - 20000) and large (80000) dictionary models concurrently with additional 1 to 3 groups and compare the latency with its original latency. We use additional

groups with average medium to large vocabularies considering the normal group formation. To verify the effectiveness of using primary grouping to mitigate timing channels, we also compare the above results with the latency in the scenario when the app simply joins another group.

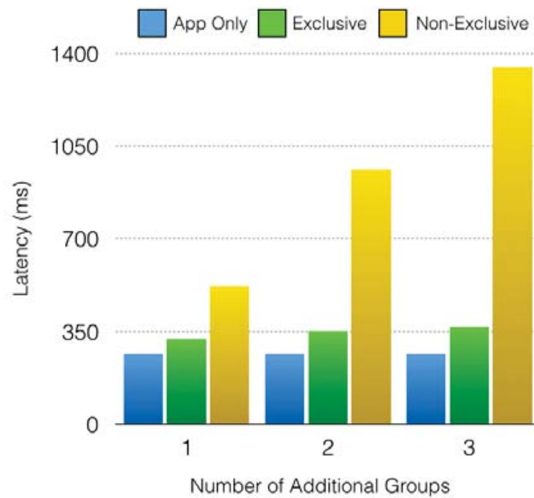


Figure 4.6: Average latency of running an app with small to medium vocabulary sizes without concurrent apps, running exclusively but concurrently with additional groups, and running non-exclusively but concurrently by joining one of the additional groups in Cashmere.

From Figure 4.6 we can see that apps with small and medium dictionary sizes are only influenced minimally when running exclusively. In contrast, without grouping, their latency increases significantly due to usually dramatically increased combined dictionary size of a group. That means that by running exclusively, the timing channel created by differences in latency is significantly alleviated. Apps with large dictionary sizes (Figure 4.7) seems not benefit by running

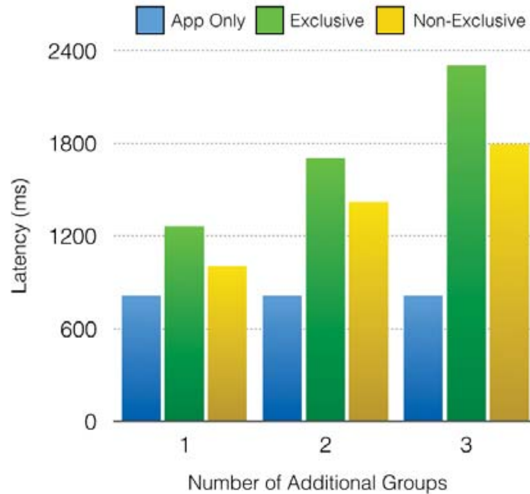


Figure 4.7: Average latency of running an app with large vocabulary sizes without concurrent apps, running exclusively but concurrently with additional groups, and running non-exclusively but concurrently by joining one of the additional groups in Cashmere.

exclusively from the timing channel alleviation perspective. However, one reason is that we run concurrent groups with medium to large vocabularies, which is rather conservative. One can imagine that if we use much larger dictionaries (above 100000, which is not unusual) for the additional groups, the exclusively running app can benefit as well similar to the first small/medium app scenario.

Also note that although timing channels are one of the major concerns in security issues, an app can potentially benefit from forming an exclusive group by owning its own computations and memories. Usually it is less likely to be corrupted by techniques like stack overflow that might target the library's own vulnerabilities. Note that the fact that apps with small and medium vocabulary

sizes can benefit well with security is sufficiently useful since security-sensitive apps are normally word restrictive, e.g. financial apps. Besides, from a performance perspective, adding a few new exclusive groups with small/medium dictionaries has only minimum to moderate impact on overall system performance. On the other hand, since it is relatively performance unfriendly for an app with a large dictionary to start a new exclusive group, if not for security reasons, an app should be inclined towards joining a current group. Also note that the difference between an app running exclusively and simply running without Cashmere with its own compiled library is the flexibility of forming its own group, and it will be beneficial with future potential collaborators. The evaluation is to show the trade-off in forming a new group beyond the scenario of simply exclusively running.

On another note, we experimentally confirmed that an app requiring primary grouping for its own partition will not change its WER from original. This supports the flexibility of action when an app determines performance, security and WER trade-offs.

Sub-grouping Performance

In order to characterize the influence of sub-grouping based on vocabulary similarities. We conduct three combinations of different scenarios. The first scenario

is for restaurant apps with voice commands. We select major inter chain pizza restaurants and port their menu into sentence corpus respectively. The corpus files are then extracted to generate dictionary models accordingly. Using prerecorded audio files, the original latency were measured as 207 ms. To evaluate the efficiency of sub-grouping, we run the four apps concurrently on Cashmere. Cashmere will extract the corpus files and generate a model covering all the vocabularies of the four and thus the model can be shared. The latency of concurrently running four apps is shown in Figure 4.8. The second scenario is targeting more profound speech recognition usage (personal assistant, searching, etc) that requires a significantly broader vocabulary, we use both the minimum and advanced health assistant applications. This scenario also represents the sub-grouping formation of apps with similar functionality but significantly different vocabulary sizes. The third scenario is the combination of all the apps and their sister apps listed in Table 4.1.

We test if Cashmere can perform desired sub-grouping and the latency is listed in Figure 4.8. For restaurant apps in scenario 1, the dictionary of each app is very small and has major overlaps. Simply to divide the apps into more groups will only create new computations that sharply increases latency without noticeable benefit from nearly identical separate dictionaries. For health assistant apps in scenario 2, since the category contains both minimum and advanced dictionary

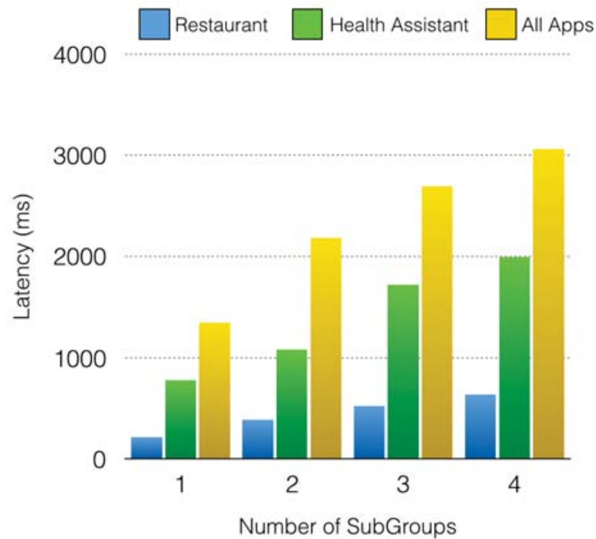


Figure 4.8: Average latency of concurrently running different combinations of apps in Cashmere.

models, when sub-grouping happens, the minimum apps can form a new group that only adds a fraction of burden to the original computation. The third scenario reveals the fact that latency scale better than linear throughout different sub group formations for even very diverse apps. The reason is that the divided dictionaries tend to be significantly smaller than combined, the more sub groups formed, the more sharply decreased average dictionary sizes. Although our sub-grouping algorithm targets WER, it directly boosts intelligent dividing of dictionaries as well. We also stress test the limit of the number and formation of apps that Cashmere can sustain. It turns out that Cashmere can support 15 concurrent apps with different functionalities and dictionary models, which is far beyond the capa-

bility of the original Android commercial environment without relying on a server.

How does sub-grouping improve recognition accuracy

Since the main reason of sub-grouping is to increase accuracy of the recognizer. We evaluate the average WER for each category of apps listed in Table 1 with sub-grouping formation in the third scenario. Note that number of subgroup equaling to 1 represents no sub-grouping. We find WER decreases significantly when proper formation of sub-grouping is allowed as shown in Figure 4.9. We adjust parameters in the sub-grouping algorithms to allow different sub-grouping formations. As seen from the results, without sub-grouping (number of sub groups is 1), all apps have to share the full combination of all vocabularies, which is mounted to approximately 156,000, yielding very bad WER for all apps. When two sub groups are formed, the largest dict category – advanced health assistants is excluded to a new group. Thus the health assistant apps and the rest both have cut-to-half dict sizes, which leads to dramatically decreased WER. Similarly, when four sub groups are formed, the formation is nearly ideal since the resulted WER for each app is very close to the original WER for each app without sharing dictionaries with other apps. Notably, sub-grouping is very efficient in lowering WER (im-

proving recognition accuracy) and is essential in compensating the accuracy loss during computation sharing.

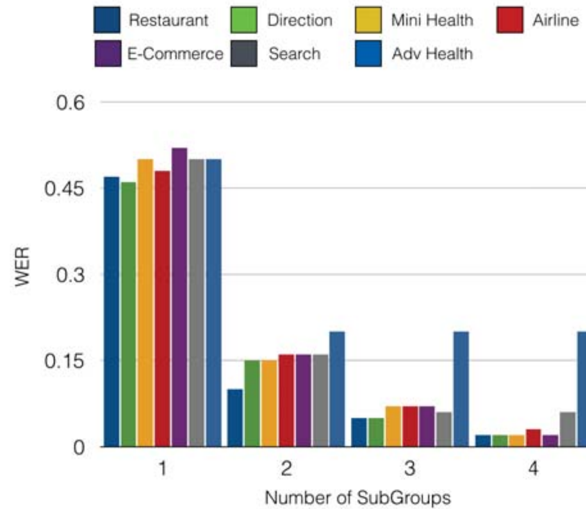


Figure 4.9: Average WER to varied vocabulary dictionary sizes.

4.5.5 Security Analysis

Our system can ensure individual application's library grouping privacy policy enforcement on Android device through dynamic binding.

Malicious apps. Cashmere allows device users to install their favorite apps on their Android smartphones. Some apps may be malicious and target at compromising our policy enforcement mechanism. However, since the user-level malicious processes are securely isolated into separate group containers, they cannot manipulate the code or the control flow of cashmere unless they have the root privilege.

We assume the Android OS can be trusted. Therefore, without the root privilege, malicious apps cannot compromise our mechanism.

Permission escalation attacks. Android system may suffer from permission escalation attacks, such as confused deputy attack and collusion attack[15][25]. In confused deputy attack, a malicious application exploits the vulnerable interfaces of another privileged (but confused) application to perform unauthorized operations. This kind of attack usually happens when a privileged app unintentionally exposes interfaces of sensitive operation to an app without required permissions. We have considered this attack during the design of guideline of grouping - when selecting grouping members, it is recommended to check candidate member's interface as strict as one's own interface to allow collaborations, this interface check can be achieved using reverse engineering and normal anti-virus software, such as AVG Antivirus Free, Lookout Security Antivirus, Norton Mobile Security Lite, etc. Since we design grouping to be a two-way agreement, it's reasonable to anticipate that the interface security of a group is not dramatically different from individual member apps alone. Meaning, even without grouping, the individual app's interface security is likely to be similar to its group's lowest.

In collusion attacks, malicious apps collude and combine their permissions in order to perform actions beyond their individual privileges.

Our platform defends this kind of attack through fine-grained low-level control. Permission is not able to be combined simply by primary grouping since our mechanism does not change permission in setgroups in Process Creation Request directly, thus grouping is on library service level rather than system level like (e.g. an app without 1007 CAMERA permission is not allowed to access camera even it is in a group where a group member is actively accessing camera). We carefully design the control level to as low level as possible to not to be confused with system permission.

4.6 Conclusions

Application concurrency has become the primary need of a mobile system. However, I/O and sensor related computation is usually expensive and is in direct conflict with the limited computation resources. Specifically on Android, its decentralized nature is another obstacle from applications' safe sharing of computation. By exploring the possibility of centralization, our proposed application-centric platform Cashmere is able to efficiently isolate applications based on each application's interests and saves computation by memorizing library calls within the group boundary. In dealing with applications using computation exhaustive speech recognition libraries, subgrouping (grouping based on vocabulary similarities) is further useful in improving and balancing latency and accuracy. The

concept can potentially be extended to other I/O areas. Although in this paper we assume PocketSphinx a benign library, the collaborate primary grouping approach proposed in the paper is promising in solving the third party library sharing problem that is widely considered one of the most challenging in the Android security community. Future investigations into how to radically eliminate timing channels among groups is also a valuable topic.

Chapter 5

Conclusions

We conclude this dissertation by summarizing our key contributions and discussing the utility and trade-offs in hardware and software sharing patterns.

Sharing for efficiency is one of the most common tricks in the computing system designers "handbook". Sharing through a traditional strict time division (e.g. via predetermined context switches) can be almost universally applied when multiple entities are supposed to utilize common resources. However, such strict time division is subject to significant performance degradation.

As noted by Menychtas et al. [61] the goals of protection, efficiency, and fairness can often be in direct conflict. Indeed, as more freedom is introduced into the schedule, more issues of protection arise. Timing channels are inevitably introduced because of contention on resources. These timing differences can introduce

both opportunities for denial-of-service attacks and more subtle leakage such as those that target cache and crypto-graphic engines to retrieve secret data. Under static sharing, e.g. using pre-defined static scheduling algorithms, the timing usage of any entity is strictly bounded and unaffected by other parties, thus eliminating timing channels. The difficulty is that there is no universal static schemes one can apply to any component that also provide performances. As such they require a deeper examination of the particular component and often demand optimizations to avoid other overhead (additional area, etc).

Across the three different systems examined, the network-on-chip, the accelerators, and the operating systems sharing scheme, we find different sharing patterns with a common theme. While schemes closer to the hardware can be more carefully scheduled to hide latency it is difficult to carry those ideas up the software stack where timing is less easy to control. However, careful static coordination of the sharing parties is still useful. At the software level this can instead manifest as collaboration groups.

5.1 Contributions

For hardware level sharing, the scalability increase usually comes from the hardware architecture design itself and has little to do with the properties of the concurrent tasks running on them. From the lessons of our successful design of

SurfNoC, we learn that to be able to defend timing channels, static schemes might have far less overhead than previously believed when latency is taken as a first class design constraint. In the case of hardware accelerators the key insight is to allow flexibility in being static or approaching static. Flexibility is important since security has to be balanced with performance. Traditional methods of sharing on accelerators are through full context switches, and for simplicity, one often chooses to drop tasks or wait for an unfinished task to finish. The former is detrimental to performance if fine-grained sharing is required, and the latter is subject to more timing leaks. By chopping states at the optimal points along a computation, and by providing a minimum interface to the software, an accelerator architecture can be granted the capability to locally determine the best time to switch. Thus a pre-defined switching frequency, that shields timing leaks, is unlikely to incur more than the minimum performance overhead. Although the verification and quantification of actual timing information leaks are not evaluated in this thesis, we can conclude from our current characterizations that flexibility of sharing is guaranteed and that the design is at least as secure has having multiple interfaces. By comparing to the verified timing channel-free SurfNoC, we know that if the number of sharing domains is pre-determined, then the waiting time and latency can be fixed. This can also be applied to the sharing scheme proposed in this

thesis, and because of its low context switch overhead, it can endure as many domains as the system requires, which is unlikely with traditional context switch.

Building from this success we were also very curious to find out if the patterns can be applied in scaling software as well. However, due to the more dynamic nature of software systems, applying static scheduling when the number and properties of entities are unknown is unwise. We were inspired by the cooperative distributed systems algorithm and thought that the spirit of that work could be extended to elsewhere in systems. We also wanted to push the work toward a more commodity application space so, in choosing a software operating system to work with, we target one that requires concurrency improvement and is under difficult security concerns, which makes Android a perfect candidate.

Fortunately, Android is already on a path that leads toward application collaboration, however, until now, it is still deeply trapped in numerous different attacks and privacy leaks from malicious applications. Even anti-virus software (third-party apps) have no privilege on Android which makes them essentially useless. Flow-tracking software can consume a significant amount of energy and is simply not realistic on many devices. In such a vulnerable state, any software applications that suffer from attack once or more will reasonably consider closing collaboration channels (ICCs), which are the source of many problems. On the other hand Android applications are also demanding more concurrency, e.g. the

newest version features split screen for multiple foreground apps. Without voluntary acts in collaborations nor a feasible static sharing scheme, how can Android improve computation concurrency?

By looking into ASR apps, we propose two key techniques – a central service platform and an application-centric grouping. The details of both techniques have already been well explained in the thesis. Although it is studied through a speech recognition library, it has general indications to other I/O libraries, and concurrent I/O designs. But beyond that, it also has potential in dealing with third-party library and third-party application sharing ultimately. This is a very hard problem due to Android's one permission for all, the whole app can be corrupted by a un-verified third-party app or library. By being central, rather than verifying (time-consuming) and storing (memory-consuming) identical third-party libraries, the verification and sharing work all come to one point of control. Note that if the multiple version and upgrade issues of a library can be dealt with, it will attract even more applications to collaborate.

5.2 Looking Forward

While this work demonstrates the power of coordinated sharing to strike a balance between security and performance, the universal pattern of developing static approaches for even hardware systems are still not addressed in this thesis.

For software systems the patterns might be too complicated to be generalized. An emerging scheme for sharing is through collaborations and we see a great deal of space for future work here. This is by no means a new topic, the cooperative and non-cooperative game theories have been well studied and applied in Economics and Computer Science. Nash equilibrium is discussed during the design for the famous distributed system algorithms for fault-tolerance. However, these and other intelligent theories are still not widely applied. When the options are very limited because we are busy defending attacks while simultaneously trying to scale performance, it can be difficult to invest the time to see if these more fundamental approaches could shift the problem significantly.

The work presented here is a step towards new sharing patterns for both computation concurrency and security. The proposed collaboration pattern can be seen as between hardware and software, and between software applications who are supposed to be competitors. While Android has already done wonderful work in promoting collaborations among applications through interfaces and function calls, the flood of malicious acts has given us an important lesson. We must amend or reform the sharing pattern in software and operating systems. We should also carry this lesson down to accelerator rich platforms where similar sharing approaches are developing to avoid repeating the same mistakes there.

Application-centric grouping is only a tip of an iceberg of new patterns for safe and high performance sharing that we believe will be an important and continuing conversation by the community going forward. Many software sub-systems are in competition as they are controlled by different sets of stake holders, but they are also subject to limited resources – this is the typical game theory pattern. Looking forward we think the study of game theory might give further insight on the design of future platforms. At the simplest, each application can define its sharing realm with even hierarchies. As we get more advanced, applications might delegate work to other representatives (an authority app it trusts or an anti-virus app that is not favored by the system but might help). Beyond that, the sharing is not limited to computation concurrency, it can cover more general sharing of functionality including for security or for storage. It might not be confined within Android, nor within operating system. The usage is only natural, the Android ecosystem is just like a society. Without collaborations, how can one go far?

Bibliography

[1] http://www.windriver.com/announces/curiosity/Wind-River_NASA_0812.pdf.

[2] https://nepp.nasa.gov/mapld_2009/talks/083109_Monday/03_Malone_Michael_mapld09_pres_1.pdf.

[3] https://qz.com/826672/android-goog-just-hit-a-record-88-market-share-of-all-smartphones.

[4] CACTI 5.3. http://quid.hpl.hp.com:9081/cacti.

[5] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. Sok: Lessons learned from android security research for appified software platforms. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 433–451. IEEE, 2016.

[6] Onur Aciiçmez. Yet another microarchitectural attack:: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07, pages 11–18, New York, NY, USA, 2007. ACM.

[7] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' track at the RSA conference on Topics in Cryptology*, CT-RSA'07, pages 225–242, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[9] MA Anusuya and Shriniwas K Katti. Speech recognition by machine, a review. *arXiv preprint arXiv:1001.2267*, 2010.

[10] Aslan Askarov, Danfeng Zhang, and Andrew C Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 297–307. ACM, 2010.

[11] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the*

2016 ACM SIGSAC Conference on Computer and Communications Security, pages 356–367. ACM, 2016.

[12] L Bahl and Frederick Jelinek. Decoding for channels with insertions, deletions, and substitutions with applications to speech recognition. *IEEE Transactions on Information Theory*, 21(4):404–411, 1975.

[13] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous i/o support in linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.

[14] S Boyce and A Gorin. User interface issues for natural spoken dialog systems. *Proc. ISSD*, 96:65–68, 1996.

[15] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, volume 17, page 19, 2012.

[16] Jun-Hong Chen, Haw-Shiuan Wu, Ming-Der Shieh, and Wen-Ching Lin. A new montgomery modular multiplication algorithm and its vlsi design for rsa cryptosystem. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3780–3783. IEEE, 2007.

[17] Nathan Clark, Amir Hormati, and Scott Mahlke. Veal: Virtualized execution accelerator for loops. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 389–400. IEEE, 2008.

[18] Design Compiler. https://www.synopsys.com/tools/implementation/rtlsynthesis.

[19] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 305–310. IEEE, 2013.

[20] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: a composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 379–384. ACM, 2012.

[21] W. J. Dally, P. P. Carvey, and L. R. Dennison. The avici terabit switch/router. In *IEEE Hot Interconnects*, 1998.

[22] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[23] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *International Conference on Information Security*, pages 346–360. Springer, 2010.

[24] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[25] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, 2011.

[26] Leandro Fiorin, Gianluca Palermo, and Cristina Silvano. A security monitoring service for nocs. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, pages 197–202, New York, NY, USA, 2008. ACM.

[27] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 377–396. IEEE, 2016.

[28] C. H. Gebotys and Y. Zhang. Security wrappers and power analysis for soc technologies. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '03, pages 162–167, New York, NY, USA, 2003. ACM.

[29] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414 – 421, sept.-oct. 2005.

[30] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 503–514. IEEE, 2011.

[31] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[32] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proceedings of the 38th annual international symposium on*

Computer architecture, ISCA '11, pages 401–412, New York, NY, USA, 2011. ACM.

[33] Boris Grot, Stephen W. Keckler, and Onur Mutlu. Preemptive virtual clock: a flexible, efficient, and cost-effective qos scheme for networks-on-chip. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 268–279, New York, NY, USA, 2009. ACM.

[34] Boris Grot, Stephen W. Keckler, and Onur Mutlu. Topology-aware quality-of-service support in highly integrated chip multiprocessors. In *Proceedings of the 2010 international conference on Computer Architecture*, ISCA'10, pages 357–375, Berlin, Heidelberg, 2012. Springer-Verlag.

[35] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, page 31, 2011.

[36] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. In *ACM SIGOPS operating systems review*, volume 41, pages 175–188. ACM, 2007.

[37] A. Hansson, M. Subburaman, and K. Goossens. Aelite: A flit-synchronous network on chip with composable and predictable services. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 250 –255, april 2009.

[38] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, January 2009.

[39] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. A file is not a file: understanding the i/o behavior of apple desktop applications. *ACM Transactions on Computer Systems (TOCS)*, 30(3):10, 2012.

[40] R Hiremane. Intel virtualization technology for directed i/o (intel vt-d). *Technology@ Intel Magazine*, 4(10), 2007.

[41] David Huggins-Daines, Mohit Kumar, Arthur Chan, Alan W Black, Mosur Ravishankar, and Alexander I Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 1, pages I–I. IEEE, 2006.

[42] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting quasi-asynchronous i/o for better responsiveness in mobile devices. In *FAST*, pages 191–202, 2015.

[43] Slavisa Jovanovic, Camel Tanougast, and Serge Weber. A hardware preemptive multitasking mechanism based on scan-path register structure for fpga-based reconfigurable systems. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 358–364. IEEE, 2007.

[44] Younghyun Ju, Youngki Lee, Jihyun Yu, Chulhong Min, Insik Shin, and Junehwa Song. Symphoney: A coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 211–224. ACM, 2012.

[45] John Kim, James Balfour, and William Dally. Flattened butterfly topology for on-chip networks. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–182, Washington, DC, USA, 2007. IEEE Computer Society.

[46] Michel Kinsy and Michael Pellauer. Heracles: Fully synthesizable parameterized mips-based multicore system. Technical Report MIT-CSAIL-TR-2010-058, MIT Computer Science and Artificial Intelligence Laboratory, December 2010.

[47] Dirk Koch, Christian Haubelt, and Jürgen Teich. Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 188–196. ACM, 2007.

[48] Thomas Kuhn, Akhtar Jameel, M Stumpfle, and Afsaneh Haddadi. Hybrid in-car speech recognition for mobile multimedia applications. In *Vehicular Technology Conference, 1999 IEEE 49th*, volume 3, pages 2009–2013. IEEE, 1999.

[49] Bjornar Larsen and Chinatsu Aone. Fast and effective text mining using linear-time document clustering. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–22. ACM, 1999.

[50] Jae W. Lee, Man Cheuk Ng, and Krste Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 89–100, Washington, DC, USA, 2008. IEEE Computer Society.

[51] K-F Lee, H-W Hon, and Raj Reddy. An overview of the sphinx speech recognition system. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38(1):35–45, 1990.

[52] Youngki Lee, Chulhong Min, Younghyun Ju, Seungwoo Kang, Yunseok Rhee, and Junehwa Song. An active resource orchestration framework for pan-scale, sensor-rich environments. *IEEE Transactions on Mobile Computing*, 13(3):596–610, 2014.

[53] Xin Lei, Andrew W Senior, Alexander Gruenstein, and Jeffrey Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. In *Interspeech*, volume 1, 2013.

[54] Jiuxing Liu and Bulent Abali. Virtualization polling engine (vpe): using dedicated cpu cores to accelerate i/o virtualization. In *Proceedings of the 23rd international conference on Supercomputing*, pages 225–234. ACM, 2009.

[55] Slobodan Lukovic and Nikolaos Christianos. Enhancing network-on-chip components to support security of processing elements. In *Proceedings of the 5th Workshop on Embedded Systems Security*, WESS '10, pages 12:1–12:9, New York, NY, USA, 2010. ACM.

[56] Slobodan Lukovic and Nikolaos Christianos. Hierarchical multi-agent protection system for noc based mpsocs. In *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems*, S&D4RCES '10, pages 6:1–6:7, New York, NY, USA, 2010. ACM.

[57] Sheng Ma, Natalie Enright Jerger, and Zhiying Wang. Dbar: an efficient routing algorithm to support multiple concurrent applications in networks-on-chip. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 413–424, New York, NY, USA, 2011. ACM.

[58] Ken MacLean. http://www.voxforge.org.

[59] James H Martin and Daniel Jurafsky. Speech and language processing. *International Edition*, 710:25, 2000.

[60] Paul McNamee and Marty Hall. Developing a tool for memoizing functions in c++. *ACM SIGPLAN Notices*, 33(8):17–22, 1998.

[61] Konstantinos Menychtas, Kai Shen, and Michael L Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 301–316. ACM, 2014.

[62] Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277. ACM, 1968.

[63] ModelSim. http://www.mentor.com/products/fv/modelsim.

[64] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[65] R. Obermaisser and O. Hoftberger. Fault containment in a reconfigurable multi-processor system-on-a-chip. In *Industrial Electronics (ISIE), 2011 IEEE International Symposium on*, pages 1561 –1568, june 2011.

[66] The Trustees of the University of Pennsylvania. https://www.ldc.upenn.edu.

[67] OpenSSL. https://www.openssl.org.

[68] William R Otte, Abhishek Dubey, Subhav Pradhan, Prithviraj Patil, Aniruddha Gokhale, Gabor Karsai, and Johnny Willemsen. F6com: A component model for resource-constrained and dynamic space-based computing environments. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, pages 1–8. IEEE, 2013.

[69] J. Porquet, A. Greiner, and C. Schwarz. Noc-mpu: A secure architecture for flexible co-hosting on shared memory mpsocs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1 –4, march 2011.

[70] Alex Rudnicky. Sphinx knowledge base tool (2010). *URL http://www. speech. cs. cmu. edu/tools/lmtool. html.[Online].*

[71] Kyle Rupnow, Wenyin Fu, and Katherine Compton. Block, drop or roll (back): Alternative preemption methods for rh multi-tasking. In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 63–70. IEEE, 2009.

[72] John Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.

[73] Johan Schalkwyk, Doug Beeferman, Françoise Beaufays, Bill Byrne, Ciprian Chelba, Mike Cohen, Maryam Kamvar, and Brian Strope. your word is my command: Google search by voice: a case study. In *Advances in Speech Recognition*, pages 61–90. Springer, 2010.

[74] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[75] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, NOCS '12, pages 152–160, Washington, DC, USA, 2012. IEEE Computer Society.

[76] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. Flexdroid: Enforcing in-app privilege separation in android. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, pages 1–53, 2016.

[77] Ming-Der Shieh, Jun-Hong Chen, Hao-Hsuan Wu, and Wen-Ching Lin. A new modular exponentiation architecture for efficient design of rsa cryptosystem. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(9):1151–1161, 2008.

[78] Thad E Starner, Cornelis M Snoeck, Benjamin A Wong, and R Martin McGuire. Use of mobile appointment scheduling devices. In *CHI'04 Extended Abstracts on Human Factors in Computing Systems*, pages 1501–1504. ACM, 2004.

[79] R. Stefan, A. Molnos, A. Ambrose, and K. Goossens. A tdm noc supporting qos, multicast, and fast connection set-up. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1283 –1288, march 2012.

[80] Radu Stefan and Kees Goossens. Enhancing the security of time-division-multiplexing networks-on-chip through the use of multipath routing. In *Proceedings of the 4th International Workshop on Network on Chip Architectures*, NoCArc '11, pages 57–62, New York, NY, USA, 2011. ACM.

[81] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2013.

[82] Paul M Stillwell, Vineet Chadha, Omesh Tickoo, Steven Zhang, Ramesh Illikkal, Ravishankar Iyer, and Don Newell. Hippai: High performance portable accelerator interface for socs. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 109–118. IEEE, 2009.

[83] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, NOCS '12, pages 201–210, Washington, DC, USA, 2012. IEEE Computer Society.

[84] Jean-Loup Terraillon. Multicore processors - the next generation computer for esa space missions. `http://www.cister.isep.ipp.pt/ae2012/presentations_pdf/thursday/k/terraillon.pdf`. "Keynote address.".

[85] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: a hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd Annual IEEE/ACM Inter-*

national Symposium on Microarchitecture, MICRO 42, pages 493–504, New York, NY, USA, 2009. ACM.

[86] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011. ACM.

[87] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 109–120, New York, NY, USA, 2009. ACM.

[88] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *NDSS*, 2015.

[89] Yao Wang and G.E. Suh. Efficient timing channel protection for on-chip networks. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 142 –151, may 2012.

[90] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[91] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society.

[92] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, September 2007.

[93] Rubin Xu, Hassen Saïdi, and Ross J Anderson. Aurasium: practical policy enforcement for android applications. In *USENIX Security Symposium*, volume 2012, 2012.

[94] Lok-Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX security symposium*, pages 569–584, 2012.

[95] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiaoyong Zhou, and XiaoFeng Wang. Leave me alone: App-level protection against runtime information gathering on android. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 915–930. IEEE, 2015.

[96] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.