EFFICIENT MICROARCHITECTURE FOR NETWORK-ON-CHIP ROUTERS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Daniel U. Becker
August 2012

© 2012 by Daniel Ulf Becker. All Rights Reserved.
Re-distributed by Stanford University under license with the author.

This dissertation is online at: http://purl.stanford.edu/wr368td5072

Includes supplemental files:
1. Parameterized Network-on-Chip Router RTL *(router.tgz)*
2. Common RTL Component Library *(clib.tgz)*

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William Dally, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Continuing advances in semiconductor technology, coupled with an increasing concern for energy efficiency, have led to an industry-wide shift in focus towards modular designs that leverage parallelism in order to meet performance goals. Networks-on-Chip (NoCs) are widely regarded as a promising approach for addressing the communication challenges associated with future Chip Multi-Processors (CMPs) in the face of further increases in integration density. In the present thesis, we investigate implementation aspects and design trade-offs in the context of routers for NoC applications. In particular, our focus is on developing efficient control logic for high-performance router implementations.

Using parameterized RTL implementations, we first evaluate representative Virtual Channel (VC) and switch allocator architectures in terms of matching quality, delay, area and power. We also investigate the sensitivity of these properties to key network parameters, as well as the impact of allocation on overall network performance. Based on the results of this study, we propose microarchitectural modifications that improve delay, area and energy efficiency: Sparse VC allocation reduces the complexity of VC allocators by exploiting restrictions on transitions between packet classes. Two improved schemes for speculative switch allocation improve delay and cost while maintaining the critical latency improvements at low to medium load; this is achieved by incurring a minimal loss in throughput near the saturation point. We also investigate a practical implementation of combined VC and switch allocation and its impact on network cost and performance.

The second part of the thesis focuses on router input buffer management. We

explore the design trade-offs involved in choosing a buffer organization, and we evaluate practical static and dynamic buffer management schemes and their impact on network performance and cost. We furthermore show that buffer sharing can lead to severe performance degradation in the presence of congestion. To address this problem, we introduce Adaptive Backpressure (ABP), a novel scheme that improves the utilization of dynamically managed router input buffers by varying the stiffness of the flow control feedback loop based on downstream congestion. By inhibiting unproductive buffer occupancy, this mitigates undesired interference effects between workloads with differing performance characteristics.

Acknowledgements

First and foremost, I must thank my adviser, Professor William J. Dally, for his support and guidance throughout my career as a graduate student. It has been a privilege to be able to draw on Bill's vast experience and breadth of knowledge, and my research has benefitted greatly from his keen insights and critical feedback over the years.

I would also like to thank the members of my reading committee, Professor Christos Kozyrakis and Professor Kunle Olukotun. I was fortunate to have both as teachers during my first year at Stanford, and I thoroughly enjoyed our interactions as part of the Pervasive Parallelism Lab in recent years. I furthermore thank Professor Yoshio Nishi for serving as the chair for my dissertation defense.

I am grateful to the past and present members of the Concurrent VLSI Architecture group at Stanford for providing a stimulating and fun work environment for the past five years. George Michelogiannakis shared an office with me for most of my time at Stanford and was a dependable comrade-in-arms in the crusade for free food. In addition to fruitful discussions, Curt Harting and Ted Jiang provided for many entertaining Friday evenings in Gates and patiently listened to my venting when the need arose. I also thank our support staff, particularly Uma Mulukutla and Sue George, for working behind the scenes to make our lives easier.

The research leading up to this dissertation would not have been possible without the generous financial support provided by the Professor Michael J. Flynn Stanford Graduate Fellowship. Additionally, I am indebted to the National Science Foundation (Grant CCF-0702341) and the National Security Agency (Contract H98230-08-C-0272-P007) for funding parts of my work.

On a more personal note, I would like to thank the many amazing folks outside of my research area that I have had the opportunity to meet along the way. I am particularly grateful to Emily Deal for all of her support and encouragement over the past three years, and especially for her patience and her help during the final stretch of my PhD; the fact that I have retained a shred of sanity in the process is largely to her credit. Five iterations of the Stanford Men's Volleyball Club team forced me to leave the comfort of my office on a somewhat regular basis and contributed many memorable moments to my Stanford experience. I also thank the Stanford German Student Association for their part in creating a home away from home. Many others have contributed to making the past five years of my life a largely enjoyable experience, and I am grateful to all of them.

Finally and most importantly, I thank my family back home for their unconditional support, their unwavering confidence in my abilities and their encouragement in all of my endeavors—even the ones that put us nine time zones apart. Without you, this journey would not have been possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Continuing advances in semiconductor process technology are providing chip designers with ever increasing transistor budgets. Traditionally, each new process generation has resulted in faster, smaller and more efficient transistors; as a result, designers have historically focused on improving single-threaded performance by means of higher clock speeds and the use of wider and more sophisticated microarchitectures that improve instruction execution rates by attempting to extract increasing amounts of parallelism from a sequential instruction stream. However, with the end of Dennard scaling [22], further clock frequency increases are constrained by practical limits on power dissipation; at the same time, timing overhead and the performance implications of pipeline flushes render further increases in pipeline depth impractical. Finally, superscalar execution techniques have reached a point of diminishing returns as typical instruction streams only offer a limited amount of parallelism that can be extracted at reasonable cost. At the same time, there is substantial demand for continued improvements in processing power. In combination with an increasing concern for energy efficiency driven by the rise of mobile devices, this has led to an industry-wide shift in focus towards designs that rely on explicit parallelism to achieve their performance and efficiency goals [2, 13, 69].

In such parallel designs, system performance is defined by the *aggregate* performance across multiple processing elements, and process scaling is exploited by increasing the number—rather than the complexity—of such elements. Hence, as scaling continues, it becomes necessary to divide a given problem into an increasingly larger number of sub-problems in order to realize continued performance improvements; for a fixed problem size, this generally implies an increase in the amount of communication between processing elements, each of which is responsible for a proportionally smaller slice of the overall problem. With future designs expected to integrate hundreds of processing cores on a single chip, on-chip communication is thus expected to have a significant impact on chip-level performance and energy efficiency [30, 48, 81, 91].

1.1.1 Networks-on-Chip

Networks-on-Chip (NoCs) are widely regarded as a promising approach for addressing the communication demands of large-scale Chip Multi-Processors (CMPs) [9, 19, 33][1]. Such packet-switched on-chip interconnects are embodied by a set of routers that are connected to each other and to the network endpoints by point-to-point links, and they are characterized by three primary design parameters:

Topology: The network topology dictates the number of routers and channels and the connectivity among them. In determining network diameter and bisection bandwidth, it establishes basic bounds for overall network performance and energy efficiency. Furthermore, by controlling the number and size of individual network components, the topology represents a critical factor in determining overall network cost.

Due to the realities of semiconductor manufacturing, NoCs typically favor the use of tiled two-dimensional topologies like the Mesh, Concentrated Mesh (CMesh) [4] or Flattened Butterfly (FBfly) [45].

[1] Additionally, compared to traditional approaches that rely on ad-hoc wiring, the use of a structured, packetized interconnect can reduce design complexity and verification effort, as it inherently imposes well-defined module boundaries.

Routing: The routing function selects the path that a given packet must take from its source to its destination endpoint. As such, it affects the average hop count and the degree to which load is balanced across network channels, effectively limiting how closely the observed performance and energy efficiency can approach the bounds established by the network topology.

Because of delay and cost constraints, NoCs typically employ simple arithmetic routing functions like Dimension-Order Routing (DOR); i.e., a packet's output port at each router is computed from its destination address and the address of the router.

Flow control: The flow control scheme governs how routers communicate with each other; in particular, it determines when packets—or, in many practical implementations, fixed-size parts of packets called *flits*—can be forwarded from one router to the next. Consequently, flow control regulates resource utilization and thus has a significant impact on performance. In addition, the buffer space requirements imposed by a given flow control scheme directly affect the implementation cost and power consumption of each router.

Most NoC designs employ Virtual Channel (VC) flow control [17, 18]; however, recent work has investigated alternative flow control schemes in an effort to reduce buffer overhead [55, 60].

While these parameters set the general framework for the network's performance, cost and efficiency, the specific characteristics of a NoC depend on the implementation of its basic components. In particular, routers and network channels set the latency and energy cost incurred for each hop that a packet takes on its way through the network. In addition, the microarchitecture of the routers governs the concrete implementation of the routing algorithm and flow control scheme, determines the behavior in the presence of congestion, and limits the maximum operating frequency; as a result, it directly affects overall network throughput. Finally, the actual implementation cost of the network is clearly defined by the cost of its components.

To highlight the importance of developing efficient NoCs, Figure 1.1 shows the power breakdown for the Intel Teraflop Research Chip [39, 95]. This chip comprises

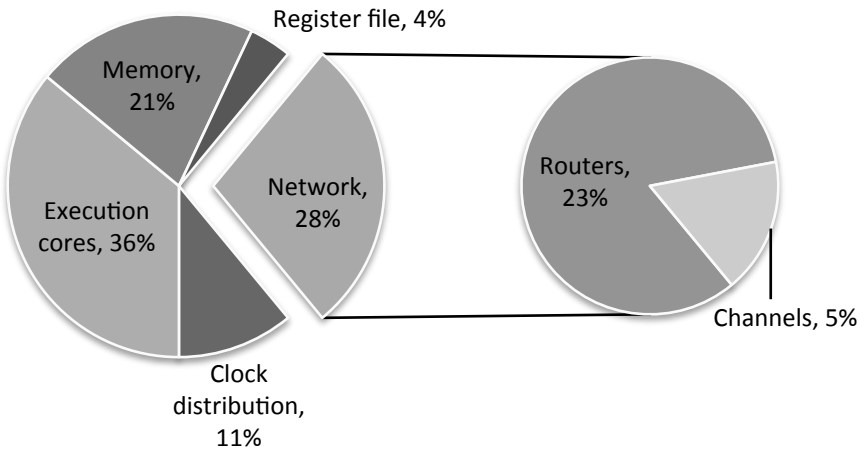


Figure 1.1: Power breakdown for Intel Teraflop Research Chip.

80 tiles—each containing a pair of simple floating point execution units and associated memory arrays—arranged in an 8×10 Mesh with 38 bit wide channels and 2 VCs. When executing a communication-heavy Stencil kernel, the NoC accounts for more than a quarter of the overall chip power; 83 % of the network power is in the routers.

As the number of cores continues to scale up, the impact of the network will become even more pronounced. For example, recent work has found that the underlying 16×16 Mesh accounts for 45 % of the total energy expended in performing a 10^6-element radix sort on a cache-coherent 256-node CMP [34].

1.1.2 Router Microarchitecture

Developing efficient channels is largely a circuit design problem [14,36,37]; in contrast, a router's performance, cost and efficiency primarily depend on its microarchitecture. The present dissertation investigates implementation aspects and microarchitectural design trade-offs for efficient high-performance NoC routers.

Compared to routers in traditional long-haul and system interconnection networks,

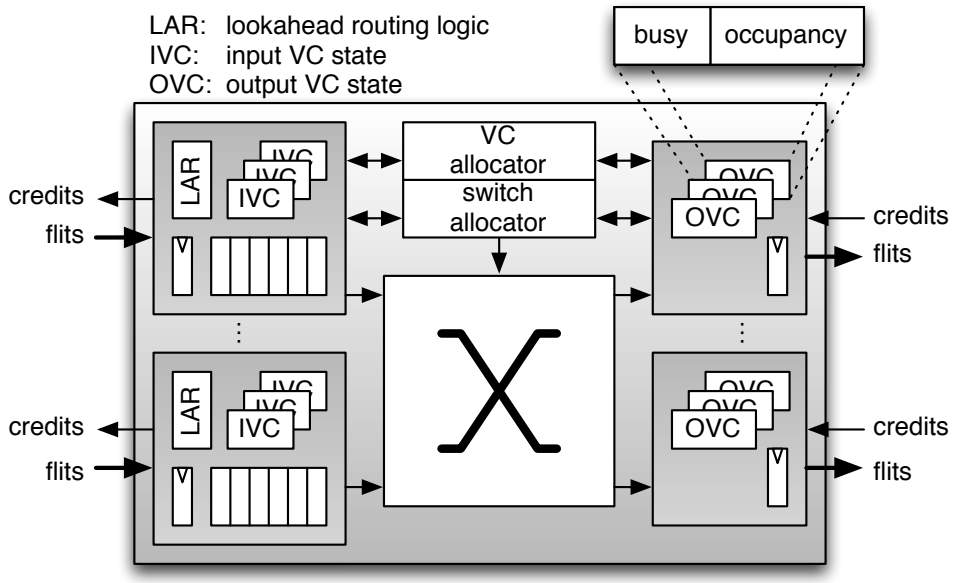


Figure 1.2: Router microarchitecture overview.

NoC routers are subject to markedly different design constraints: As modern semiconductor processes provide an abundant supply of on-chip wiring resources, there is typically no need to use SERDES or sophisticated encoding schemes on the channels. At the same time, performance in CMPs is typically much more sensitive to network latency, mandating both the use of shallow router pipelines and aggressive cycle time targets. Finally, NoCs are generally subject to stringent area and power constraints to avoid interference with the requirements of the network endpoints.

Input-queued routers have emerged as the architecture of choice in current NoC research; in such designs, packets that cannot be forwarded immediately are temporarily held in FIFO buffers at the routers' input ports until they can proceed to the next hop. Buffer space is allocated at the granularity of fixed-size flits, one or more of which comprise a packet, and is logically divided into multiple VCs in order to avoid deadlock and to reduce Head-of-Line (HoL) blocking [17, 18]. Figure 1.2 shows an illustration of a typical NoC router.

A router's primary function is to forward each flit that arrives on one of its inbound channels to an appropriately selected outbound channel. To this end, the following

steps must be completed before the flit can depart from its desired output port:

Route computation: The routing logic selects a suitable output port and a set of candidate output VCs for each packet according to the routing algorithm implemented in the network. This step only needs to be performed for the first flit—also termed *head flit*—of each packet; any subsequent flits from the same packet simply inherit the selected output port.

As latency is a critical performance metric in many CMPs, NoC routers commonly implement *lookahead routing* in order to reduce pipeline delay [28].

VC allocation: VC flow control requires each packet to secure exclusive access to an output VC at the selected destination port before flits can be forwarded. The VC allocator assigns available output VCs to waiting packets at the router's input ports. Similar to route computation, VC allocation only needs to be performed for head flits, as the assigned output VC is inherited by subsequent body and tail flits from the same packet.

Switch allocation: Once a packet has been assigned an output port and VC, it can participate in switch allocation. The switch allocator is responsible for establishing a crossbar schedule by assigning time slots to waiting flits at the router's input ports. In particular, it must resolve conflicts between flits destined for the same output port.

Switch traversal: Finally, after receiving a grant from the switch allocator, a flit can traverse the router's central crossbar switch in the next cycle to arrive at its destination output and continue its journey through the network.

As in the case of network channels, designing fast and efficient crossbars is primarily a circuit design problem.

Allocators represent a particularly important aspect of router design, as they directly affect overall network performance in several ways: Allocation quality, measured in the cardinality of matchings between requests and available resources, determines the utilization of the router's VCs, the crossbar and the network channels;

as such, it has an immediate impact on the network's throughput under load and on the queuing delay that packets incur in a congested network. Allocators furthermore control the network's fairness properties. Finally, in many typical router designs, allocation directly affects the critical path delay; consequently, delay-optimized allocator implementations are required to enable the network to achieve high operating frequencies.

Prior research has shown that input buffers can account for a significant fraction of a router's overall area and power budget [15,96,98] and that buffer space represents an expensive commodity in the on-chip environment [19, 40]; for example, buffers account for 27 % of the router power in the Intel Teraflop Research Chip [39]. At the same time, network utilization and performance are highly sensitive to the amount of buffer space that is available to individual packets [17,40,66,80]. Flexible buffer management schemes that yield improvements in buffer *utilization* represent an attractive approach for reducing buffer cost without sacrificing performance [66, 90]. However, in developing such schemes, we must carefully consider any overhead introduced by the buffer management logic itself and avoid undesired side effects on other important characteristics of the network.

Throughout this dissertation, we focus primarily on synthesis-based implementations of router components. This is reflective of recent industry trends, which restrict the use of full-custom logic to critical data path components and large regular structures, particularly memories, in an effort to maximize designer productivity. For example, over the past five generations of IBM POWER and zSeries processors, the fraction of full-custom blocks has shrunk by a factor of ten [27].

1.2 Contributions

This dissertation investigates implementation aspects and microarchitectural design trade-offs for efficient high-performance NoC routers. In particular, it makes the following specific contributions:

- We evaluate and compare standard-cell implementations of key router control

logic components in terms of delay, area and energy efficiency in a commercial 45nm process.

- We develop efficient wavefront allocator implementations that avoid combinational cycles, facilitating their use in synthesis-based design flows; our designs improve delay and cost compared to the state-of-the-art implementation described in [43]. We furthermore propose a simple mechanism for alleviating inherent fairness issues in wavefront allocation by modifying the order in which priority diagonals are selected.

- We propose sparse VC allocation, a scheme that reduces the complexity—and hence the delay and cost—of VC allocators by exploiting restrictions on the possible transitions between VCs assigned to different packet classes. In doing so, it increases the router's scalability and facilitates the use of higher-radix network topologies.

- We develop two new implementation variants for speculative switch allocation: Pessimistic speculation takes advantage of the fact that speculative switch allocation is most beneficial at low to medium network load where most requests are granted, while priority-based speculation uses a priority-aware allocator to handle both speculative and non-speculative requests instead of using two separate sub-allocators. Both variants sacrifice some of the performance benefits of speculation near saturation in order to reduce delay and cost compared to the canonical implementation described in [77]; however, they maintain the critical latency benefits under low to medium load.

- We describe a practical implementation of combined VC and switch allocation and compare the resulting performance and cost to a canonical router design. Combined allocation yields the same latency improvements as speculative switch allocation at low to medium network load. While allocation inefficiencies lead to slightly reduced throughput near saturation, the cost and delay benefits of avoiding a dedicated VC allocator render combined allocation an attractive design choice for many network configurations.

- We investigate the key trade-offs in the organization and design of router input buffers and evaluate practical buffer management schemes in terms of overhead and performance.

- We identify a performance pathology associated with dynamic buffer management that can lead to undesired interference between multiple traffic classes with different performance characteristics. To address this problem, we develop Adaptive Backpressure (ABP), a novel mechanism that avoids unproductive use of buffer space by regulating credit flow based on observed performance characteristics. We show that ABP can be implemented with minimal changes to the router, and that it effectively mitigates interference without degrading performance under benign load conditions

- Finally, we have developed a parameterized RTL implementation of an NoC router, which we have released to the research community as open source. In addition to facilitating detailed evaluations of implementation trade-offs for the present dissertation, the router has since found use in a number of other research efforts at Stanford and beyond [10, 11, 44, 53, 55, 56, 59, 63, 64, 72, 73].

Parts of our work on VC and switch allocator implementations were previously published in [7]; the work on ABP has been accepted for publication in [8].

1.3 Outline

The remainder of this dissertation is organized as follows:

Chapter 2 discusses elementary arbiter designs. We investigate different arbiter types, discuss approaches for providing fairness, scalability and support for multiple priority levels, and evaluate delay and cost in a commercial standard-cell design flow.

Chapter 3 similarly investigates allocators. In particular, our focus is on wavefront allocators: We develop a scheme to alleviate inherent fairness issues in wavefront allocation, and we propose several synthesis-friendly implementation variants that improve delay and cost compared to a state-of-the-art design.

In Chapter 4, we discuss how the basic allocator designs described in Chapter 3 can be used to implement VC allocation. We give an overview of practical implementation variants, describe sparse VC allocation, and present detailed evaluation results for delay, cost and performance.

Chapter 5 analogously addresses the application of elementary allocator designs in the implementation of practical switch allocators. We describe and evaluate exemplary architectures. Furthermore, we discuss speculative switch allocation, and we suggest two implementation variants that improve delay and cost over the canonical design. Finally, we describe and evaluate a practical implementation of combined VC and switch allocation.

Chapter 6 explores trade-offs in input buffer organization and examines state-of-the-art static and dynamic buffer management schemes. We compare the individual schemes in terms of implementation overhead, discuss deadlock avoidance considerations and evaluate cost-performance trade-offs.

In Chapter 7, we show that dynamic buffer management can lead to severe performance degradation and undesired interference between different traffic classes in the presence of congestion. We develop ABP to mitigate this effect, and we present simulation results both for synthetically generated traffic and for application traffic on a heterogeneous CMP to demonstrate its efficacy.

To conclude the dissertation, Chapter 8 summarizes our contributions and briefly outlines opportunities for future work.

Finally, Appendix A provides a brief overview of the parameterized router RTL that was developed as part of the work described in this dissertation.

Chapter 2

Arbiters

2.1 Overview

Mediating access to a shared resource between multiple agents is one of the fundamental operations performed by the control logic in a router. We refer to the act of coordinating access in this way as *arbitration*, and to the logic that performs this function as an *arbiter*.

Formally, we can describe the process of arbitration among n agents as a vector operation Φ on a binary request vector $R = \{r_0, r_1, \ldots, r_{n-1}\}$ that produces a grant vector $G = \{g_0, g_1, \ldots, g_{n-1}\}$ whose elements satisfy certain properties:

$$G = \Phi(R) \tag{2.1}$$

$$g_i \Rightarrow r_i \tag{2.2}$$

$$g_i \Rightarrow \bigwedge_{j \neq i} (\neg g_j) \tag{2.3}$$

Equation 2.2 requires that the generated grant vector be consistent with the request vector; i.e., only those agents that actually issued a request in the first place can receive a grant. Equation 2.3, on the other hand, ensures that in case of conflict, only a single agent receives a grant.

While not strictly required, most practical arbiters generate grant vectors that

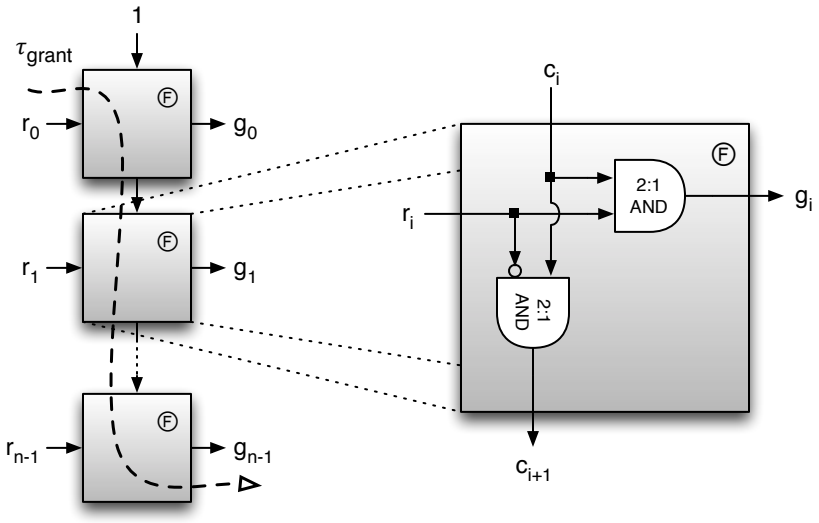


Figure 2.1: Linear implementation of a fixed-priority arbiter.

satisfy an additional property:

$$\bigvee_i r_i \Rightarrow \bigvee_i g_i \tag{2.4}$$

That is, if one or more agents request access to the shared resource, one of them will receive a grant. We can take advantage of this property to simplify logic in cases where we need to know whether a resource was granted, but not which particular agent it was granted to.

2.2 Fixed-Priority Arbiters

This simplest form of arbiters grants access to a shared resource based on a predetermined priority order. If the request inputs are sorted in descending priority order, solving this problem is equivalent to finding the first set bit in a bit vector. Figure 2.1 shows a straightforward implementation using a linear array of basic bit cells Ⓕ, each of which generates a grant g_i if both its request input r_i and the incoming priority signal c_i are asserted. In addition, the incoming priority signal is propagated to the next cell only if r_i is not asserted. This design minimizes hardware complexity; however,

its critical path delay τ_{grant} scales linearly with the number of inputs, as indicated by the dashed arrow in Figure 2.1.

If a large number of inputs must be supported, we can improve delay by taking advantage of the fact that the logic equations for the g_i and c_{i+1} outputs are structurally similar to those for a binary half adder's sum and carry outputs, respectively. As such, it is possible to transform the design shown in Figure 2.1 into an equivalent prefix network that hierarchically computes propagation conditions for the initial priority signal, causing the delay to scale logarithmically with the number of inputs.

2.3 Round-Robin Arbiters

Fixed-priority arbiters require that there be a clear, pre-established priority order among requests. However, in the context of routers, we frequently encounter situations where undifferentiated agents compete for access to a shared resource. In such cases, we are generally interested in maintaining a degree of fairness among the agents: At a minimum, we want to ensure that every request is granted eventually (*weak fairness*); ideally, though, grants should be distributed equitably among agents (*strong fairness*).

The round-robin arbiter shown in Figure 2.2 extends the fixed-priority scheme from Section 2.2 by adding a priority select input p_i to each bit cell Ⓡ and wrapping around the last bit cell's priority output c_n to the first one's priority input c_0. The priority select inputs are driven by a state register that contains the most recent grant, rotated by one bit position[1]. Thus, every time a grant is generated, the next agent in line after the one being granted becomes the highest-priority request in the next cycle; this scheme provides strong fairness.

The priority mechanism introduces an additional timing arc τ_{update}, as shown in Figure 2.2. Depending on the timing constraints for the grant outputs g_i, either τ_{update} or τ_{grant} may represent the critical path for a given arbiter instance.

[1] For large arbiters, it is often preferable to reduce overhead by storing this value in binary-encoded form if delay constraints permit.

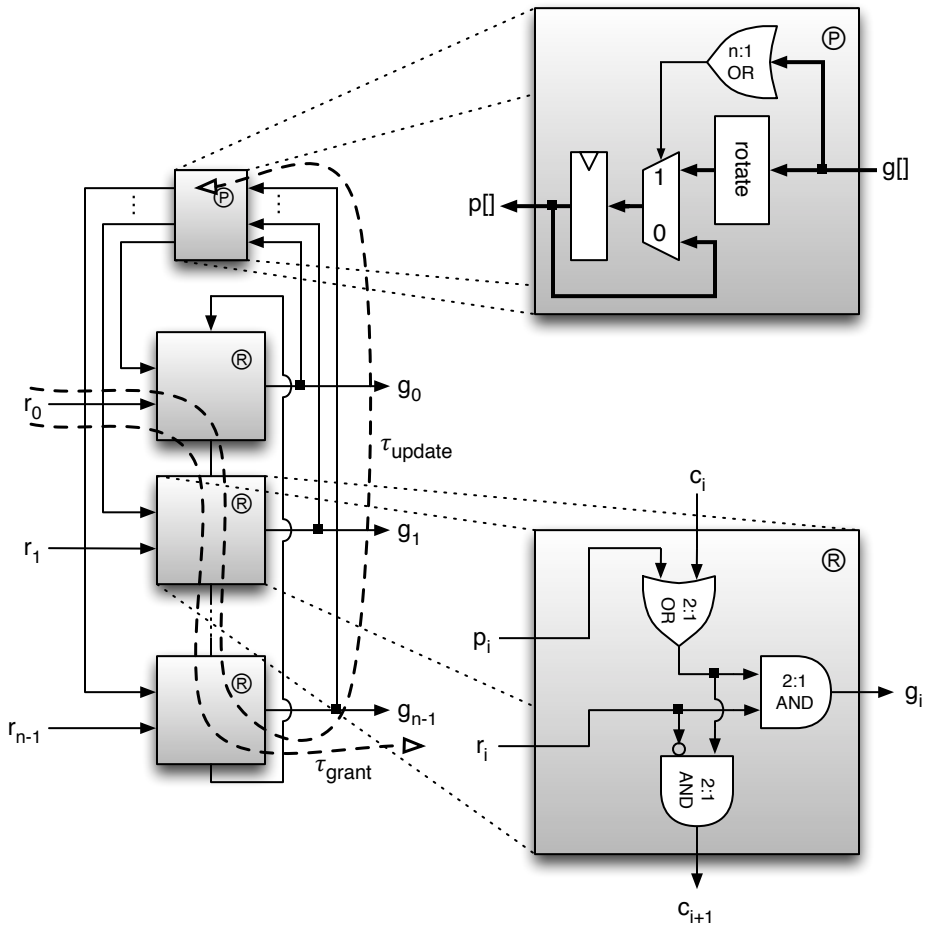


Figure 2.2: Linear implementation of a round-robin arbiter.

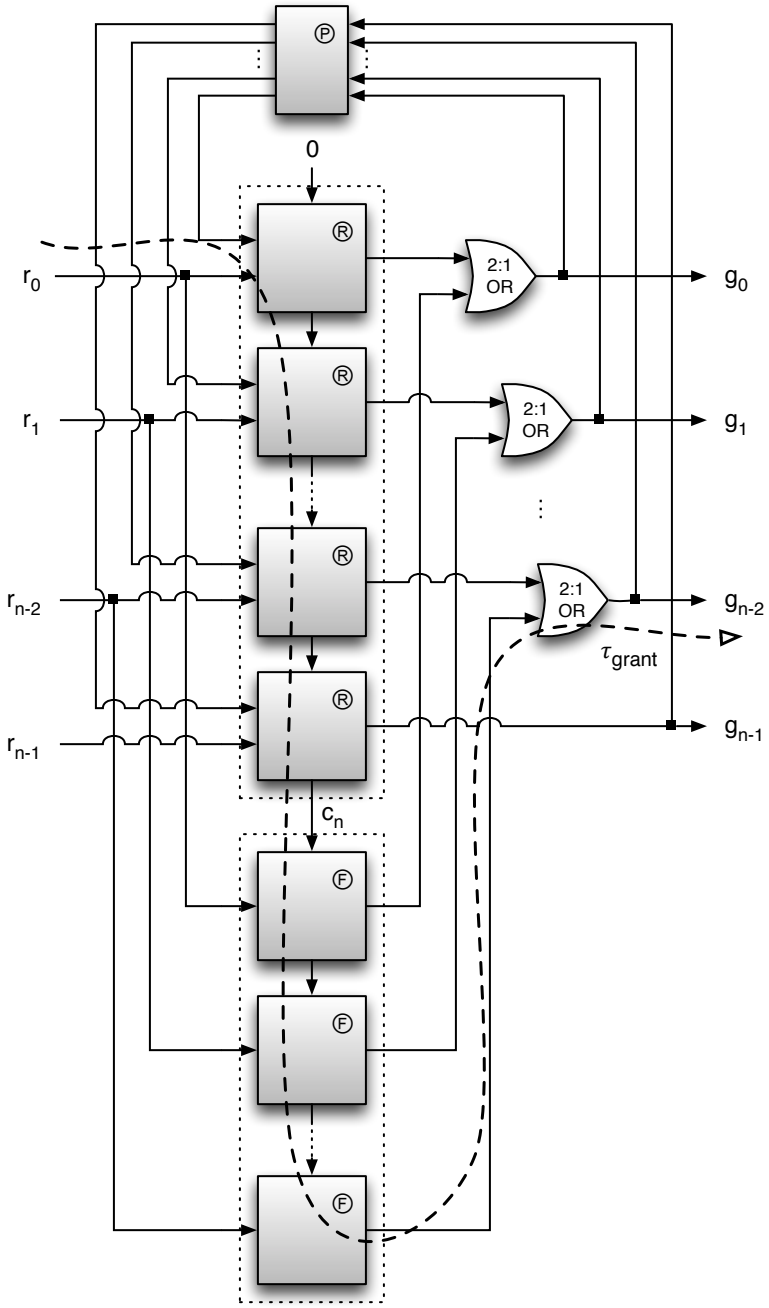


Figure 2.3: Acyclic implementation of a round-robin arbiter.

Commercially available timing analysis tools are typically unable to properly analyze circuits that contain combinational loops like the one generated by wrapping around the priority signals. This is particularly problematic when using a standard cell design flow, as it prevents synthesis from performing proper gate sizing. We can avoid the combinational loop by severing the wraparound priority signal and connecting it to a chain of fixed-priority cells Ⓕ as shown in Figure 2.3[2]. In a naïve implementation, doing so nearly doubles the grant generation delay τ_{grant}; however, in practice, the associated delay penalty can be minimized by computing the intermediate grant signals for the block of Ⓡ-type cells and the block of Ⓕ-type cells in parallel and conditionally selecting either set of grants based on the value of the first block's final priority output c_n[3]. As in the case of fixed-priority arbitration, we can further improve delay by replacing the blocks of Ⓡ-type and Ⓕ-type cells with prefix networks. Alternatively, it is possible to implement the entire round-robin arbiter as a single acyclic prefix network; Dimitrakopoulos et al. describe such a design in detail in [23].

2.4 Matrix Arbiters

Matrix arbiters represent another implementation alternative for providing strong fairness. This is achieved by explicitly tracking pairwise precedence between all request inputs and updating it in response to grants in a way that implements a *least-recently-served* policy. Specifically, for every pair of inputs i and j, the precedence indicator $p_{i,j}$ determines whether a pending request from input i has higher priority than a pending request from input j. The precedence indicators are stored in a matrix of registers that lends this type of arbiter its name[4]. Any time an input k is granted, it assumes lowest priority by setting $p_{i,k}$ and resetting $p_{k,j}$ for all i and j.

Using the precedence values, the grant signal g_k for a given agent k can thus be

[2] Because priority selection never causes r_{n-1} to be skipped, it does not require an Ⓕ-type cell.
[3] We will use a similar approach to implement multi-priority arbitration in Section 2.6.
[4] Only the upper triangle of the matrix actually requires registers, as $p_{i,j} = \neg p_{j,i}$ and $p_{i,i} = 0$.

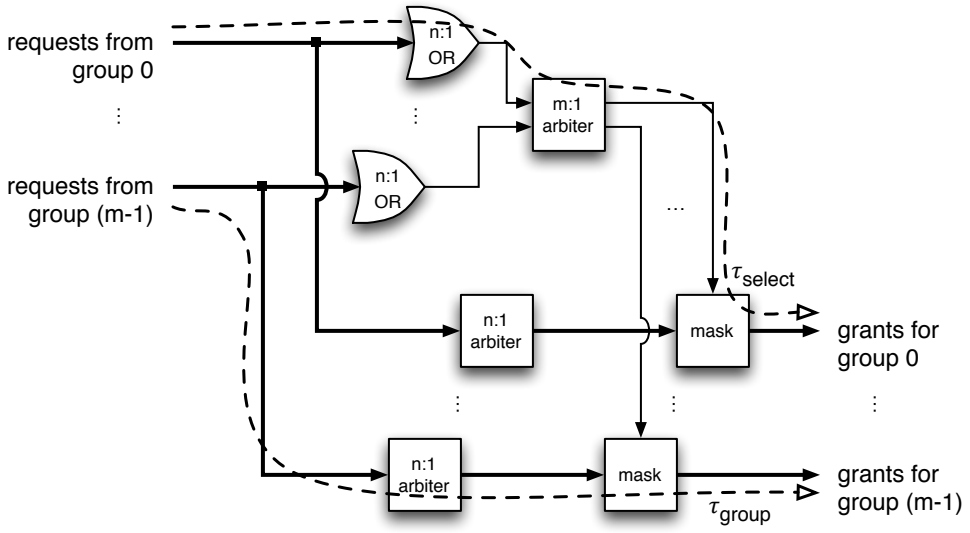


Figure 2.4: Tree arbiter with $m \times n$ inputs.

computed using a simple reduction tree:

$$g_k = r_k \wedge \neg \bigvee_{i \neq k} (r_i \wedge p_{i,k}) \tag{2.5}$$

The associated grant generation delay τ_{grant} grows logarithmically with the number of inputs. However, updating the precedence values involves fanout from each grant signal to $2 \times n$ registers, leading to a larger update delay τ_{update} compared to a comparable round-robin arbiter. Furthermore, the number of registers required to hold the precedence matrix—a primary factor in the allocator's implementation cost—exhibits quadratic scaling[5]. As a result, this scheme is typically only attractive for arbiters with a relatively small number of inputs.

2.5 Tree Arbiters

In many applications that require large arbiters, the requesting agents are logically organized into multiple groups. In such cases, a single arbiter as described in the preceding two sections distributes grants evenly to all requesting agents regardless of which group they fall into. However, in practice, it is typically preferable to first distribute grants fairly among the different groups and then among the individual agents within each group. This can be achieved using a hierarchical organization of smaller arbiters as shown in Figure 2.4: An $m \times n$-input *tree arbiter* uses m independent n-input arbiters of arbitrary type to determine a winning agent for each individual group; in parallel, a single m-input arbiter selects a winner among all groups that have at least one request and enables only that group's outputs.

If the number and size of groups in a tree arbiter is optimized to match the delay along the two timing arcs τ_{select} and τ_{group} shown in Figure 2.4, it can achieve lower delay than a monolithic arbiter with the same number of ports. Additionally, for matrix arbiters, tree organization can significantly reduce area and power compared to a monolithic implementation; in particular, implementing an n-input arbiter as l levels of m-input matrix arbiters with $m = \sqrt[k]{n}$ reduces the total number of registers required by a factor of m compared to a monolithic n-input matrix arbiter[6].

We can construct tree arbiters with more than two levels by recursively implementing the individual m- and n-input arbiters in Figure 2.4 as tree arbiters themselves. For such multi-level tree arbiters, it is typically beneficial to combine the group enable signals across the levels of the tree and to perform a single stage of masking after the last level, rather than performing individual masking steps after each individual level.

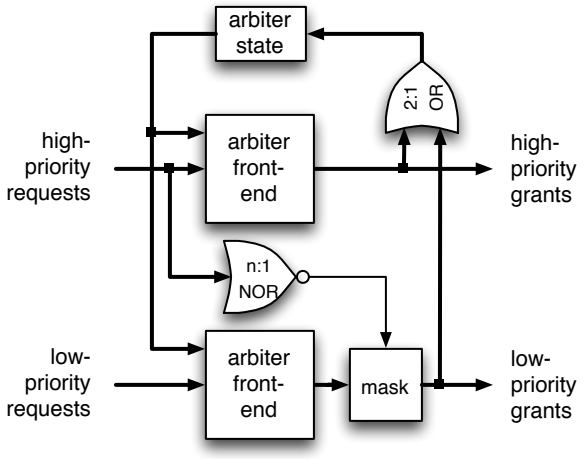


Figure 2.5: Dual-priority arbiter.

2.6 Multi-Priority Arbiters

Many applications in which arbiters are used call for the ability to dynamically prioritize a subset of requests that satisfy certain criteria over other requests. E.g., as we shall see in Chapter 5, a speculative switch allocator prioritizes requests from packets that have completed Virtual Channel (VC) allocation over those from packets that have not yet been assigned an output VC.

In many such instances, only a small number of distinct priority levels is required; this enables us to implement multi-priority arbitration by replicating the arbiter front-end—i.e., the part of the arbiter that is responsible for computing grant signals—and masking the intermediate grants generated by the individual front-end instances based on the presence of higher-priority requests as illustrated in Figure 2.5[7]. Masking outputs in this way ensures that only a single request across all priority levels can receive a grant, while sharing the arbiter state among all front-end instances reduces hardware overhead.

[5] In particular, a total of $\frac{n(n-1)}{2}$ registers are required for an n-input matrix arbiter.
[6] Note that the l-level tree comprises a total of $\frac{n-1}{m-1}$ m-input matrix arbiters.
[7] We assume here that the individual arbiters satisfy Equation 2.4.

While this approach can support a small number of priority levels with comparatively little area and delay overhead, it is impractical for applications that require fine-grained priorities. Such applications fall outside the scope of the present contribution.

2.7 Evaluation

In this section, we evaluate the efficiency of the different types of arbiters discussed in the present chapter. To this end, we develop parameterized RTL implementations and synthesize them in a commercial standard-cell design flow. In particular, we consider three different implementations:

Round-robin: An acyclic round-robin arbiter as described in Section 2.3, with prefix networks used in place of the two chains of bit cells and conditional grant selection.

Prefix: An implementation using a single acyclic prefix networks as described by Dimitrakopoulos et al. [23].

Matrix: A matrix arbiter as described in Section 2.4.

To evaluate scaling behavior, we compare instances of each type of arbiter with five, eight and ten ports. These sizes were chosen to be reflective of typical and large arbiters in Networks-on-Chip (NoCs), respectively. Situations in which arbitration among more than ten agents is necessary typically call for the use of tree arbiters.

2.7.1 Experimental Setup

For each configuration, we sweep the target clock frequency under worst-case process conditions and report the resulting cell area, as well as the Power-Delay Product (PDP) assuming a 50 % activity factor at all request inputs. The latter quantity represents a measure of the amount of energy that is expended in each round of arbitration. We begin each sweep at a base frequency of 1.0 GHz and add 100 MHz

Table 2.1: Experimental setup details.

Design Compiler version	G-2012.06
Target library	TCBN45GSBWP
FO4 delay	34.6 ps
Driving / receiving cells	INVD8BWP

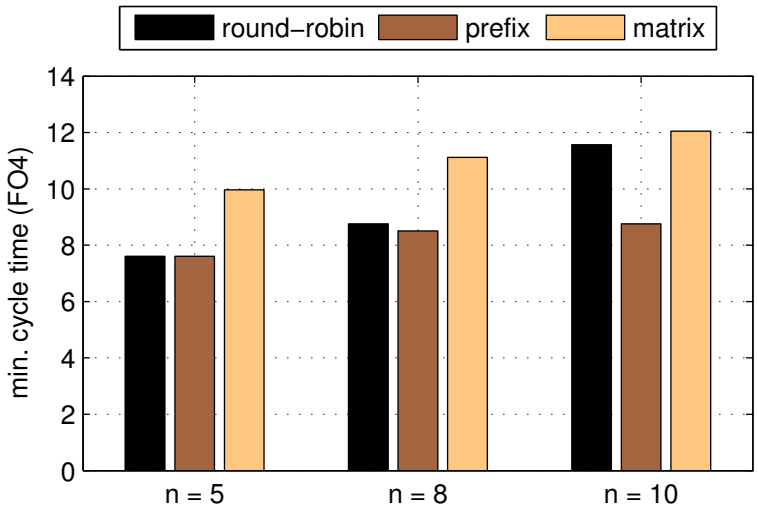


Figure 2.6: Minimum delay for n-port arbiter implementations.

increments until a timing violation is reported. We convert all clock frequencies into their corresponding cycle times and normalize them to the FO4 delay for our target library.

Synthesis is performed using the Synopsys Design Compiler Reference Methodology, targeting a general-purpose library in the TSMC 45GS process. We disable Design-for-Test (DFT) synthesis and enable dynamic power optimization, but otherwise use the default parameters specified by the reference methodology. Arbiter inputs and outputs are connected to standard library inverter cells to ensure realistic drive strength and load, respectively. Table 2.1 provides additional details.

2.7.2 Delay

Contrary to popular wisdom, the synthesis results shown in Figure 2.6 indicate that the minimum cycle time at which a matrix arbiter with a given number of ports can operate exceeds that of both the round-robin arbiter and the prefix arbiter by 30 % in the five- and eight-port configurations. The state update logic represents the critical timing path in each case, and the matrix arbiter is penalized by the large fanout involved in updating the precedence matrix, as mentioned in Section 2.4.

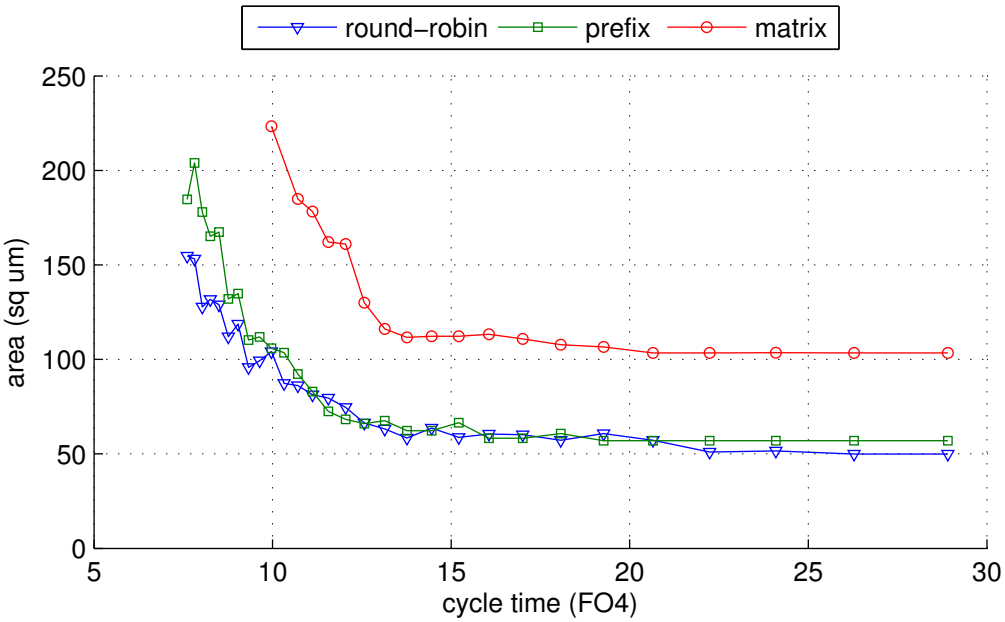
With ten ports, the minimum delay for the round-robin arbiter increases substantially, reducing the difference to the matrix arbiter to 3 %; in contrast, the prefix implementation increases its speed advantage even further at this design point.

2.7.3 Area

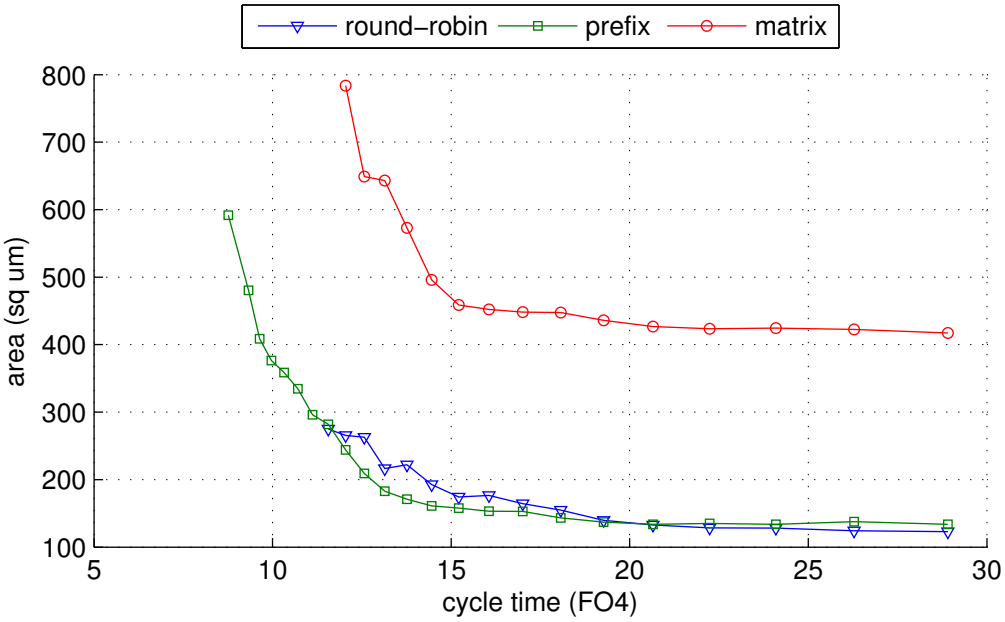
Figure 2.7 shows the trade-off between cell area and cycle time for the three types of arbiters. We omit detailed results for $n = 8$ for brevity; they are consistent with our observations for the two remaining design points.

At the beginning of the sweep, corresponding to the right side of the figure, all three designs are able to satisfy the timing requirements using minimum-size gates. As a result, area initially remains flat as frequency increases. The knee in each curve indicates the maximum operating frequency that a minimum-size implementation can support; as the target frequency increases beyond this point, synthesis must make use of larger standard cells with increased drive strength in order to be able to satisfy the tighter delay requirements, leading to a rapid increase in overall cell area.

Across both design points, the round-robin and prefix network arbiters both yield lower minimum delay and require less area. This difference is primarily a result of the relatively large number of registers required for implementing a matrix arbiter. Comparing the results for $n = 5$ and $n = 10$ at low clock frequencies clearly illustrates the differences in scaling behavior between the round-robin and prefix arbiters (linear scaling) on the one hand and the matrix arbiter (quadratic scaling) on the other hand. The difference in cell area between the round-robin and prefix arbiters is relatively small; the former produces slightly more compact designs at lower target frequencies



(a) $n = 5$.



(b) $n = 10$.

Figure 2.7: Area-delay trade-off curves for n-port arbiters.

and for small arbiters, while the latter becomes more efficient as the number of ports increases.

2.7.4 Power-Delay Product

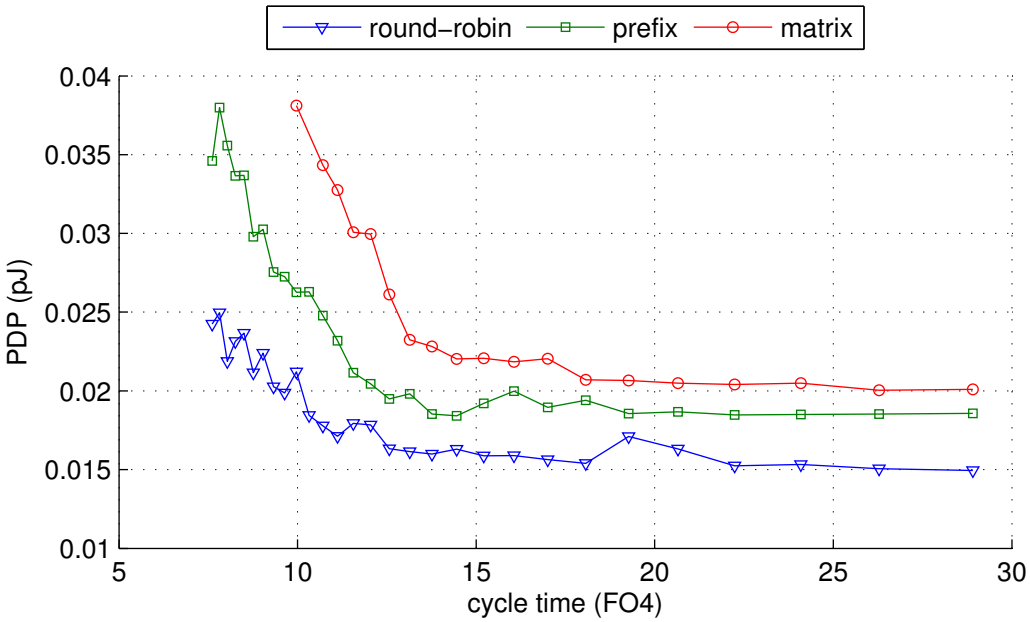
The PDP values, shown in Figure 2.8, largely mirror the results for cell area: Matrix arbiters consistently incur higher energy cost than both other types of arbiters, round-robin arbiters are more efficient for small configurations, and prefix arbiters become more efficient as the number of ports increases. However, compared to the area results, the differences between the round-robin and prefix arbiters are slightly more pronounced, while the differences between those two and the matrix arbiters have become less pronounced. Nevertheless, round-robin arbiters remain the most efficient choice for small configurations, and prefix network arbiters maintain their benefits for large configurations.

2.7.5 Multi-Priority Overhead

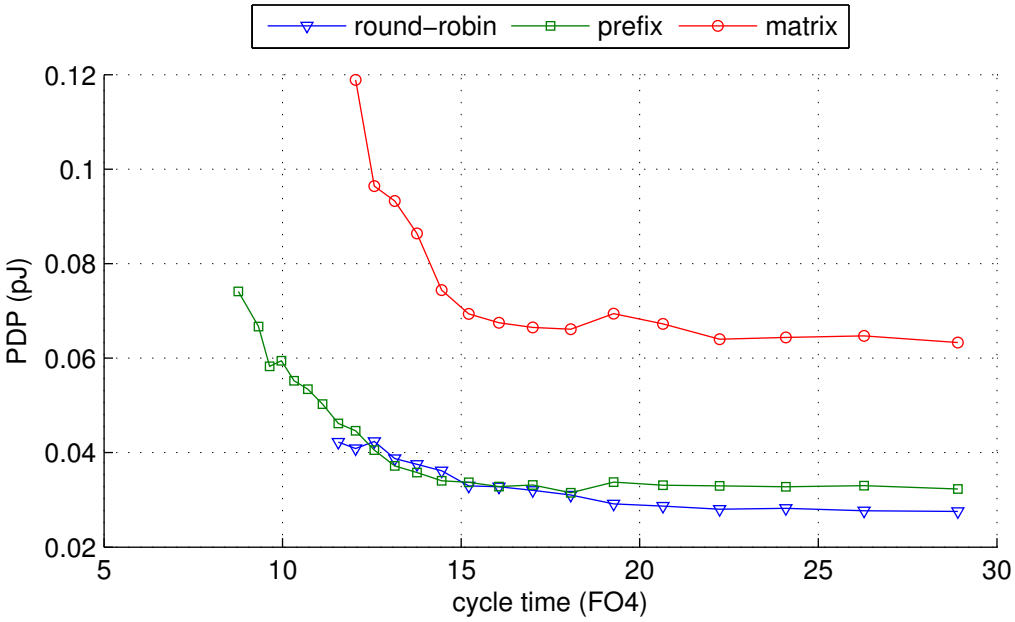
Figure 2.9 illustrates the impact that adding support for two priority levels as described in Section 2.6 has on the minimum cycle time. The white segment at the bottom of each bar corresponds to the minimum cycle time for the single-priority case from Figure 2.6.

The delay increase is most pronounced for the round-robin arbiter and least significant for the matrix arbiter; in the latter case, priority selection can be overlapped with existing logic to a large extent, minimizing the impact on the critical path. With ten inputs, this allows the matrix arbiter to achieve lower cycle times than the dual-priority round-robin arbiter. However, the prefix arbiter remains the fastest implementation across all configurations.

While both area and PDP increase in magnitude when adding dual-priority support, the overall trends remain the same as with a single priority level, with the difference in efficiency between matrix arbiters on the one hand and round robin and prefix arbiters on the other hand decreasing slightly for the larger configurations. We omit the corresponding figures in the interest of brevity.



(a) $n = 5$.



(b) $n = 10$.

Figure 2.8: Power-delay-product for n-port arbiters.

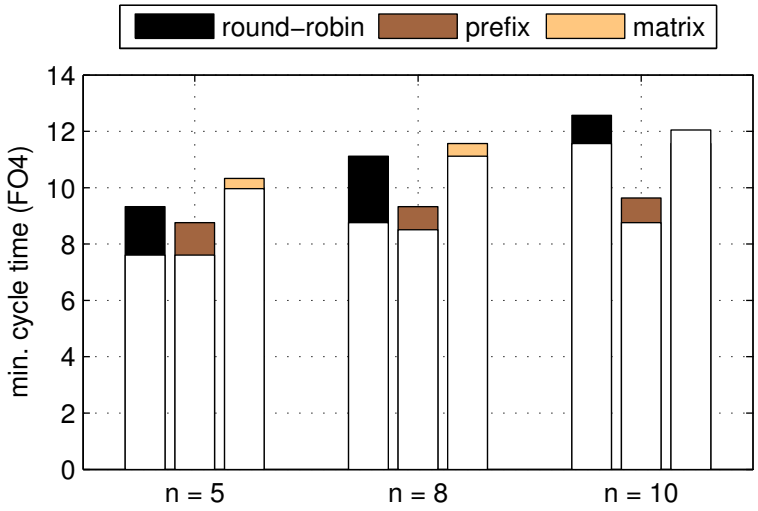


Figure 2.9: Cycle time penalty for implementing dual-priority support.

2.8 Related Work

Pankaj and McKeown [71] provide detailed block-level descriptions of basic arbiter implementations used in the context of the *Tiny Tera* research project. Huang et al. [41] describe fast full-custom implementations. Preußer et al. [79] develop techniques for using high-speed adder designs to implement arbitration; this approach is of particular benefit for applications that target Field-Programmable Gate Arrayss (FPGAs), which often provide delay-optimized adder blocks that yield better performance than general-purpose programmable logic. Dimitrakopoulos et al. [23] propose a fast arbiter design based on prefix networks. Lee at al. [51] develop a multi-priority arbiter that treats priorities in a *probabilistic*—rather than absolute—manner; i.e., priorities determine the statistical distribution of grants instead of imposing a strict order among requests. Their design has applications in improving global fairness in the network. Finally, Shin et al. [86] present a tool that can generate synthesizable arbiter implementations with arbitrary numbers of inputs.

2.9 Summary

In this chapter, we have presented a brief overview of arbitration and provided detailed descriptions of representative arbiter implementations, which will we will use as building blocks for more complex structures in subsequent chapters. A comparison of standard-cell arbiter implementations in terms of delay, area and energy efficiency shows that—contrary to popular wisdom—matrix arbiters are both less efficient and slower than round-robin arbiters at sizes commonly encountered in NoC routers. Throughout the remainder of this dissertation, we will therefore consider the latter type of arbiter exclusively.

Chapter 3

Allocators

3.1 Overview

Where arbitration addresses the problem of coordinating access to a single shared resource between multiple competing agents, *allocation* extends the problem to situations where agents compete for multiple resources simultaneously. Specifically, given a set of resources and a set of agents, each of which can request access to one or more of the former, an *allocator* grants resources to agents subject to three basic constraints[1]:

- Resources are only granted to agents that requested them.
- Each agent is granted access to at most one resource.
- Each resource is granted to at most one agent.

Similar to the approach taken in Chapter 2, we can thus formally describe allocation as an operation Ψ on a two-dimensional binary request matrix $R = \{r_{i,j}\}$ that produces

[1] We also refer to the agents as the allocator's inputs, and to the resources as its outputs; both terms are used interchangeably in the remainder of this dissertation.

a grant matrix $G = \{g_{i,j}\}$ with the following properties:

$$G = \Psi(R) \tag{3.1}$$

$$g_{i,j} \Rightarrow r_{i,j} \tag{3.2}$$

$$g_{i,j} \Rightarrow \bigwedge_{k \neq j} (\neg g_{i,k}) \tag{3.3}$$

$$g_{i,j} \Rightarrow \bigwedge_{k \neq i} (\neg g_{k,j}) \tag{3.4}$$

Any grant matrix that satisfies these constraints is called a *matching*. The *cardinality* of a matching is the number of generated grants; it represents a metric for matching quality. We refer to matchings in which no further resource can be granted except by replacing another existing grant as *maximal*. Among these, the matchings with the highest possible cardinality are called *maximum* matchings.

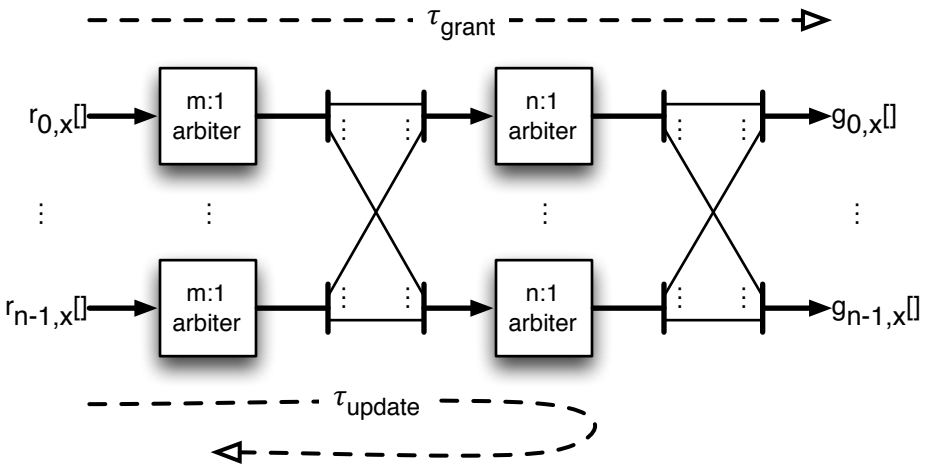
In order to maximize resource utilization, it is desirable for an allocator to produce matchings with the highest possible cardinality. In practice, however, there is a trade-off between matching quality on the one hand and delay, area and power constraints that limit the allocator's logic complexity on the other hand. The remainder of this section discusses allocator implementations that represent different trade-offs between these two qualities.

3.2 Separable Allocators

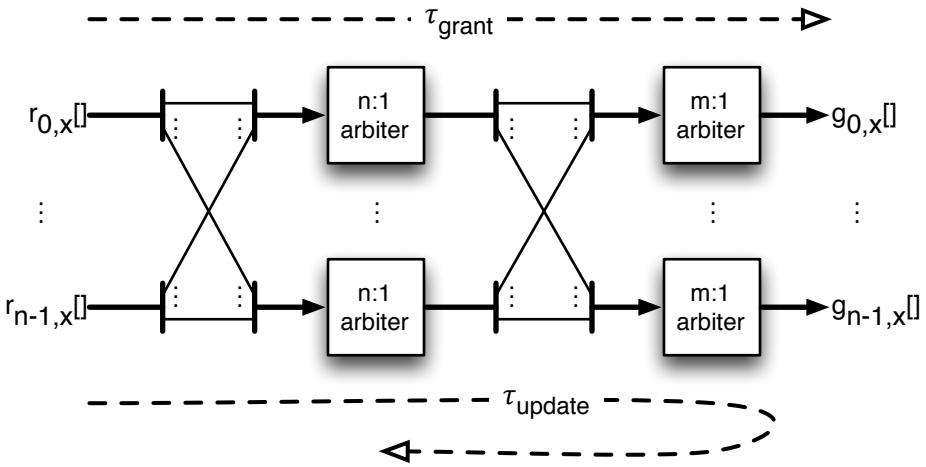
A separable allocator generates a matching by decomposing allocation into two successive phases of arbitration. Formally, each phase corresponds to applying an arbitration function Φ to each row or column of the request matrix.

For *separable input-first allocation*, as shown in Figure 3.1a, each agent independently selects a single resource to request in the first round. A second round of arbitration is then performed at each resource to select a winner among all incoming requests.

In contrast, for *separable output-first allocation*, shown in Figure 3.1b, agents eagerly forward all of their requests to the associated resources. The latter again

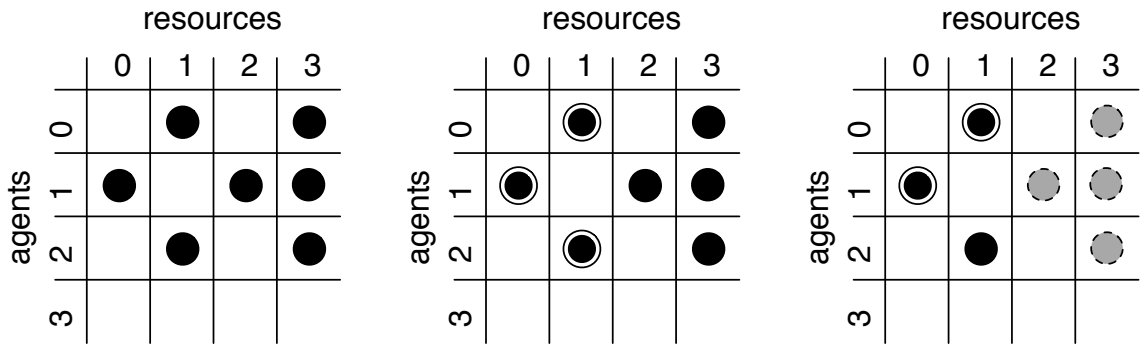


(a) Input-first.



(b) Output-first.

Figure 3.1: Separable allocators with n inputs and m outputs.



(a) Requests. (b) Input grants. (c) Output grants.

Figure 3.2: Example of inefficiency in separable input-first allocation.

perform arbitration among all incoming requests and send grants back to the winning agents. As multiple resources may select the same winning agent in this stage, a second round of arbitration is required in which each agent chooses a winner among all resources that were granted to it in the first stage. Compared to input-first allocation, this incurs additional propagation delay on the timing arc for arbiter state updates as shown in Figure 3.1[2].

Arbiters—and therefore separable allocators—can be designed such that their delay scales approximately logarithmically with the number of inputs, enabling relatively fast allocation even for high-radix routers. However, because the arbiters in each stage make arbitration decisions independently from one another, multiple arbiters in the first stage can select requests for the same second-stage arbiter from their respective set of available requests, resulting in a non-maximal matching. Figure 3.2 shows an example for separable input-first allocation: Agent 0 and agent 2 both independently select resource 1 in the first arbitration stage, leading to a conflict in the second arbitration stage. Resource 1 can only satisfy one of the two requests, leaving agent 2 unassigned even though it could have used resource 3.

We can reduce the likelihood of such allocation inefficiencies—and consequently increase matching quality—by staggering arbiter priorities using the *iSLIP* approach

[2] The timing paths for τ_{update} shown in Figure 3.1a and Figure 3.1b assume that updates are performed according to the *iSLIP* scheme described later in this section.

described in [54]; to this end, we only update the priority state for any given arbiter in the first stage if it produced a request that subsequently resulted in a grant in the second arbitration stage. Further improvements in matching quality can be achieved by performing multiple iterations of separable allocation; however, tight delay constraints typically make this iterative approach unattractive in the context of Networks-on-Chip (NoCs).

3.3 Wavefront Allocators

Wavefront allocators [89] take advantage of the fact that, by construction, no two entries on any given diagonal in the request matrix[3] share the same row or column; consequently, all requests on a diagonal can be granted independently of one another. This allows us to compute a maximal matching by first granting all requests on a highest-priority diagonal, eliminating any additional requests in the same rows or columns as the granted requests, and then repeating this process for all remaining diagonals in the request matrix.

Assuming that the diagonals are traversed in linear order, a wavefront allocator can be implemented as a regular array of simple bit cells as shown in Figure 3.3. This facilitates efficient full-custom implementation [21] with area and delay that scales quadratically and approximately linearly with the number of ports, respectively.

Because each row and column of the wavefront array must contain one element of the selected priority diagonal, wavefront allocators are inherently square. If the number of agents and resources differs, a wavefront allocator of sufficient size to accommodate the larger of the two must be used, with the request inputs for any unused bit cells tied to zero.

3.3.1 Fairness

Fairness in wavefront allocators can be controlled by managing the order in which the highest-priority diagonal is selected at the beginning of each round of allocation.

[3] Each diagonal corresponds to the set of requests $r_{i,j}$ for which $(i+j) \bmod n$ has the same value; i.e., diagonals wrap around at the edges of the matrix.

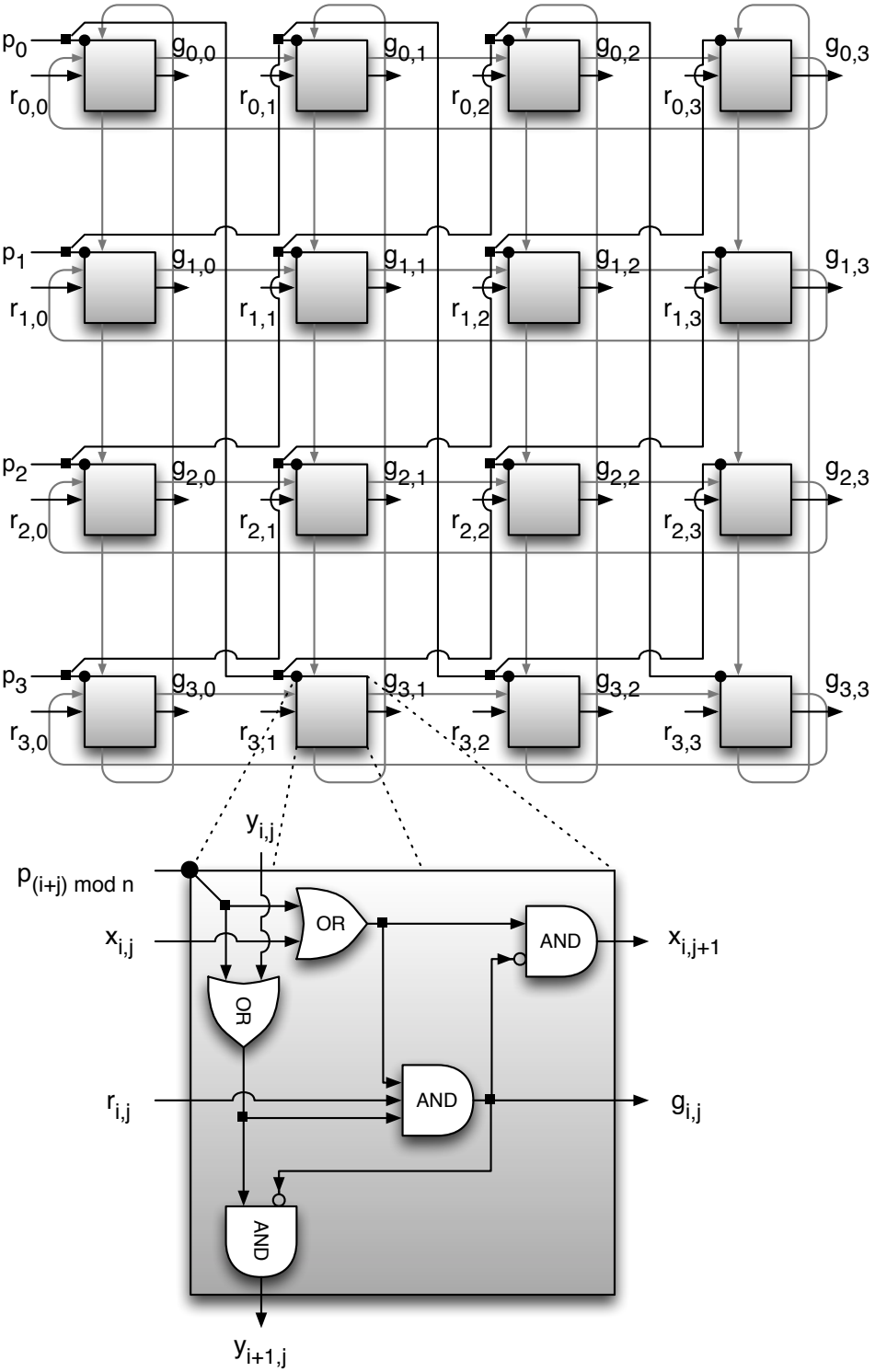
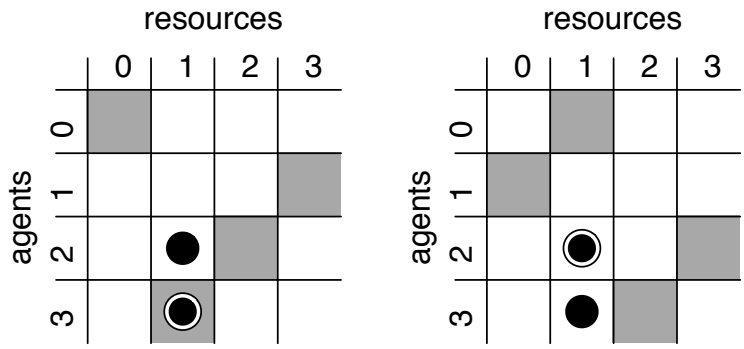
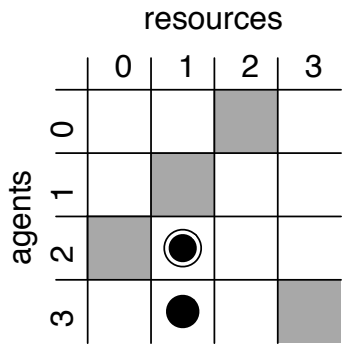


Figure 3.3: Implementation of a wavefront allocator with $n = 4$ ports.

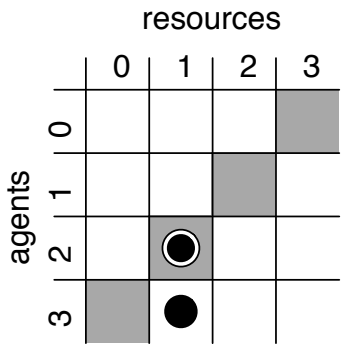


(a) $P = \{1, 0, 0, 0\}$.

(b) $P = \{0, 1, 0, 0\}$.



(c) $P = \{0, 0, 1, 0\}$.



(d) $P = \{0, 0, 0, 1\}$.

Figure 3.4: Circular priority diagonal selection results in uneven distribution of grants.

Existing designs commonly implement weak fairness by connecting the priority select signals $P = \{p_0, p_1, \ldots, p_{n-1}\}$ to a circular shift register in which a single bit is set. This approach ensures that any given request will be granted after at most n cycles; however, it can lead to unfair allocation of resources, as the example in Figure 3.4 shows: Because diagonals are evaluated in ascending order, three out of four possible choices for the starting diagonal—indicated by shaded cells—lead to request $r_{2,1}$ being granted, while request $r_{3,1}$ is only granted in the one remaining case where the diagonal it is on has highest priority. Thus, on average, $r_{2,1}$ is granted three times as often as $r_{3,1}$.

To facilitate more balanced allocation of resources, we modify the canonical design such that the starting diagonal for the next cycle is selected based on the grants that were generated in the current cycle. Specifically, we extend the priority update mechanism for round-robin arbiters described in Section 2.3 from grant vectors to grant matrices: Whenever one or more grants are produced, the successor of the highest-priority diagonal that had any requests—and, by extension, grants—in the current cycle becomes the starting diagonal in the next cycle. While this is not sufficient to guarantee strong fairness—in particular, if there are requests from all inputs or for all outputs, the behavior is the same as in the canonical implementation—, it avoids pathological behavior when a small number of agents compete for the same resource.

The proposed scheme is readily implemented in hardware by performing an OR reduction across the diagonals of the request matrix, feeding the resulting vector into a round-robin arbiter and connecting the latter's state variable to the wavefront array's priority select inputs p_k. As the starting diagonal for the next cycle depends only on the request matrix and the starting diagonal for the current cycle, the update can be performed in parallel with the actual allocation and consequently does not extend the allocator's critical path.

3.3.2 Acyclic Implementations

As in the case of round-robin arbiters, combinational loops—formed by signals $x_{i,j}$ and $y_{i,j}$ in Figure 3.3—can interfere with the ability of commercially available tools

to properly perform static timing analysis. In order to facilitate the use of wavefront allocation with synthesis-based design flows, we develop acyclic alternatives to the canonical design described in [89].

We can eliminate combinational loops by unrolling the wavefront array as we did for linear round-robin arbiters in Section 2.3. Figure 3.5 shows the arrangement of the replicated cells; their grant outputs are ORed with those of the corresponding original cells. The one-hot priority signals p_k logically divide the extended array into multiple sub-arrays:

- Any part of the extended array above the first active priority diagonal is effectively disabled as the $x_{i,j}$ and $y_{i,j}$ signals for each cell (cf. Figure 3.3) are deasserted.

- The first activated priority diagonal marks the beginning of the sub-array in which grants are generated. This sub-array is equivalent to the canonical wavefront array for the current value of the priority signals $\{p_k\}$ and extends for a total of n successive diagonals.

- Any remaining diagonals at the bottom of the extended array, starting with the second instance of the selected priority diagonal, are also effectively inactive: Because they replicate earlier active diagonals, any requests in this region of the array are guaranteed to be masked by prior grants.

Hurt et al. [43] describe a similar implementation that uses a sliding window of enable signals to activate a subset of the unrolled array explicitly. All in all, unrolling the wavefront array roughly doubles area, power and critical path delay compared to the canonical implementation.

An alternative approach for eliminating the combinational loops takes advantage of the fact that each each of them is *logically* severed at the currently selected priority diagonal: In the basic cell shown in Figure 3.3, if $p_{(i+j) \bmod n}$ is asserted, the two OR gates mask the values of x_i and y_i. Hence, for each individual priority selection, we can construct an equivalent loop-free wavefront array as shown in Figure 3.6. We can use these fixed-priority equivalents to implement an acyclic wavefront allocator.

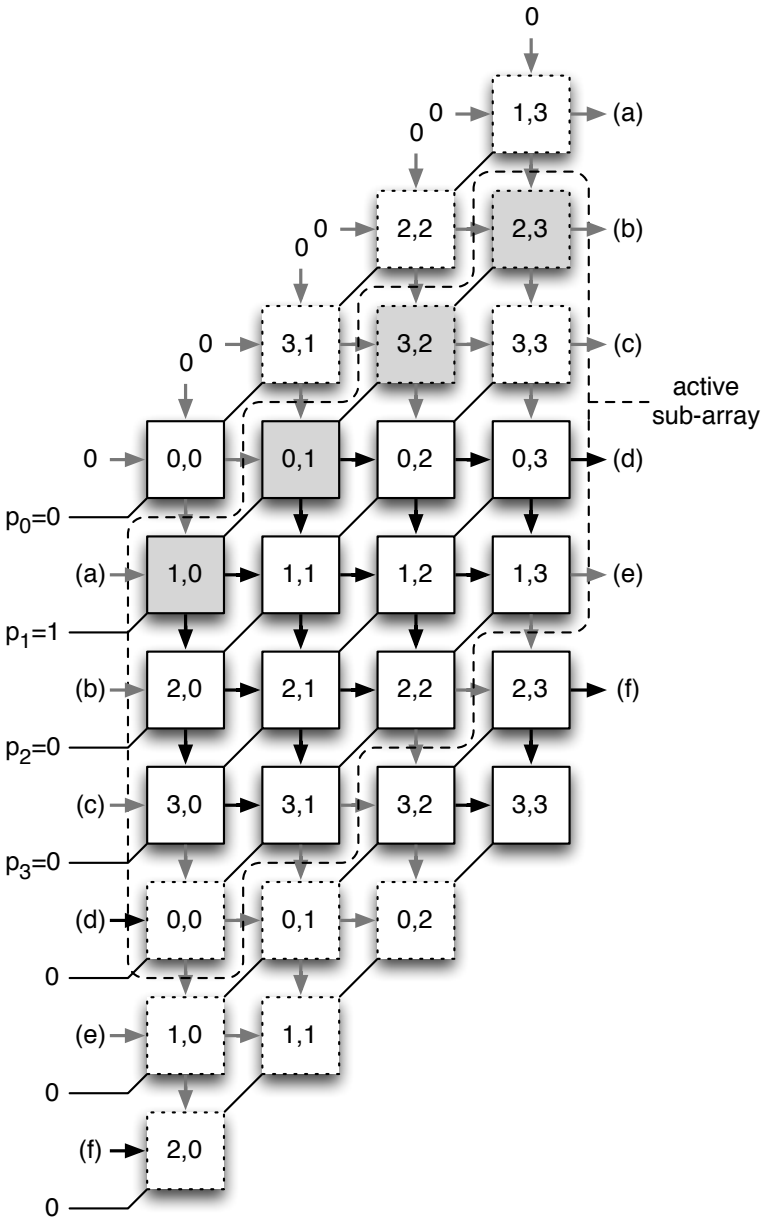
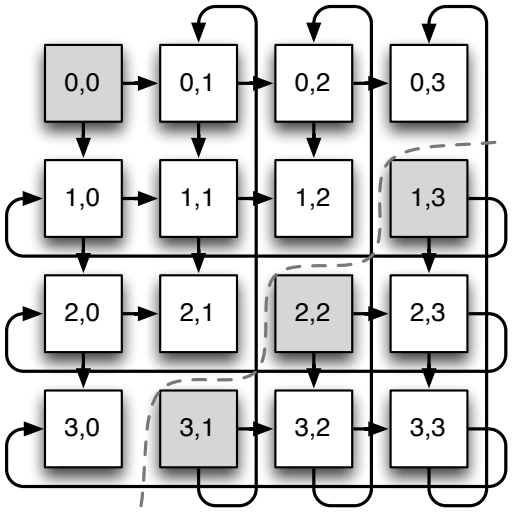
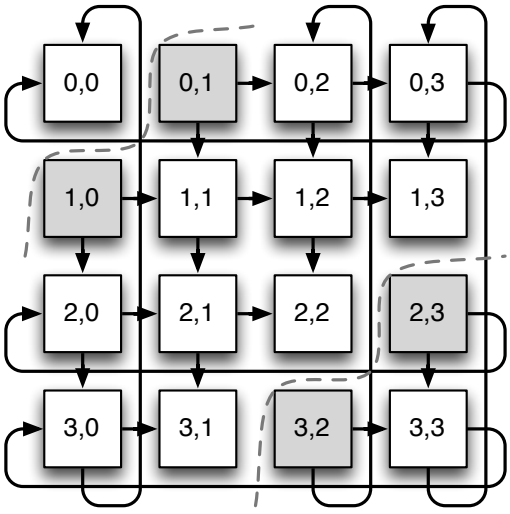


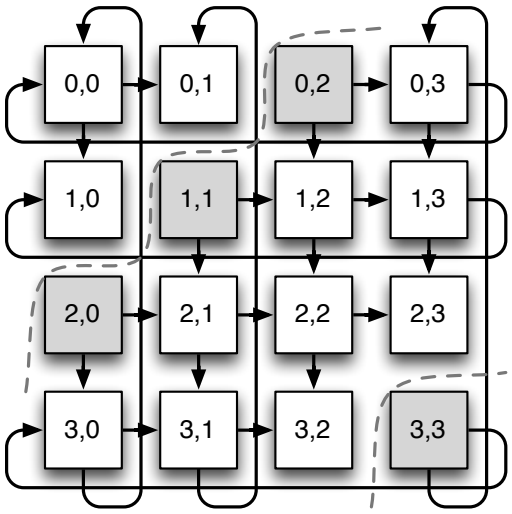
Figure 3.5: Acyclic wavefront allocator using unrolling with $n = 4$ ports.



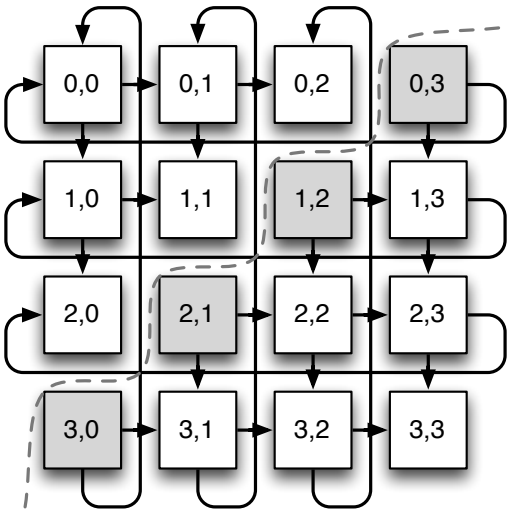
(a) $P = \{1, 0, 0, 0\}$.



(b) $P = \{0, 1, 0, 0\}$.



(c) $P = \{0, 0, 1, 0\}$.



(d) $P = \{0, 0, 0, 1\}$.

Figure 3.6: Loop-free equivalent wavefront arrays for individual priority selections.

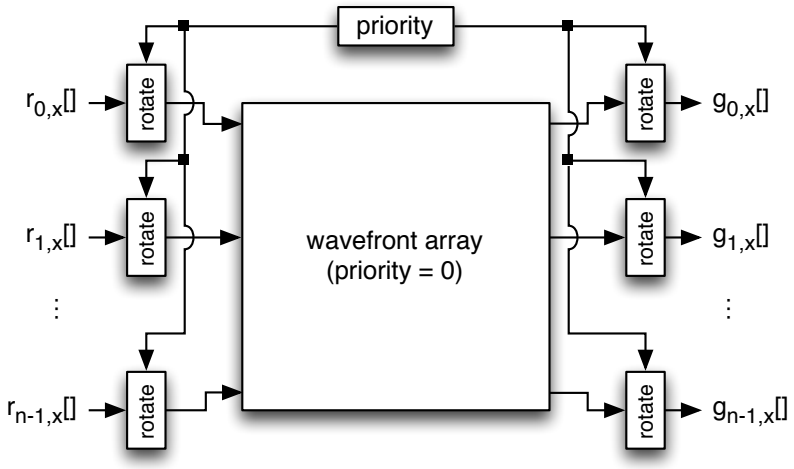


Figure 3.7: Acyclic wavefront allocator using input/output transformation.

The symmetry of the wavefront array implies that the equivalent array for a given priority k can be derived from that for priority 0 by rotating either each row or each column of the request matrix by k bit positions. Thus, we can construct an acyclic wavefront allocator using a fixed-priority wavefront array and a set of barrel shifters at the inputs and outputs as shown in Figure 3.7. This eliminates the combinational loops at the cost of increased delay.

Alternatively, an acyclic implementation can be constructed by applying Shannon decomposition based on the priority selection signals p_k to the canonical implementation. In this case, the loop-free implementation—as shown in Figure 3.8—computes the resulting grant matrix for each possible priority level in parallel using a set of fixed-priority wavefront arrays; a set of multiplexers then selects the correct intermediate result based on the actual priority value. The replication of wavefront arrays results in cubic scaling behavior for area and power, rendering this approach unattractive for large allocators; however, for allocators with a relatively small number of ports, its comparatively low delay can outweigh the cost differential compared to other implementation alternatives.

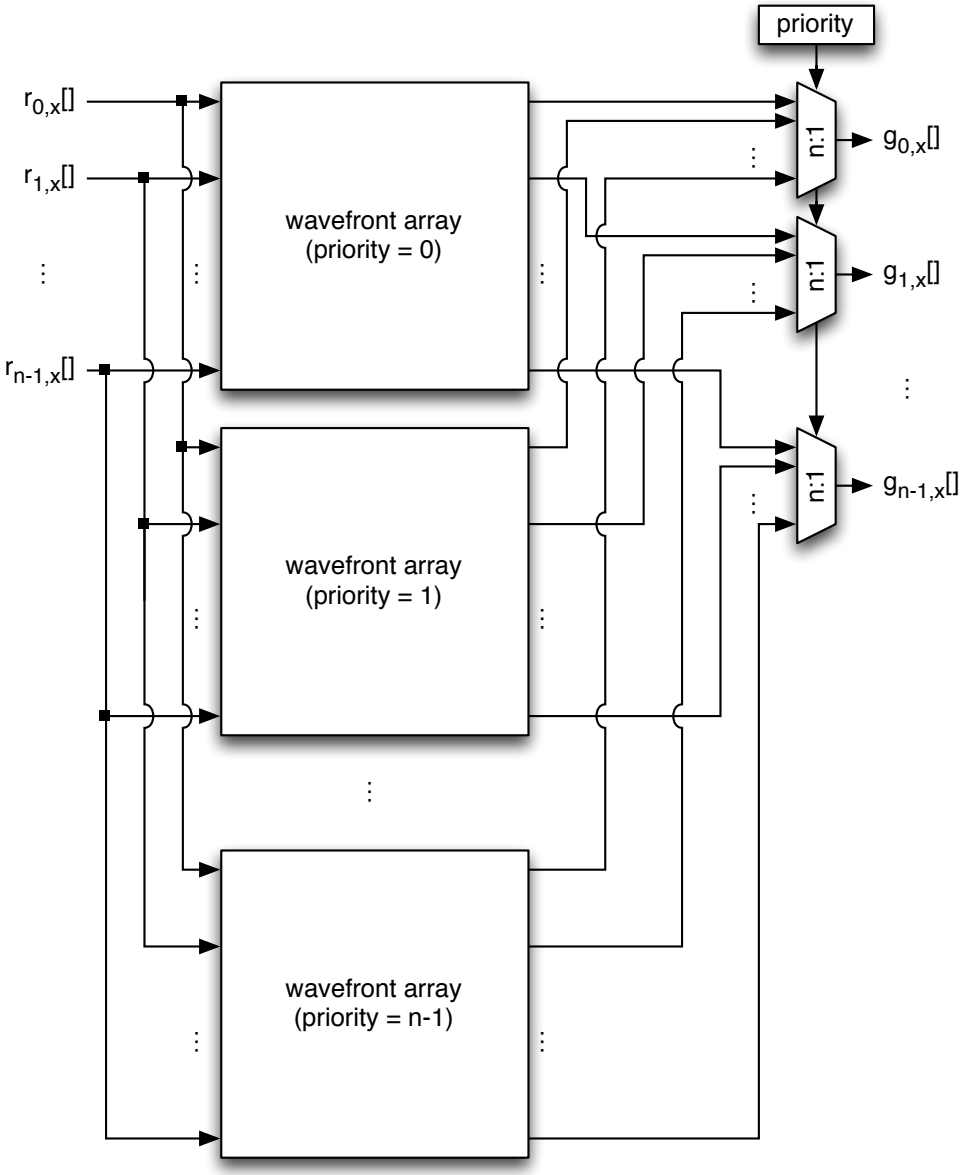
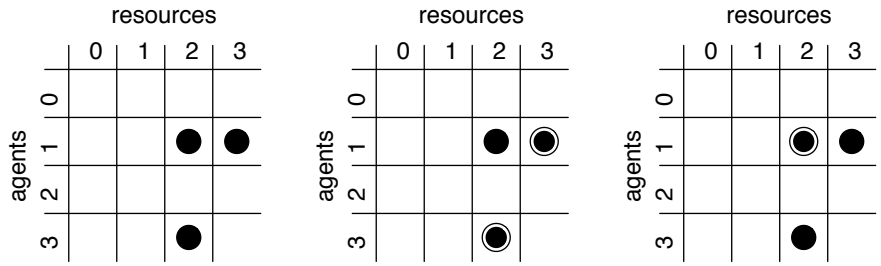


Figure 3.8: Acyclic wavefront allocator using replication.



(a) Requests. (b) Maximum matching. (c) Suppressed matching.

Figure 3.9: Example of starvation in maximum-size allocation.

3.4 Maximum-Size Allocation

Conceptually, a maximum matching for a given set of requests and resources is readily found by performing successive iterations of an augmenting path algorithm [26]. However, while hardware implementations have been proposed that can perform one such augmentation step in each cycle [38], the associated complexity as well as the inherently iterative nature of generating a complete matching in this fashion limit their applicability to NoC routers.

Furthermore, given a request matrix, a maximum-size allocator will never generate a grant that is not part of any maximum matching; as a result, it fails to satisfy the elementary fairness property that every request eventually be granted. Figure 3.9 shows an example of a request matrix with two possible maximal matchings that differ in cardinality. In this scenario, a maximum-size allocator will always produce the higher-cardinality matching shown in Figure 3.9b, causing the remaining request to block indefinitely.

However, despite its implementation complexity and susceptibility to starvation, maximum-size allocation provides a useful upper bound on matching quality that other allocators can be benchmarked against.

3.5 Evaluation

For separable allocators, implementation trade-offs are largely determined by the characteristics of their constituent arbiters, which we evaluated in Chapter 2. In the present section, we compare the delay, area and energy efficiency of different wavefront allocator implementations. Specifically, we consider the following designs:

Unroll: An acyclic wavefront allocator using unrolling as shown in Figure 3.5.

Rotate: An implementation using input/output transformation (cf. Figure 3.7).

Replicate: An implementation using replicated wavefront arrays (cf. Figure 3.8).

DPA: A Diagonal Propagation Arbiter[4] as described in [43].

3.5.1 Experimental Setup

We conduct our evaluation using the same experimental setup that was used to compare arbiter implementations in Section 2.7. However, due to the greater logic complexity of wavefront allocators compared to individual arbiters, we begin each sweep at a lower base frequency of 500 MHz.

3.5.2 Delay

Figure 3.10 compares the minimum cycle time that each implementation alternative can operate at for three exemplary allocator sizes. Due to the expansion of the wavefront block, cycle time grows more quickly for the Diagonal Propagation Arbiter (DPA) and the unrolled implementation as the number of ports increases than for the two other implementations. With $n = 5$ ports, the additional overhead for transforming inputs and outputs increases the delay for the rotation-based allocator by 11 % compared to these two implementations; however, its more benign scaling behavior allows it to operate at lower cycle times for configurations with eight or more ports. Despite the overhead associated with creating multiple instances of the

[4] Note that, contrary to its name, the DPA is actually an allocator, not an arbiter.

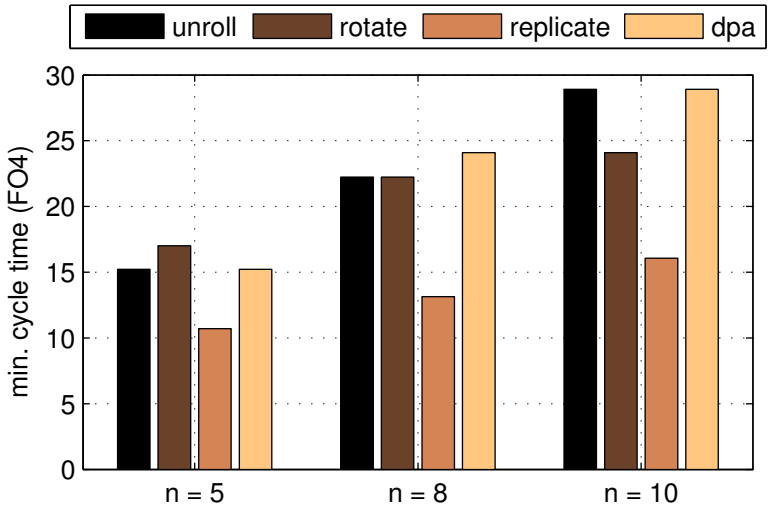


Figure 3.10: Minimum cycle time for n-port wavefront allocators.

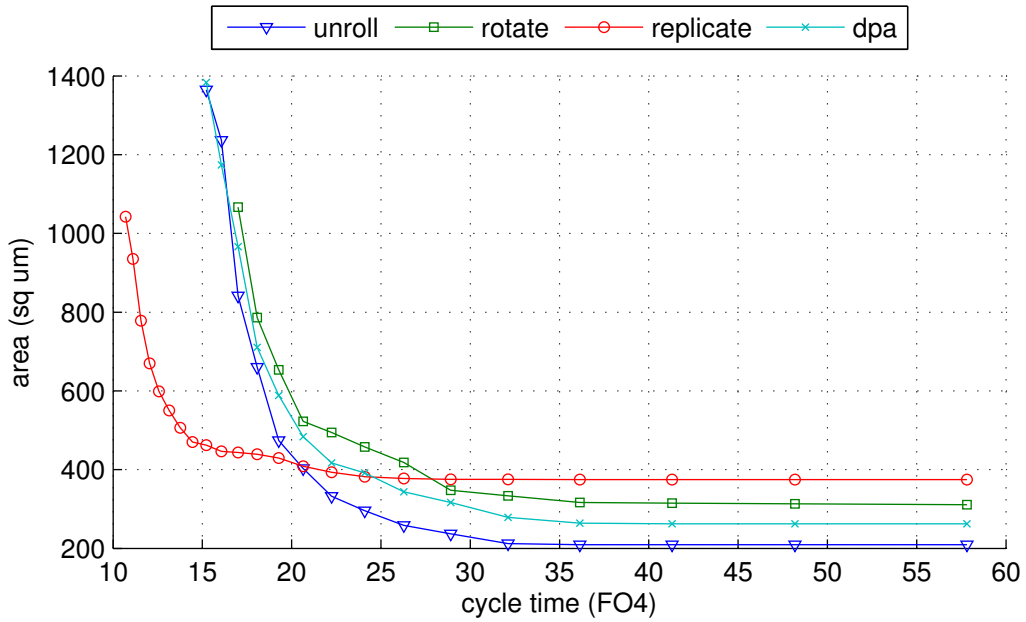
central wavefront array, the replication-based implementation is the fastest across all three design points, outperforming the closest competitor in each case by 30–41 %.

3.5.3 Area

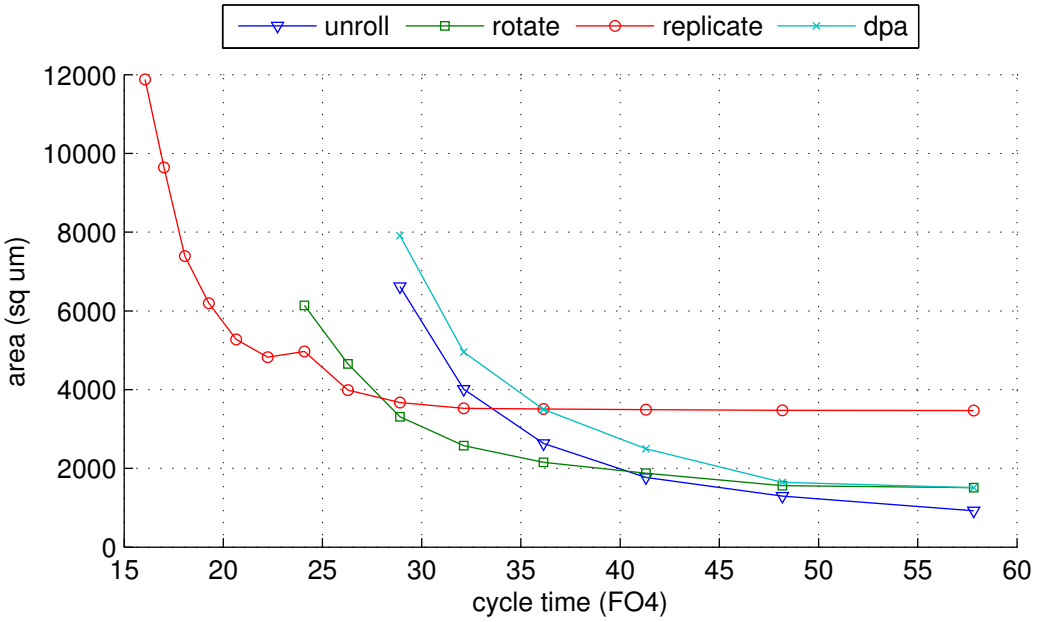
Figure 3.11 shows the area-delay trade-off for the individual wavefront allocator variants; as in Chapter 2, we omit the results for the intermediate configuration ($n = 8$) for brevity.

With five ports, the unrolled design illustrated in Figure 3.5 yields the best area efficiency among all four designs for cycle times above 20 FO4. While the cost inherent in instantiating multiple wavefront arrays causes the replication-based approach to be less area-efficient for cycle times in excess of 28 FO4, the situation is reversed at the opposite end of the delay spectrum as the higher delay of the other implementation variants necessitates the use of increasingly larger gate sizes in order to meet timing constraints.

For the ten-port configuration, the cubic scaling behavior of the replication-based implementation substantially increases its base cost at low target frequencies. The



(a) $n = 5$.



(b) $n = 10$.

Figure 3.11: Area-delay trade-off curves for n-port wavefront allocators.

unrolled implementation continues to require the least area at low operating frequencies; however, due to its lower minimum delay, the rotation-based implementation represents the most area-efficient choice for target cycle times between 28 FO4 and 40 FO4. As in the five-port case, the replication-based implementation achieves the lowest delay overall and thus represents the best trade-off at high target frequencies.

For either design point, at any given target cycle time, at least one of the wavefront implementations described in in Section 3.3 provides better area efficiency than the previously proposed implementation from [43].

3.5.4 Power-Delay Product

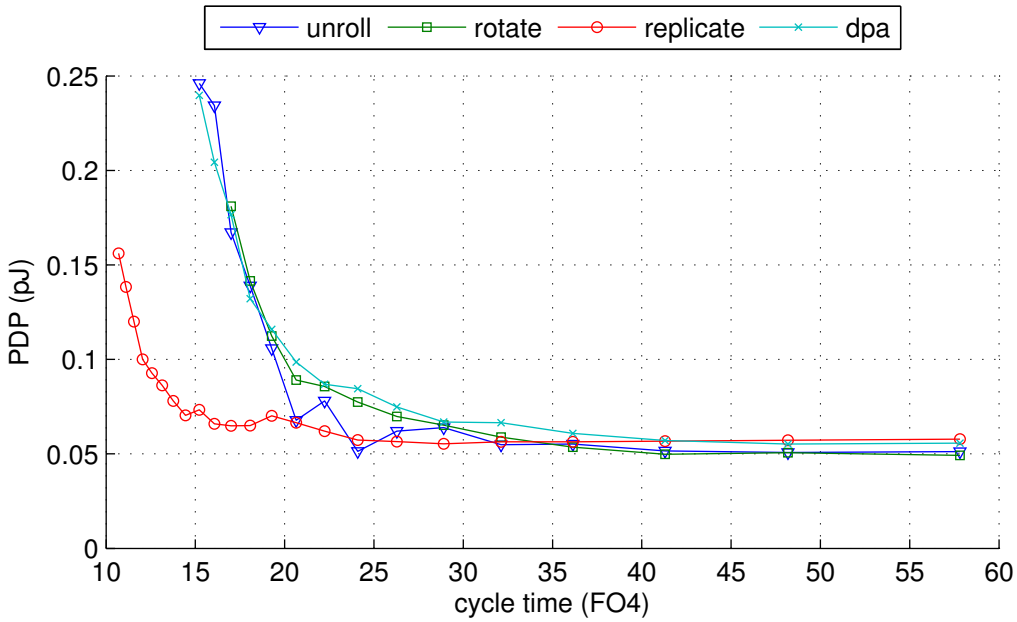
Results for energy efficiency, shown in Figure 3.12, follow the same overall trends as those for area, with slightly shifted transition points and less pronounced differences between implementations at low target frequencies.

3.5.5 Fairness Overhead

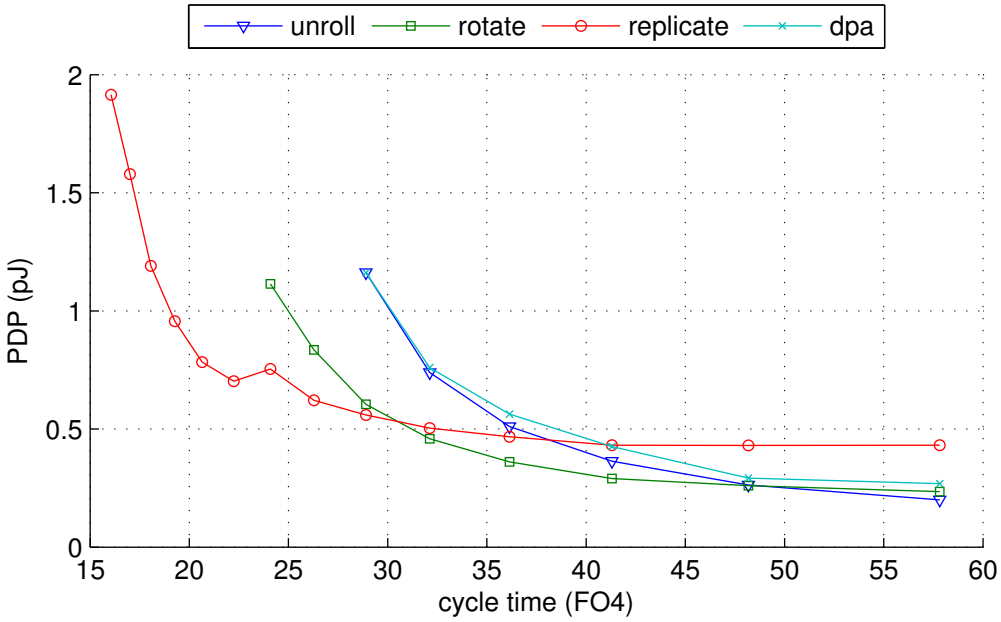
Additional synthesis runs, the detailed results of which we omit for brevity, show that implementing the fair priority diagonal selection scheme described in Section 3.3.1 does not measurably affect critical path delay and has minimal impact on area and energy efficiency.

3.6 Related Work

Wavefront allocators were first developed by Tamir and Chi [89]. Gopalakrishnan [29] develops an asynchronous implementation of a wavefront allocator using micropipelining principles; this design avoids clock distribution and synchronization overhead at the cost of increased design and verification complexity. Hurt et al. [43] present wavefront allocator implementations that avoid combinational loops, including the DPA. Delgado-Frias and Ratanpal [21], on the other hand, describe efficient full-custom VLSI implementations that do include combinational cycles. Tian et al. [93] describe an alternative scheme for avoiding fairness issues in wavefront allocators. Olesinski



(a) $n = 5$.



(b) $n = 10$.

Figure 3.12: Power-delay-product for n-port wavefront allocators.

et al. [67] propose a variation of a wavefront allocator suitable for use in very large switches; however, the proposed design incurs high latency under light load. A follow-on paper [68] remedies this deficiency by using a fast secondary allocator to back-fill generated grant matrices.

3.7 Summary

In the present chapter, we have discussed the key aspects of allocation and provided detailed descriptions of representative hardware implementations. In doing so, we have examined allocation inefficiencies in separable allocators, fairness issues in wavefront allocators and starvation scenarios in maximum-size allocation. Furthermore, we have investigated approaches for eliminating combinational loops in wavefront allocators to facilitate their use in synthesis-based design flows. Experimental results for common NoC design points indicate that our designs compare favorably to a state-of-the-art implementation of an acyclic wavefront allocator, improving delay, area and energy efficiency.

Chapter 4

Virtual Channel Allocation

4.1 Overview

In Virtual Channel (VC) flow control [17], when the head flit of a packet arrives at a router, it must acquire one of the VCs associated with the physical channel that connects to its destination output before it can proceed. To achieve this, the head flit sends a request to the *VC allocator* once it reaches the front of its input VC. The VC allocator generates a matching between any such requests from the input VCs on the one hand and those output VCs that are not currently in use by another packet on the other hand.

In the general case, a router with P ports and V VCs per port therefore requires a VC allocator that can match $P \times V$ agents (all input VCs at all input ports) to $P \times V$ resources (all output VCs at all output ports). However, in practice, the range of output VCs that any given packet can request is typically subject to additional constraints. In particular, many commonly used routing functions in Networks-on-Chip (NoCs) return only a single output for any given packet, limiting the set of candidate output VCs to those at its selected destination port. We will assume that the routing function is restricted in this way throughout the remainder of this chapter; specifically, we assume that the set of allowable output VCs for a given packet is represented by two signals generated by the routing logic: One that that selects its destination port and second one that carries a bit vector of candidate VCs

at the selected port. This motivates several modifications compared to a canonical $P\times V$-by-$P\times V$ allocator implementation according to Chapter 3.

In response to successful allocation, each granted output VCs is marked as being in use, and its state is updated to reflect which input and VC it is currently assigned to. At the same time, each winning input VCs updates its state to indicate that VC allocation has been completed and stores the assigned output VC in a register.

4.2 Implementation

We can implement VC allocation using separable allocators as described in Section 3.2. Compared to the canonical designs, a separable VC allocator requires additional logic for generating requests and grants, and its input stage is slightly simplified as a result of the restricted routing function.

In the input-first implementation, shown in Figure 4.1, since the destination port is known, we can avoid $P\times V$-input arbitration in the input stage and instead simply select among the V candidate output VCs specified by the routing function; a demultiplexer can then be used to expand the result into a $P\times V$-wide vector of output-side requests. However, only those output VCs that are not currently in use by another packet should be considered in this arbitration step. To this end, a multiplexer selects the availability signals for the VCs at the destination port, and the resulting bit vector is used to mask the candidate VCs ahead of input-side arbitration. As the availability signals originate at the individual output VCs, this masking step increases the delay $\tau_{grant,in}$ for generating the final input VC grant signal as shown in Figure 4.1.

Output-side arbitration is implemented using two-level tree arbiters rather than monolithic $P\times V$-input arbiters, as the former distribute grants more fairly across input ports. A given output VC can be marked as unavailable if it receives requests from any input VCs, as one of these requests will result in a grant[1].

The results of output-side arbitration are then distributed back to the inputs VCs. Because a given input VC only issues requests to a single output port, it is safe to combine the incoming grant signals from all output ports using a P-input OR gate;

[1] We assume here that the arbiters satisfy the property described in Equation 2.4 in Chapter 2.

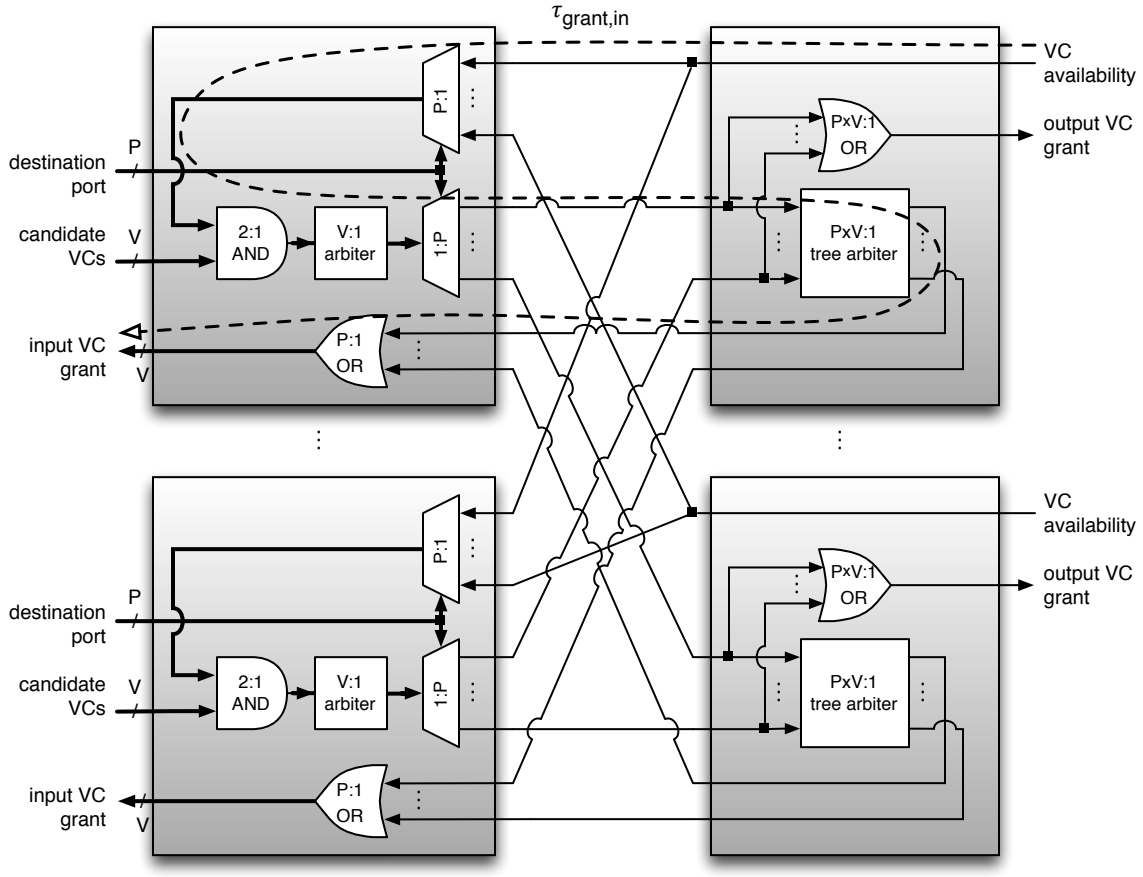


Figure 4.1: Separable input-first VC allocator.

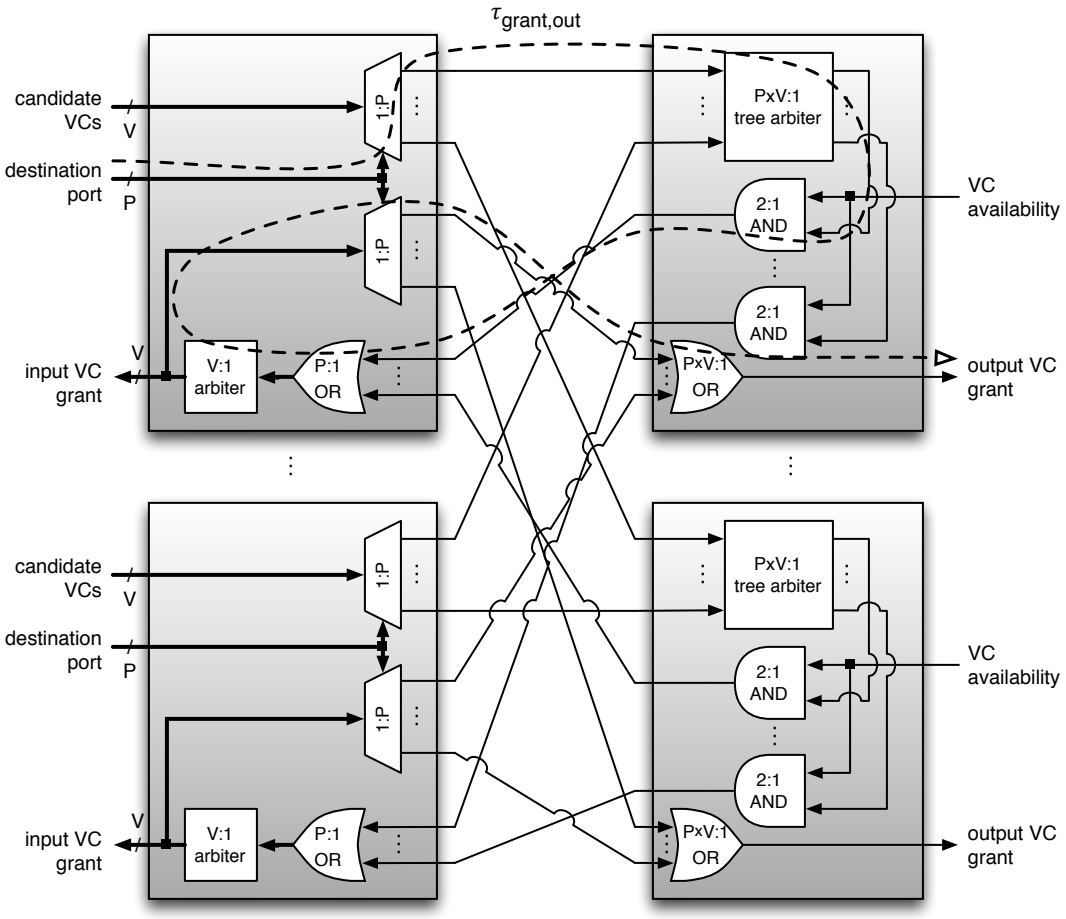


Figure 4.2: Separable output-first VC allocator.

the result is a bit vector that indicates the granted output VC—if any—at the selected destination port.

The output-first implementation variant is depicted in Figure 4.2. Here, each input VC sends requests to all candidate output VCs at the selected destination port, where arbitration is performed as in the input-first case. Because selection among output VCs is not performed until after output-side arbitration, we can account for VCs that are unavailable simply by masking their output arbiters' grant signals.

At the input side, we can combine the grants from the different output ports using a P-input OR as in the input-first case. However, as a given input VC may receive

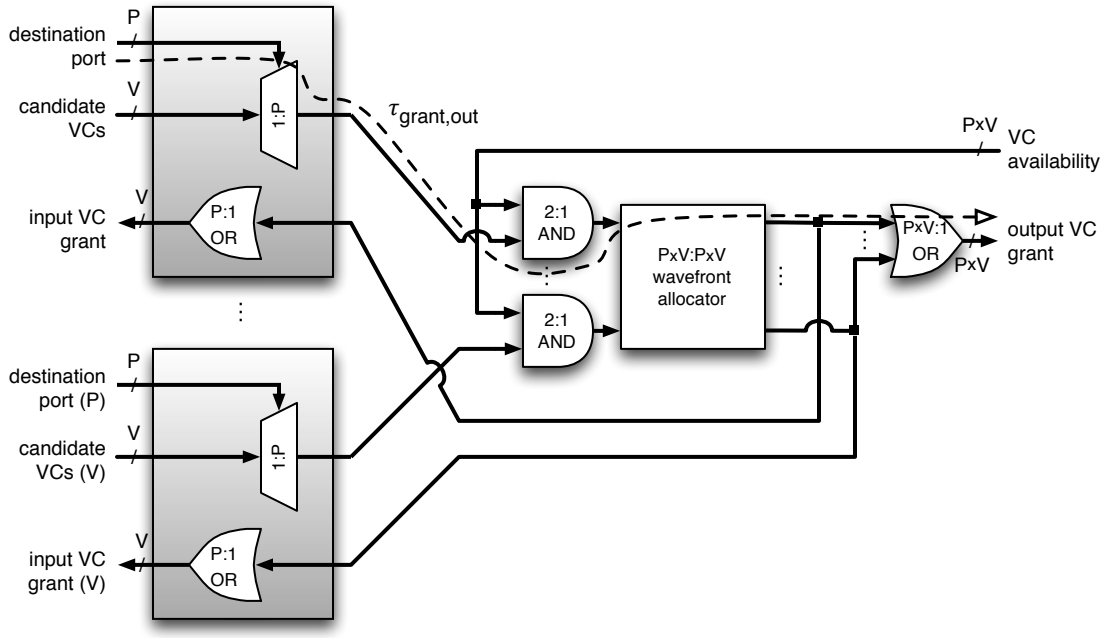


Figure 4.3: Wavefront-based VC allocator.

grants from multiple output VCs at its destination port, an additional arbitration stage is needed to select a winner among them.

Because output VCs that were allocated must be marked as unavailable, the final input-side grant signals must be propagated back to the output side; thus, while output-first allocation reduces the delay for generating the input-side grant signals, it increases the delay $\tau_{grant,out}$ for computing the grant signals for individual output VCs.

A wavefront-based VC allocator, as illustrated in Figure 4.3, consists of a canonical $P\times V$-input wavefront allocator, with additional logic for generating the $P\times V$-wide request vector for each input VC as in the separable output-first case, and for reducing the $P\times V$-wide grant vectors to V-wide vectors as in the input-first case. Availability masking can be performed at the inputs to the wavefront allocator, while output-side grants are generated by combining the grant signals for all $P\times V$ input VCs for each individual output VC. As in the output-first case, the latter timing arc represents the allocator's overall critical path; its delay $\tau_{grant,out}$ is dominated by the actual

wavefront allocator.

4.3 Sparse VC Allocation

VCs are employed for a variety of different purposes in modern interconnection networks:

- Dependencies between different types of packets, such as request and reply packets in memory traffic, can lead to *protocol deadlock* at the network boundary if the network interfaces impose additional dependencies, e.g. as a result of finite buffer resources [20]. Such deadlock can be avoided by partitioning the total set of VCs into subsets corresponding to different *message classes* and mapping the different types of packets to these subsets appropriately. In addition, message classes can be employed to implement Quality-of-Service (QoS) policies and to provide traffic isolation between concurrently executing applications or virtual machines.

- In order to prevent deadlock scenarios caused by cyclic resource dependencies *within* the network, each message class can be partitioned into multiple *resource classes*, with transitions between the latter being restricted such as to enforce a partial order of resource acquisition. Examples of this approach include dateline routing in torus networks and two-phase routing as implemented in Valiant's algorithm [94], O1TURN [82] or UGAL [87].

- Finally, it is often beneficial to assign multiple VCs to each class; this increases network performance under load by reducing head-of-line blocking, and it improves channel utilization by increasing the number of logical connections that are multiplexed onto each physical link.

Thus, the total number of VCs V is determined by the number of message classes M, the number of resource classes R and the number of VCs assigned to each class C:

$$V = M \times R \times C \tag{4.1}$$

Prior work on router design has typically treated VCs in a uniform manner for the purpose of allocation: The allocator logic is designed such that it can handle requests from any given input VC to the whole range of output VCs, and a bit vector generated by the routing logic is used to constrain that range to a subset of allowable VCs on a packet-by-packet basis. However, based on the following observations, we can exploit the assignment of VCs to different message and resource classes to statically constrain the set of candidate output VCs that a given input VC can generate requests for, and thus greatly reduce the VC allocator's logic complexity.

As packets are assigned to message classes based on immutable properties—e.g., the packet's type or the application or virtual machine it belongs to—, a given packet can never transition from a VC belonging to one message class to one belonging to another. Therefore, we can partition the VC allocator into a set of smaller, completely independent allocators, each of which is dedicated to handling VCs for a specific message class. Since allocator complexity scales super-linearly with the number of ports, this can lead to substantial reductions in area and power consumption.

While a packet's resource class can change as it traverses the network, by construction, it can only do so subject to specific restrictions imposed by the routing function to prevent cyclic dependencies. Additionally, the routing function will generally limit the set of candidate output ports for the same purpose. This allows us to further constrain the set of possible transitions from input VCs to output VC.

As an example, Figure 4.4 illustrates the set of legal resource class and port transitions for Universal Globally Adaptive Load-Balanced (UGAL) routing on a 64-node two-dimensional Flattened Butterfly (FBfly). Upon injection into the network, each packet is either routed minimally to its destination, or it is first routed to a random intermediate router using a separate resource class (*non-minimal*) before being routed minimally to its final destination. Additional restrictions are imposed by the fact that each phase uses Dimension-Order Routing (DOR) and that minimal routing only allows for a single hop per dimension in FBfly networks. In aggregate, these restrictions reduce the number of possible transitions by 60 %.

In combination with the restrictions imposed by message classes, this enables us to reduce the size of each input- and output-side arbiter in a separable VC allocator

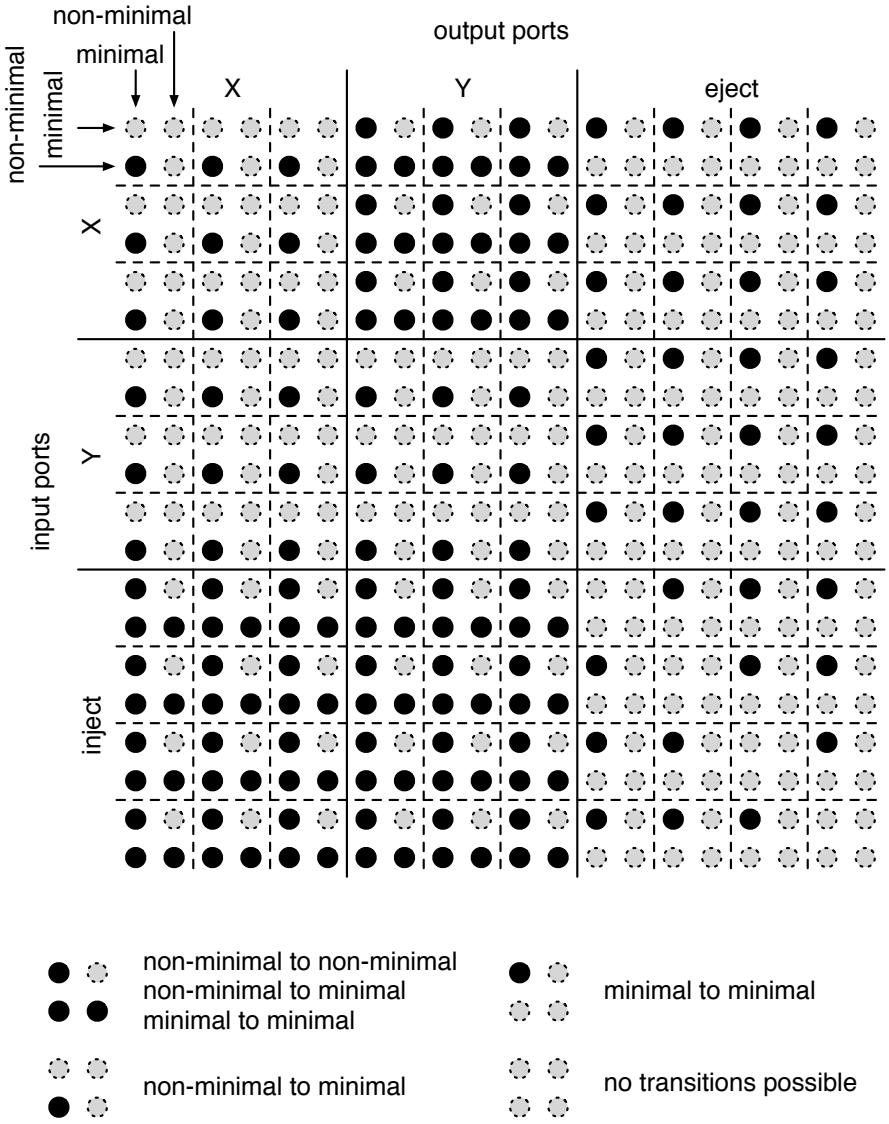


Figure 4.4: Legal class transitions for UGAL routing on FBfly networks.

to the total number of possible successors and predecessors for the resource class it belongs to, respectively, times the number of VCs in each class[2]. Additional optimizations in the input stage are possible If the routing function further limits the number of candidate resource classes for each individual packet.

Finally, all VCs belonging to the same class are equivalent from a functional point of view. As such, a given input VC can either use all of the output VCs that belong to a given class or none of them. VC allocator requests therefore need not specify individual candidate VCs within the class, but instead can select the class as a whole. This facilitates additional logic optimizations.

4.4 Evaluation

In the present section, we investigate the impact of the allocator implementations described earlier in this chapter on router and network performance, quantify their delay, area and energy efficiency, and evaluate the efficacy of the proposed sparse VC allocation scheme.

4.4.1 Experimental Setup

In order to perform detailed cost evaluations and to develop insights about basic trends and trade-offs that are independent of network-level implementation details, we first investigate individual allocator instances. To this end, we have developed parameterized RTL implementations for the three allocator architectures discussed in this chapter.

We first evaluate matching quality for each implementation using open-loop simulations: For each VC configuration, we apply a set of 10000 pseudo-randomly generated request matrices that are consistent with that configuration, and we count the total number of grants produced by each allocator. As a point of comparison, we also compute the maximum possible number of grants for the sequence of request matrices using maximum-size allocation as described in Section 3.4.

[2] Except for special cases in which the set of resource classes is divided into multiple disconnected subsets, wavefront-based VC allocators only benefit from the restrictions imposed by message classes.

Subsequently, we evaluate the impact of VC allocation on overall network performance using the cycle-accurate BookSim 2.0 network simulator. We model input-queued router designs with VC flow control [17]. Input buffers are statically partitioned, with eight entries assigned to each VC. Lookahead routing [28] is used to eliminate the need to perform route computation in a separate pipeline stage; consequently, we model a pipeline that comprises three stages: VC allocation, switch allocation and switch traversal.

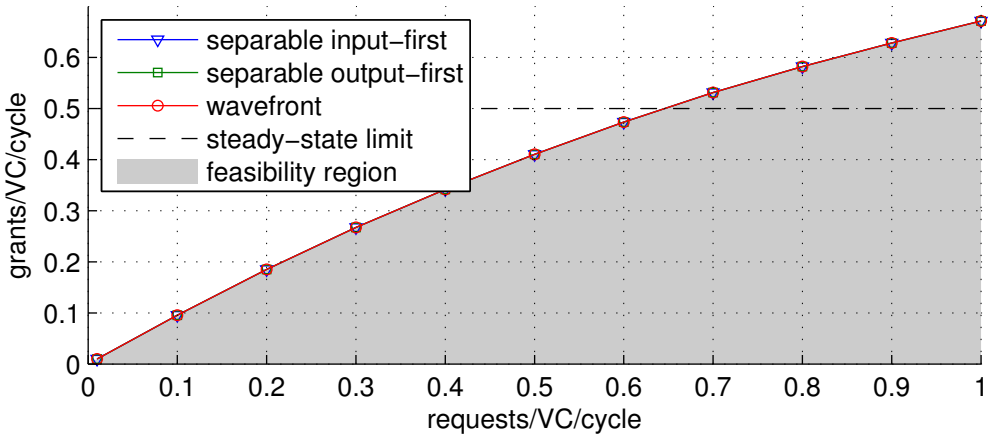
We exercise the network by injecting synthetic traffic; packet inter-arrival times follow a Bernoulli distribution with configurable rate, and destinations are selected randomly or according to a set of permutation patterns [20]. To model memory traffic, we generate packets with a bimodal length distribution: Short packets—representing read requests and write replies—comprise two flits, whereas long packets—representing read replies and write requests—include four additional flits that carry payload data. For each traffic pattern, we sweep injection rates and measure the resulting packet latency for each rate. Measurements are taken after an initial warm-up period that allows the network to reach steady state, and all results include source queuing delay.

Finally, to investigate delay and cost trade-offs and to quantify the benefits afforded by sparse VC allocation, we synthesize each allocator in a commercial standard-cell design flow. For additional details on our synthesis setup, we refer to Section 2.7.1.

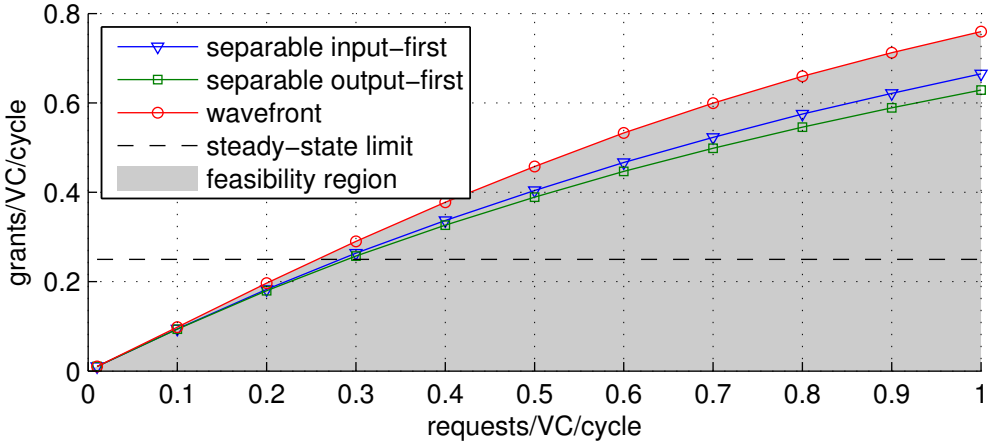
4.4.2 Matching Quality

Figure 4.5 compares matching quality for the three allocator implementations considered in this chapter and illustrates how it changes as the number of VCs per class increases. The shaded area in each graph marks the feasibility region; its upper boundary corresponds to the maximum number of grants that can be generated for the given request patterns.

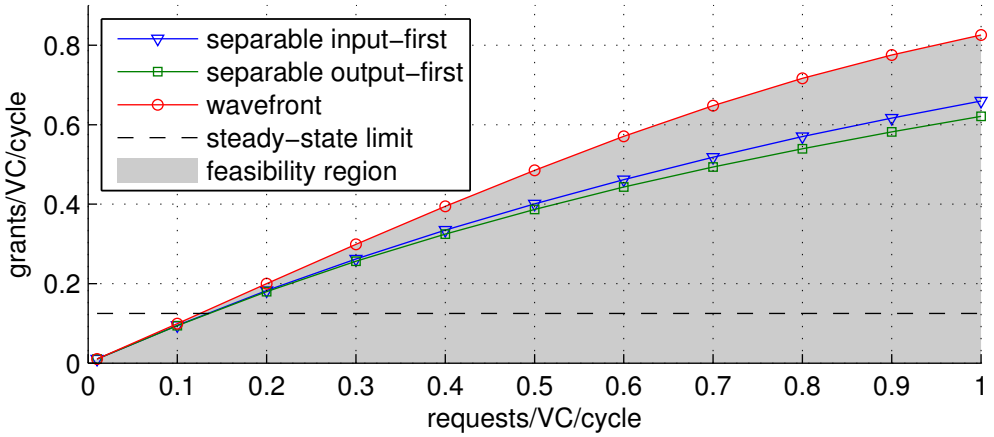
We report results for port (P) and VC (V) configurations representative of a Mesh network with request-reply traffic; additional simulation runs show that configurations with more ports or packet classes exhibit similar characteristics as long as the same number of VCs is assigned to each packet class. We omit detailed results for brevity.



(a) $P = 5$, $V = 2{\times}1{\times}1$.



(b) $P = 5$, $V = 2{\times}1{\times}2$.



(c) $P = 5$, $V = 2{\times}1{\times}4$.

Figure 4.5: VC allocator matching quality.

For cases where a single VC is assigned to each packet class, as shown in Figures 4.5a, all three allocator implementations generate maximum matchings for every valid request matrix: As each input VC can use only one specific output VC, each allocator will produce grants for all non-conflicting requests, as well as a single grant for each group of conflicting requests. This represents the best possible—i.e., maximum—matching for such configurations.

Results start to diverge as additional VCs are assigned to each packet class, as illustrated in Figures 4.5b and 4.5c. As before, all three allocator types produce grants for non-conflicting requests. In the presence of conflicts—i.e., multiple input VCs requesting output VCs from the same packet class at a given output port—the wavefront allocator will grant as many of the requests as there are available VCs in that class, and therefore continues to produce maximum matchings[3]. For the separable allocators, on the other hand, scenarios as described in Section 3.2 can arise where some of the VCs within a given class are left unassigned even in the presence of unsatisfied requests; e.g., for a separable input-first implementation, multiple input VCs destined for the same packet class might select the same output VC during input-side arbitration, leaving other available VCs in that class unused. Consequently, matching quality for the separable implementations decreases both for higher injection rates and for larger numbers of VCs per class, as both increase the probability that such lockouts will occur.

Input-first allocation provides slightly better matching than output-first allocation, as it performs the narrower $V : 1$ arbitration at the input side before the wider $P \times V : 1$ arbitration at the output side; because an n-input arbiter grants only one of up to n requests, eliminating up to $n - 1$ others, this allows more requests to be propagated from the first arbitration stage to the second one than in the output-first case.

Under heavy load, the wavefront allocator generates 20 % and 25 % more grants

[3] Note that while a wavefront allocator is not guaranteed to produce maximum matchings for *arbitrary* request matrices, legal request matrices *for VC allocation* exhibit additional properties; in particular, the set of output VCs requested by any two input VCs either overlaps completely (if they request the same class) or not at all (if they request different classes). With this additional constraint, any *maximal* matching is also a *maximum* matching.

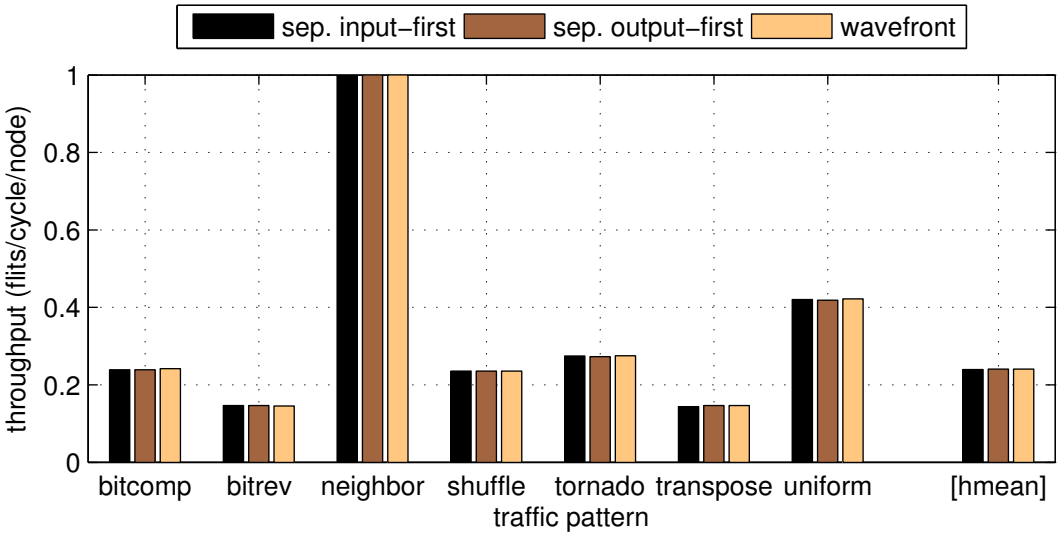


Figure 4.6: Impact of VC allocation on saturation throughput (8×8 Mesh, 8 VCs).

than the separable input- and output-first allocators, respectively. However, in practice, a given load level can only be sustained in steady state if the average number of generated grants does not exceed the number of new requests. The latter quantity is inherently limited by the fact that at most one new packet can arrive at each input port—shared among multiple VCs—in any given cycle; the dashed lines in Figure 4.5 denote this threshold, expressed as a fraction of the per-VC request rate, for each configuration. VC allocation can only operate in the regions above these curves for limited periods of time, e.g. as a result of bursty traffic. While a wavefront allocator's higher matching quality facilitates faster recovery from such transient network states, its overall utility in the context of VC allocation is significantly reduced by this effect.

4.4.3 Network-Level Performance

As outlined in Section 4.4.1, a request-reply packet pair in our traffic model always comprises a total of six flits; consequently, as VC allocation only needs to be performed once per packet, on average only one in three flits generates a requests, resulting in low effective load on the VC allocator. We saw in Figure 4.5 that all three allocator types

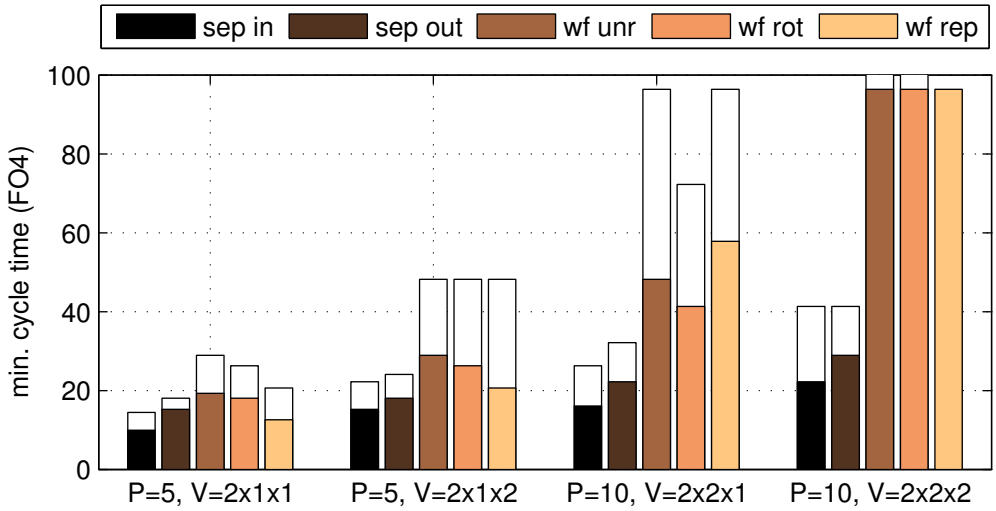


Figure 4.7: Minimum cycle time for VC allocator implementations.

produce near-maximum matchings when the request arrival rate is low; therefore, we expect the choice of VC allocator to not significantly affect performance at the network level.

Indeed, simulation results for an 8×8 Mesh network—shown in Figure 4.6—confirm that VC allocation does not affect zero-load latency and has minimal impact on the saturation throughput measured for each traffic pattern. This holds even as the number of VCs or the switch allocation scheme—Figure 4.6 uses 8 VCs and a separable input-first switch allocator—is varied: While the absolute throughput values measured for individual traffic patterns change, no significant discrepancies between the results for the different VC allocator implementations are introduced.

4.4.4 Delay and Cost

Figure 4.7 compares the minimum cycle time at which the individual allocators can run for configurations with different numbers of ports (P) and VCs (V). For the wavefront-based VC allocator, we consider implementation variants using unrolling,

input transformation and replication as described in Section 3.3.2. The white segments at the top of each bar indicate the minimum cycle time for naïve implementations, while the colored sections show the cycle time achieved after applying the optimizations described in Section 4.3[4].

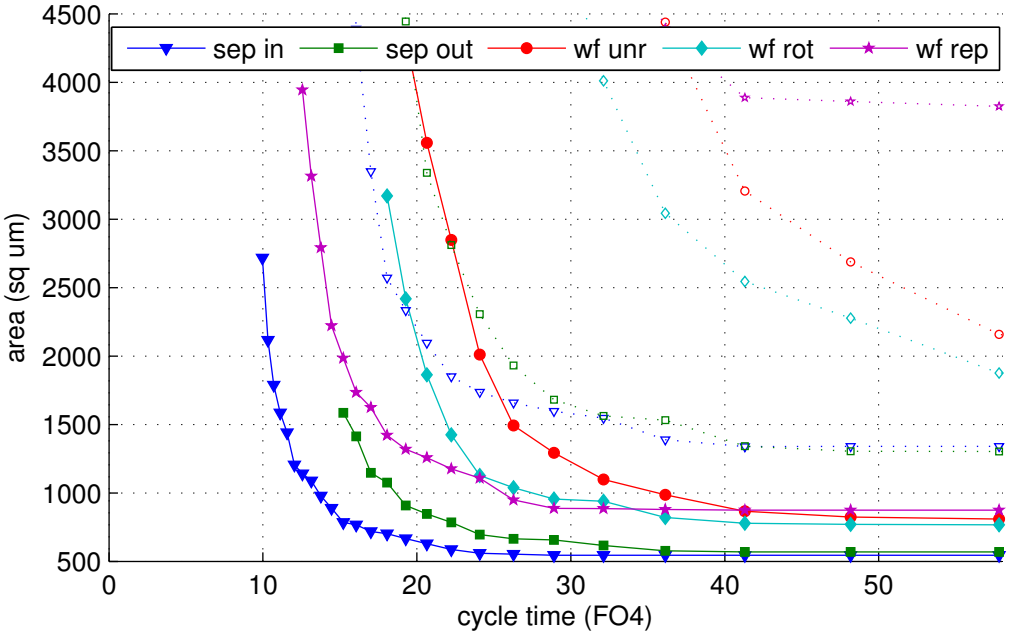
Sparse VC allocation yields significant speedup in all cases; in particular, it improves the minimum cycle time of the fastest implementation for each design point—generally the separable input-first implementation—by of 31–46 %. Between the wavefront-based implementations, the one using replication can operate at significantly higher clock frequencies than the ones employing unrolling and I/O transformation for the five-port configurations; with ten ports, the maximum operating frequencies for all three variants are significantly lower than those for the two separable allocators.

Figure 4.8 illustrates the area-delay trade-off with (solid markers) and without (dotted lines) sparse VC allocation. We show results for the five-port configuration with two message classes and a single resource class; results for configurations with more ports and packet classes largely follow the expected scaling behavior of the underlying allocator type, and thus the benefits afforded by sparse VC allocation become more pronounced.

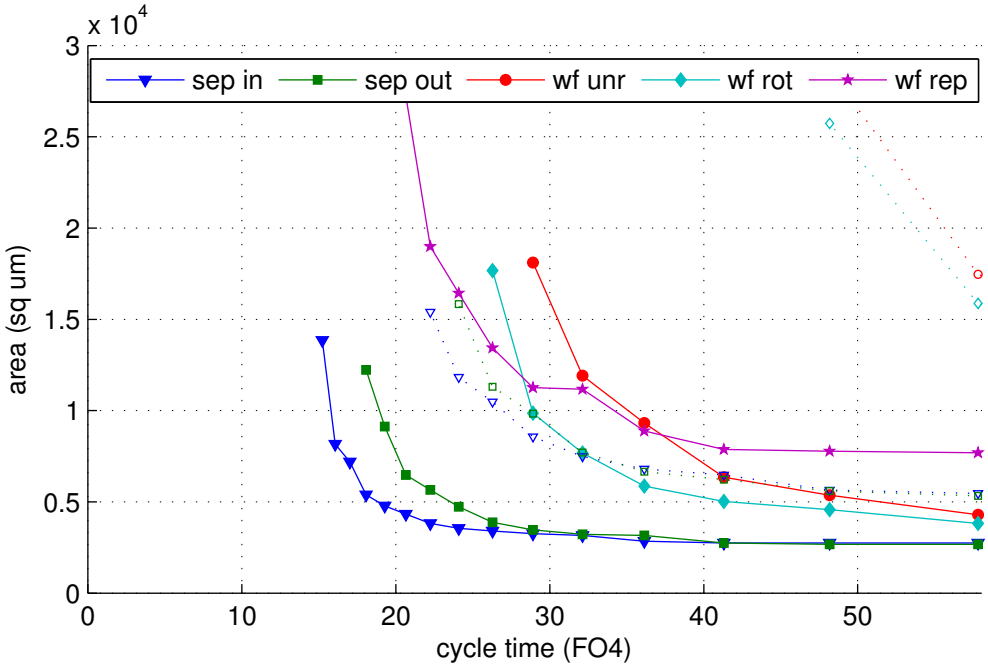
With a single VC per packet class, as shown in Figure 4.8a, cell area at low target frequencies is reduced by 55–50 % for the separable implementations, as well as the unrolling- and rotation-based wavefront implementations; with a second VC per class—as shown in Figure 4.8b—the improvement is slightly smaller. For the replication-based wavefront design, sparse VC allocation reduces area by up to 75 % at low target frequencies. Nevertheless, the three wavefront-based implementations remain significantly more expensive than either of the separable implementations and as such do not represent attractive choices for VC allocation.

By reducing allocator complexity, sparse VC allocation also significantly improves

[4] For the largest configuration shown, the minimum cycle times for the naïve implementations of the unrolled and the rotation-based wavefront allocators exceed 250 FO4; we truncate the corresponding bars at 100 FO4 to ensure that other results remain readable. We were unable to successfully synthesize the naïve implementation of the replication-based wavefront allocator due to its excessive size.



(a) $P = 5$, $V = 2{\times}1{\times}1$.



(b) $P = 5$, $V = 2{\times}1{\times}2$.

Figure 4.8: Area-delay trade-off for VC allocator implementations.

energy efficiency. Detailed results—omitted here in the interest of brevity—show that area and Power-Delay Product (PDP) exhibit essentially the same scaling behavior and consequently follow the same trends.

For all design points considered, results show that the separable input-first implementation achieves better delay, area and PDP than the other implementations under investigation. As the choice of VC allocator does not significantly affect network performance, it thus represents the preferable overall design choice for this use case.

4.5 Related Work

VC allocation in interconnection networks has been addressed in a number of prior research contributions:

Peh and Dally [76, 77] present an analytical delay model for separable VC allocators. Their model is derived from gate-level schematics using the *logical effort* method [88]; as such, it is primarily geared towards full-custom implementations that optimize for minimum critical path delay regardless of the implications for area and power. Furthermore, the models do not account for wire delay, which has become a critical factor in modern sub-micron semiconductor processes.

Mullins et al. [62] propose a technique for reducing the delay of separable input-first VC allocators by precomputing arbitration decisions and by using a free VC queue at each output port, the front-most element of which is assigned to incoming requests.

Kumar et al. [47] describe a scheme that combines VC and switch allocation into a single step; we explore a similar design in Section 5.4.

Finally, Zhang and Choy [99] investigate approaches for reducing the complexity of separable VC allocators based on utilization statistics for their individual constituent arbiters and present detailed delay, area and power results.

4.6 Summary

In this chapter, we have explored the design space for VC allocators in the context of NoC routers. In particular, we have presented practical hardware implementations for three exemplary VC allocator architectures based on the elementary designs described in Chapter 3.

In comparing matching quality between the three allocator implementations, we find that differences primarily manifest at load levels that cannot be sustained continuously. Consequently, network-level steady-state performance in largely insensitive to the choice of VC allocator; we have confirmed this using extensive simulation runs on an exemplary 64-node Mesh network. Thus, when selecting a VC allocator implementation, the optimal choice is primarily determined by delay and cost considerations.

In order to improve the latter characteristics, we have furthermore developed sparse VC allocation, a scheme that reduces the complexity of the VC allocator by taking advantage of the fact that many common use cases organize VCs as multiple disjoint packet classes. By structuring the allocator to explicitly enforce restrictions on transitions between packet classes in hardware, sparse VC allocation reduces its minimum cycle time by up to 46 % compared to a naïve implementation, while improving area and energy efficiency by up to 75 %.

Overall, our results suggest that the separable input-first implementation provides the optimal delay, area and energy efficiency among the three designs considered in the present chapter.

Chapter 5

Switch Allocation

5.1 Overview

Once a packet has completed Virtual Channel (VC) allocation, its flits can be forwarded to the selected destination port subject to buffer space availability. For each flit to be transferred, a crossbar connection between the corresponding input and output ports must be established for one cycle. The *switch allocator* is responsible for scheduling such crossbar connections; in particular, it generates matchings between requests from active VCs at each of the router's P input ports on the one hand and crossbar connections to its P output ports on the other hand[1]. The quality of the generated matchings directly affects the router's latency and throughput under load.

With VC flow control, flits may only be sent downstream if sufficient buffer space is available at the receiving router. To this end, routers maintain a set of credit counters at each output port that track the number of available buffer entries for each downstream VC. A given input VC can only request access to the crossbar if its destination VC has at least one credit available.

We assume throughout the remainder of this chapter that each input VC maintains a proxy of the credit count for its assigned destination VC; this allows credit

[1] To minimize the cost of the crossbar, Network-on-Chip (NoC) routers typically do not implement speedup [20].

checks to be performed locally at each input port, eliminating substantial propagation and multiplexing delay. Maintaining such proxy registers is feasible because the destination port and VC are fully determined at the end of VC allocation, allowing the proxy to be set up in time for the first cycle of switch allocation[2].

The grant signals generated by the switch allocator are used to set up the registers that control crossbar connectivity. In addition, the switch allocator notifies the winning VC at each input port, causing the latter to prepare its frontmost flit for crossbar traversal—e.g., by initiating a read access to the input buffer—and to decrement its credit count proxy. Finally, the output-side credit counter for each winning flit's destination VC is updated to reflect the fact that a credit has been consumed.

5.2 Implementation

As in the case of VC allocation, we can implement switch allocators by adapting the canonical designs described in Chapter 3. Specifically, additional logic is required to combine the requests from individual VCs at each input port, as well as for notifying the winning input VCs and initiating output-side credit count updates.

In a separable input-first implementation, shown in Figure 5.1, a V-input arbiter first selects a winner among all active VCs at each input port. As each VC can only request a single output port, this effectively replaces the input-side P-input arbiter found in the canonical implementation shown in Figure 3.1a[3]. Each input port then proceeds to request the desired output port for its winning VC.

At the output side, a round of P-input arbitration is performed as in the canonical design. The grants generated by these arbiters are used to set up the crossbar control registers; furthermore, combined with the winning VCs from the selected input ports, they are used to identify the output-side credit counters that need to be updated[4].

[2] By the same argument, it is generally not feasible to use a similar approach for masking unavailable VCs at the beginning of VC allocation.

[3] We could instead select an output port by performing P-input arbitration over the combined requests from all VCs; however, unless the number of VCs significantly exceeds the number of ports, which is not commonly the case in NoCs, this generally leads to a less efficient implementation.

[4] Recall from Section 4.1 that the state maintained for each output VC includes registers that track the input port and VC to which it is currently assigned.

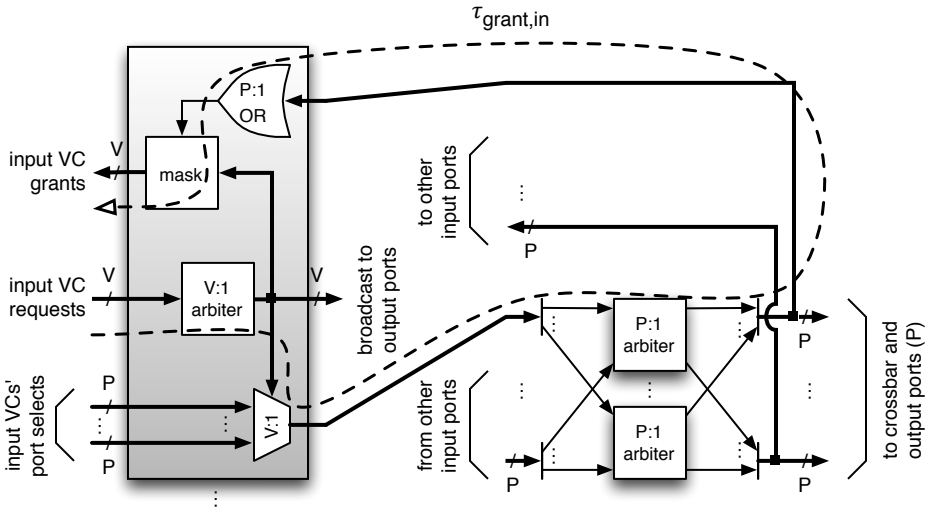


Figure 5.1: Separable input-first switch allocator.

Finally, the winning input ports are notified, prompting each to generate a grant for the VC that was selected by the input arbiter at the beginning of allocation. This timing arc, with an associated delay of $\tau_{grant,in}$, represents the allocator's critical path.

Figure 5.2 illustrates the separable output-first implementation. Here, requests from all active input VCs are combined and forwarded to the output side, where P-input arbitration is performed among all incoming requests and the resulting grants are propagated back as in the input-first case. Since a given input port can receive grants from multiple outputs, it is necessary to select one among them; to this end, the allocator performs arbitration among all those input VCs whose requests match one of the granted output ports.

Unlike in the input-first case, the grant signals generated by the output arbiters cannot drive the crossbar control registers directly, as this could cause an input to be connected to multiple outputs. Instead, each input port sets up its crossbar connection using the winning VC's port select signal at the end of allocation. The final port select signals, combined with the winning VC from each input port, furthermore trigger output-side credit count updates as in the input-first case; the associated timing arc with delay $\tau_{grant,out}$ represents the output-first allocator's critical path.

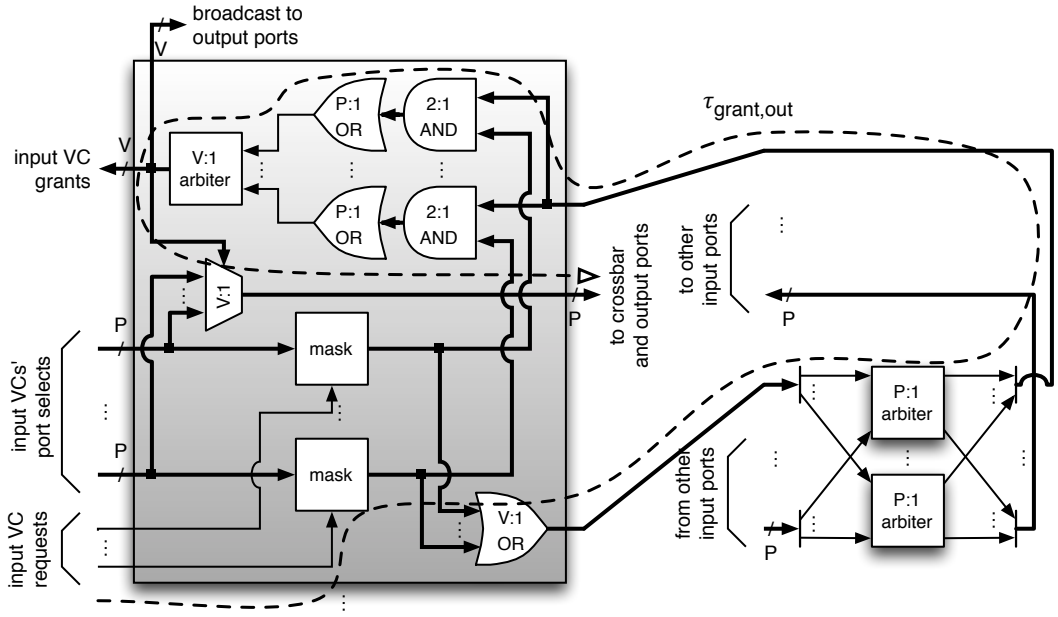


Figure 5.2: Separable output-first switch allocator.

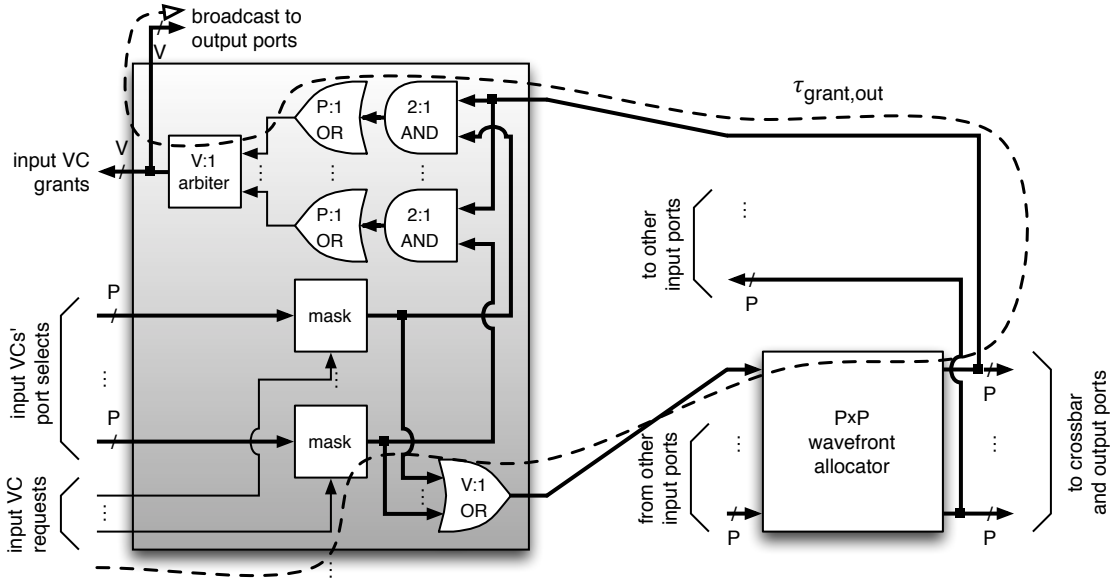


Figure 5.3: Wavefront-based switch allocator.

The implementation for a wavefront-based switch allocator, shown in Figure 5.3, largely follows that of the separable output-first variant: At the input side, requests from all active VCs are combined and forwarded to the central wavefront allocator. However, as the latter guarantees that at most one output port is granted to any given input port and vice versa, its grant signals can be used to set up the crossbar control registers directly.

Despite the fact that each input port can only receive a single grant, a final input-side arbitration step must be performed in the same way as in the separable output-first implementation. This is necessary to account for situations where multiple input VCs request the same output port.

In order to correctly update the output-side credit counts, the winning VCs have to be communicated to the output ports. As in the output-first implementation, the timing arcs involved in performing the credit count update represent the overall critical path of the design; however, because the port selections are fully determined by the wavefront allocator and thus available earlier than in the output-first case, it is the timing of the input VC selection signals that determines the delay $\tau_{grant,out}$ of the wavefront allocator's critical path.

5.3 Speculative Switch Allocation

In addition to using lookahead routing, we can further reduce the router's pipeline delay by allowing a packet's head flit to bid for crossbar access *in parallel* to participating in VC allocation, *speculating* that a VC will be assigned [76, 77]. If a grant is received from the VC allocator as predicted, the outcome of switch allocation determines what action to take next as in the non-speculative case. However, in case of mis-speculation, the flit must ignore the outcome of switch allocation—potentially leaving an assigned crossbar time slot unused—and issue requests to both allocators again in the next cycle.

Because speculative requests are issued before an output VC has been assigned, it is not feasible to perform input-side credit masking using proxy registers as described in Section 5.1. Instead, we must check for buffer space availability late in the cycle,

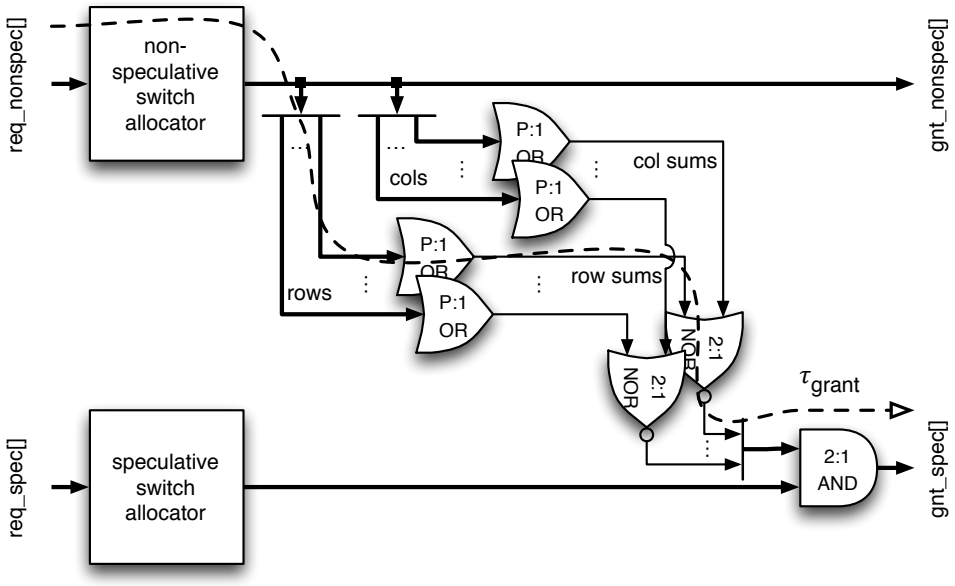


Figure 5.4: Canonical implementation of speculative switch allocation.

after the VC allocator has computed its grants. This effectively introduces an additional degree of uncertainty: Even after receiving grants from both allocators, the head flit must leave its assigned crossbar time slot unused if the allocated output VC has no credits available.

As all speculative requests thus carry the risk of not being able to use their assigned crossbar time slots, we must prioritize non-speculative requests during switch allocation in order to avoid potential performance degradation. In the following, we investigate approaches for extending the previously described switch allocator designs accordingly.

5.3.1 Canonical Speculation

The canonical implementation of speculative switch allocation—as described in [76, 77]—uses two separate allocators for handling non-speculative and speculative requests. Allocation is performed independently for each set of requests, and a final merging stage resolves conflicts between the grants generated by both allocators. A

block diagram of the canonical implementation is shown in Figure 5.4.

The checks for input and output conflicts require two sets of reduction ORs to generate row- and column-wise summary bits that indicate the presence of a non-speculative grant. Pairs of summary bits are then combined to form a $P \times P$ matrix that serves as a mask for speculative grants. For a given type of allocator, this increases the critical path delay τ_{grant} by the sum of the delays incurred by the reduction trees and the masking logic when compared to a purely non-speculative implementation[5].

As suggested in [76], for separable input-first allocators, we can exploit Equation 2.4 to speed up the computation of the column-wise summary bits by performing it in parallel with the output arbitration stage; however, as the row-wise summary bits cannot be determined before output arbitration has completed, this does not shorten the overall critical path. A similar argument can be made for separable output-first allocators.

5.3.2 Pessimistic Speculation

The benefits of speculative switch allocation are most pronounced when network load is low and end-to-end packet latency is dominated by pipeline delay. In such cases, the expected number of pending requests at each router—and hence, the likelihood of any given request not being granted due to a conflict—is small. Consequently, we can pessimistically mask those speculative grants that conflict with one or more non-speculative *requests*—rather than *grants*—while maintaining the desirable performance characteristics of the canonical implementation at low to medium load levels.

As shown in Figure 5.5, this allows us to compute the row- and column-wise reduction trees in parallel with allocation, removing them from the critical path. Pessimistic masking thus reduces the delay penalty associated with speculative switch allocation to the delay of the final stage of 2-input AND gates.

As network load—and thus the level of congestion in the routers—increases, more

[5] Note that we could exploit the additional slack on the input side of the speculative path by using a more complex allocator for these requests; however, in practice, this yields little benefit, as the speculative request matrix is typically less densely populated than the non-speculative one.

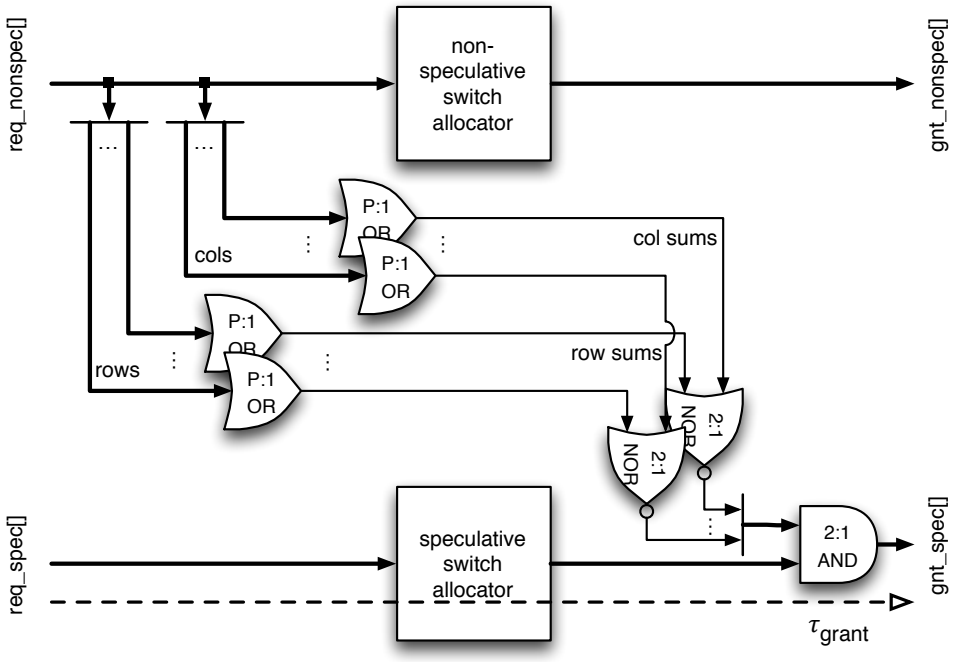


Figure 5.5: Pessimistic speculation shortens the critical path.

and more speculative requests are unnecessarily masked by non-speculative requests that fail to produce a grant. Consequently, the efficacy of pessimistic speculation is reduced as the network approaches saturation, and its performance approaches that of a purely non-speculative implementation. However, in many practical applications, NoCs operate at modest load levels, and performance is primarily limited by end-to-end latency rather than available network bandwidth [81]. In such scenarios, pessimistic speculation represents an attractive design choice.

5.3.3 Priority-Based Speculation

Instead of prioritizing non-speculative requests by handling speculative ones in a separate low-priority allocator and resolving conflicts externally, we can often achieve the same goal with less overhead using a single *priority-aware* allocator.

For example, we can implement priority-aware separable allocators by replacing each arbiter in the canonical designs shown in Figure 5.1 and Figure 5.2 with a multi-priority arbiter as described in Section 2.6. Compared to implementing speculative switch allocation using two separate allocators, effectively doubling the implementation cost, this introduces significantly less overhead. In particular, priority-based speculation allows the arbiter state registers, which account for a substantial fraction of the overall cost of the allocator, to be shared among both priority levels.

As priorities are enforced at each individual arbiter, priority-based speculation exhibits slightly different performance characteristics than both the canonical and the pessimistic implementations. E.g., in an input-first design, the input arbitration stage can propagate at most a single request from each input port to the output stage; in contrast, each input in the canonical design can propagate both a non-speculative and a speculative request, improving its odds of receiving a grant. However, at the same time, the priority-based implementation avoids cases where a given input receives grants for different outputs from the non-speculative and the speculative allocator, leaving the speculative crossbar assignment unused.

In principle, we can apply the same approach to wavefront-based switch allocators; however, in practice, implementing a multi-priority wavefront allocator essentially

involves performing two successive rounds of wavefront allocation[6]. The substantial associated delay increase typically makes such designs unattractive for NoC routers.

5.4 Combined VC and Switch Allocation

A canonical VC allocator as described in Chapter 4 is capable of assigning a new output VC to every input VCs in any given cycle. However, when using speculative switch allocation as described in the previous section, at most one of the input VCs can also receive a grant from the switch allocator and thus begin crossbar traversal; any additional output VC assignments made have no immediate effect on throughput[7]. While over-provisioning VC allocation bandwidth in this manner minimizes the likelihood of mis-speculation, this is achieved at the cost of the VC allocator's substantial logic complexity.

Based on the insight that only a single VC assignment per input port is required in order to be able to sustain full throughput, we can achieve the performance benefits afforded by speculative switch allocation in a more efficient way by assigning both VCs and crossbar time slots using a *combined* VC and switch allocator. Eliminating the dedicated VC allocator shortens the router's critical path and significantly reduces its cost and complexity.

We can implement combined allocation by building on the basic structure of a speculative switch allocator and introducing additional logic for assigning VCs based on the generated grants. Allocation for body and tail flits is performed in the same way as when using separate VC and switch allocators. Head flits similarly issue requests to the combined allocator; each winning head flit is implicitly granted the next available output VC for its selected packet class at the assigned destination port. If no suitable VC with at least one credit is available, allocation must be attempted again in the next cycle; the same is true for any head flits that fail to receive a grant. As such, only those head flits that can immediately begin crossbar traversal

[6] In particular, unlike for arbiters, it is not possible to compute the grants for the different priority levels in parallel, as the set of grants for any given level depends on each individual higher-level grant.

[7] They can, however, *indirectly* improve throughput in the presence of congestion, as subsequent requests from the assignees will no longer be speculative.

are assigned an output VC; conversely, only head flits that have not yet been assigned an output VC participate in allocation[8].

As any grants issued to head flits may subsequently need to be discarded if all suitable output VCs are found to be in use or to not have credits available[9], it is necessary to prioritize requests from body and tail flits in order to avoid potential performance degradation; this mirrors the handling of non-speculative requests in speculative switch allocation. As a side benefit, this prioritization scheme also reduces unnecessary packet fragmentation: Once a packet's head flit has been granted an output port, another packet will only be granted the same output while the first packet is still active if it either temporarily exhausts its supply of body and tail flits—e.g., as a result of earlier fragmentation—or experiences a credit stall. In both cases, interleaving packets is necessary to avoid throughput degradation.

While prior research has proposed the use of a FIFO queue per packet class at each output port to track the next available output VCs for each class [47,62], we find that we can implement the same functionality in a more efficient way by replacing each queue with a simple round-robin arbiter[10]: The input to each arbiter is a bit mask that indicates which—if any—of the output VCs assigned to a particular class are currently not assigned to an input VC and have at least one credit available; its output corresponds to the next VC for each class to assign to a requesting head flit. Because these arbiters operate in parallel with switch allocation, they do not increase the router's critical path delay.

In a baseline router that performs VC and switch allocation separately, a fair VC allocator can partially mitigate fairness issues inherent in the design of the switch

[8] This differs from speculative switch allocation, where a head flit that is granted an output VC but no crossbar time slot becomes non-speculative.

[9] In the general case, it is not feasible to ascertain availability of a suitable output VC at the beginning of allocation due to timing constraints. While one might imagine using input-side proxy registers that track for each output port and packet class whether at least one VC is available, such an approach would incur significant overhead as every input port would have to track all possible packet classes for all output ports concurrently.

[10] In principle, we could further reduce complexity by using fixed-priority arbiters as described in Section 2.2 to perform VC selection; however, this concentrates load on the minimum required number of VCs and thus leads to increased packet latency. In contrast, round-robin arbiters lead to more evenly balanced load across all available VCs.

allocator; this is because any newly arriving head flits either have to undergo VC allocation before competing for crossbar access (in a purely non-speculative implementation) or have lower priority during switch allocation than any flits that have already been assigned an output VC (in a speculative implementation). With combined allocation, on the other hand, head flits that fail to secure access to the crossbar are not assigned an output VC and must thus re-attempt switch allocation with the same priority as newly arriving head flits. This can exacerbate the starvation effects that cause unstable network behavior when the load is increased beyond the saturation point [20]. As a result of the fairness issues described in Section 3.3.1, this effectively renders wavefront-based switch allocators unattractive for use in combined allocation.

5.5 Evaluation

In this section, we compare the performance of the allocator designs described in Section 5.2, and we evaluate the different implementation alternatives for speculative switch allocation outlined in Section 5.3, as well combined VC and switch allocation as described in the preceding section.

5.5.1 Experimental Setup

We employ the same methodology for evaluating switch allocator implementations that we used for VC allocators in Section 4.4; i.e., we first compare matching quality for individual allocator instances, then evaluate network-level performance, and finally investigate implementation cost and delay.

In order to be able to present meaningful results for combined VC and switch allocation, we furthermore synthesize complete router instances. Based on the results from Chapter 4, we use separable input-first VC allocators and employ sparse VC allocation for all configurations that do not employ combined VC and switch allocation. Furthermore, we allow synthesis to optimize switch allocator and crossbar by taking advantage of turn restrictions imposed by the routing function where applicable.

5.5.2 Matching Quality

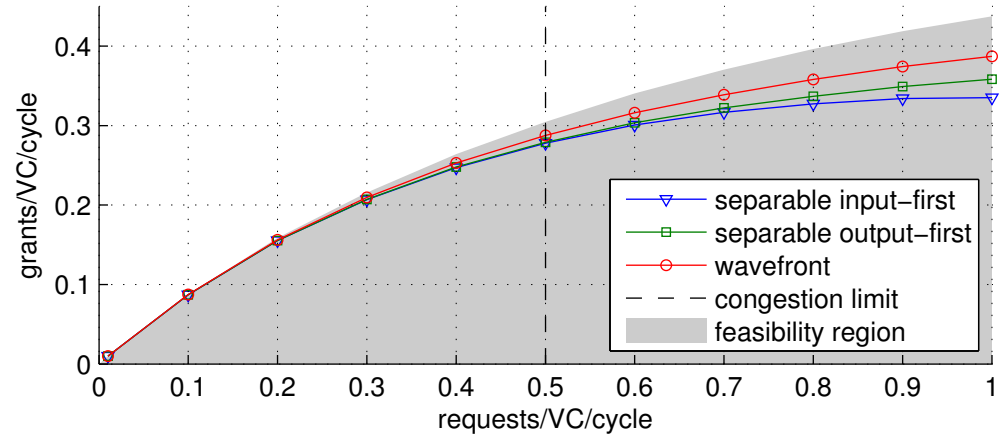
Figure 5.6 illustrates the matching quality of separable input-first, separable output-first and wavefront-based switch allocators as a function of the average rate at which each individual VC generates requests. We consider the corresponding configurations to those shown in Figure 4.5. Unlike VC allocators, switch allocators are not affected by the division of VCs into packet classes.

The shaded area in each graph represents the range of possible matching cardinalities, with an upper limit given by the performance of a maximum-size allocator. Because all V VCs at any given input port share a single crossbar input, the average grant rate across all VCs is inherently limited to V^{-1} grants per VC per cycle.

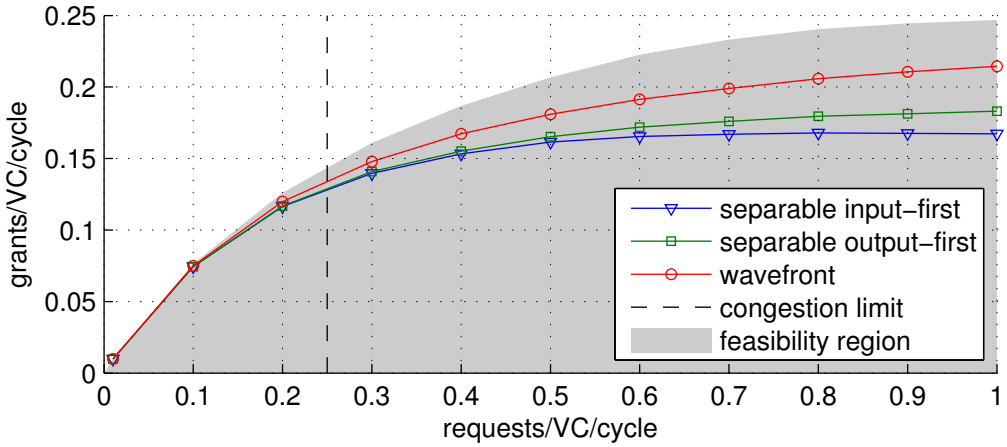
Dashed lines indicate the maximum request rate per VC that can be sustained in the absence of congestion, which corresponds to a new flit arriving at each input port in every cycle; this represents a measure for the maximum *instantaneous* load on the switch allocator. However, it does not represent a limit for steady-state operation: As a result of congestion, multiple VCs at an input port may have buffered flits and thus issue requests to the switch allocator at the same time. Because only one of these requests can be granted in any given cycle, at most one VC at a time can become inactive; at the same time, a new flit may arrive at the input port and either cause an additional VC to become active or prevent a VC that received a grant from becoming inactive. Thus, regardless of how many VCs were active in one cycle, it is possible for the total number of active VCs—and hence, requests—to remain the same in the next cycle regardless of whether a grant was received. Any given load on the switch allocator can therefore be sustained indefinitely once congestion has built up.

As in the case of VC allocation, all three allocator variants yield near-maximum matchings at low request rates where conflicts are unlikely, and differences in matching quality become more pronounced as the number of VCs and the average request rate per VC increase. The number of VCs limits the maximum number of outputs that any given input port can request, and the combination of both quantities thus determines the average density of the the generated request matrices.

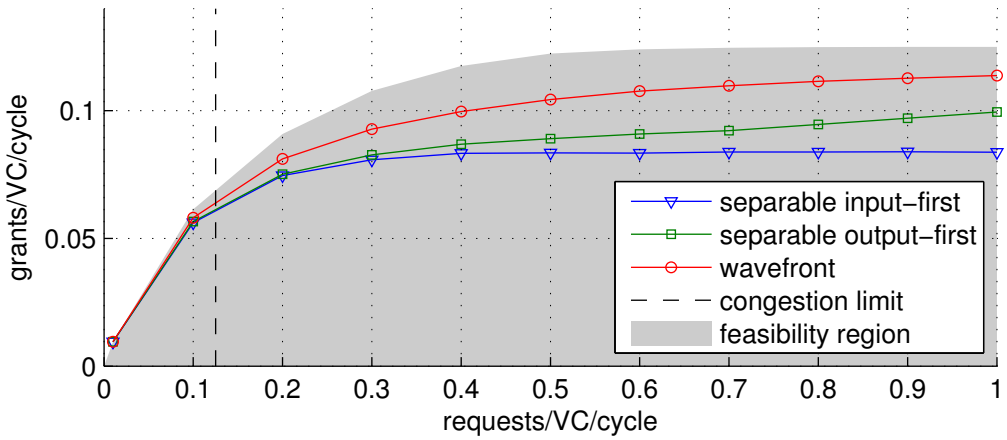
For a fully populated request matrix, any maximal matching is also a maximum



(a) $P = 5$, $V = 2$.



(b) $P = 5$, $V = 4$.



(c) $P = 5$, $V = 8$.

Figure 5.6: Switch allocator matching quality.

matching; however, as the number of zero elements in the matrix increases, the likelihood of any given maximal matching also being a maximum matching decreases. Statistically, if each VC issues a request in every cycle, the expected number of VCs necessary to ensure that the combination of their requests includes each of the P output ports—and thus, that the request matrix is fully populated—is given by the following expression[11]:

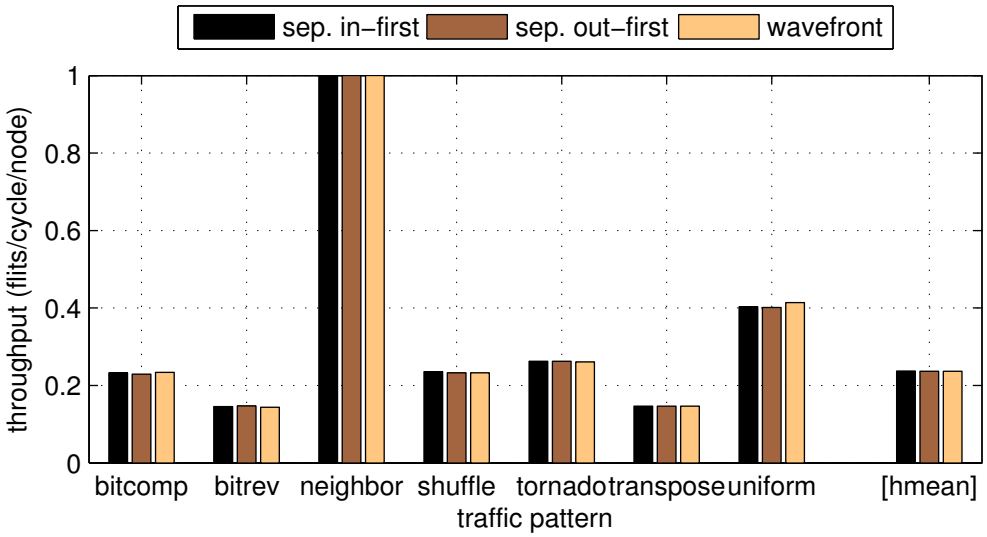
$$V_{min} = P \times \sum_{i=1}^{P} \frac{1}{i} \tag{5.1}$$

Evaluating for $P = 5$, we find that the expected number of VCs necessary to ensure a fully populated request matrix is $V_{min} = 11.42$. Thus, the wavefront allocator's matching quality remains below that of a maximum-size allocator for all three configurations we consider.

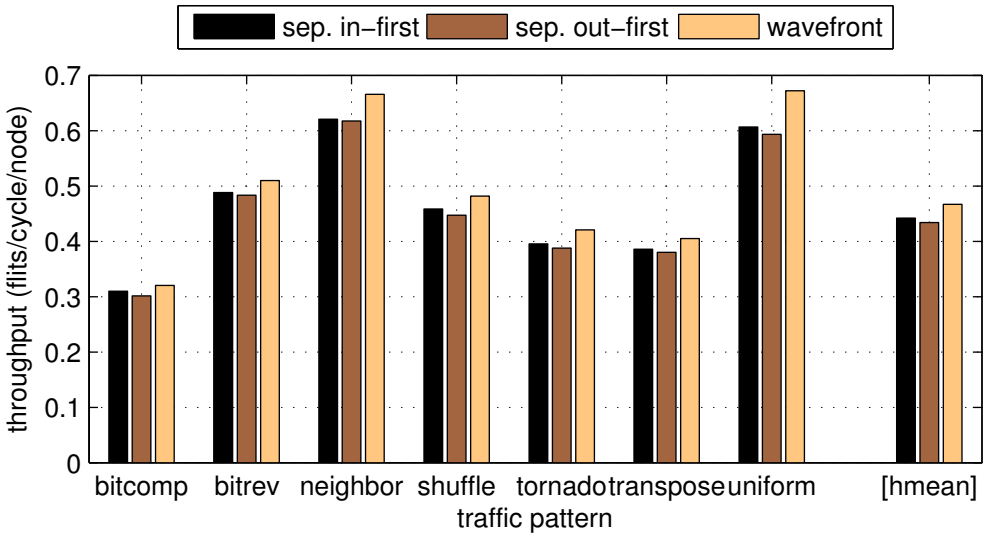
The separable input-first switch allocator eventually becomes limited by the fact that it can only propagate a single request per input port to the output arbitration stage; as such, once the probability $p_{sat} = 1 - (1 - p_{req})^V$ that at least one VC per input port has a flit available in any given cycle approaches one, further increases in the request rate p_{req} do not lead to additional grants.

In contrast, as the separable output-first implementation propagates all input-side requests, it continues to generate more grants as the request rate approaches its maximum; however, due to the allocation inefficiencies described in Section 3.2, its matching quality remains below that of the wavefront-based implementation.

Results for configurations with ten input ports exhibit the same overall characteristics; however, when comparing configurations with the same number of VCs, the differences between the wavefront-based implementation on the one hand and the separable implementations on the other hand are slightly more pronounced, while the differences between the two separable implementation variants are less pronounced. We omit the corresponding figures in the interest of brevity.



(a) Mesh, 4 VCs.



(b) FBfly, 8 VCs.

Figure 5.7: Impact of switch allocation on saturation throughput.

5.5.3 Network-Level Performance

We first compare the performance of different allocator types using a non-speculative router implementation. Figure 5.7 shows the saturation throughput for individual synthetic traffic patterns. We present results for two network configurations with multiple VCs assigned to each traffic class; configurations where the total number of VCs is significantly smaller than the number of ports generally yield only minimal performance differences between allocator types, as the resulting request matrices are only sparsely populated. We omit detailed latency results as all three allocator variants exhibit the same behavior under low load.

In the Mesh, shown in Figure 5.7a, differences in matching quality between the different allocator implementations have minimal impact on overall network performance. This is primarily a result of routing restrictions, as the use of Dimension-Order Routing (DOR) results in an uneven distribution of output port requests: Because packets are only allowed to turn at the end of each dimension traversal, the vast majority of flits arriving at a given input port request the output port on the opposite side of the router. This leads to a sparsely populated request matrix, which—as we saw in Section 5.5.2—allows all three allocator variants to produce a near-maximum matching. Uniform Random (UR) traffic yields the biggest performance difference (3 %), as output port requests from different packets are least correlated in this pattern.

In contrast, using a wavefront-based allocator noticeably improves saturation throughput for the Flattened Butterfly (FBfly), as shown in Figure 5.7b. In FBfly networks, packets change their direction of travel after every hop and select an output port based on their destination coordinate in a given dimension; as such, requests are more evenly distributed across output ports than in the case of the Mesh. The performance differences are most pronounced for benign traffic, including the UR (11 %) and Nearest Neighbor (NN) patterns (7 %); the remaining adversarial traffic patterns exhibit smaller differences. This is because throughput for minimally routed traffic in these cases is limited by oversubscribed network channels; only the fraction

[11] Note that each VC selects a random output in our experiment.

of traffic that is adaptively routed through a random intermediate destination benefits from the wavefront allocator's higher matching quality. Nevertheless, the average improvement across traffic patterns is close to 6 %.

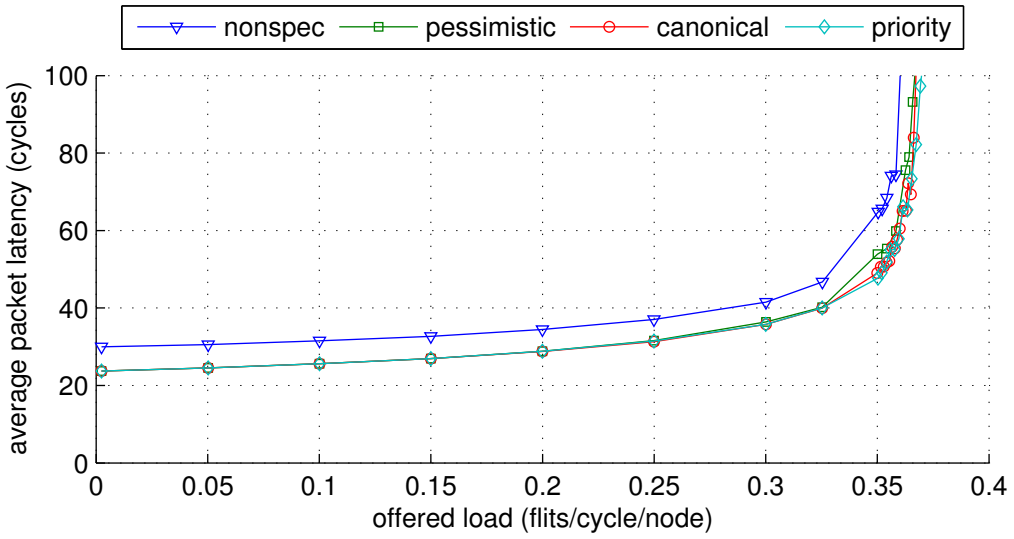
Figure 5.8 compares the efficacy of the speculation mechanisms described in Section 5.3 in a Mesh network. Results for FBfly networks follow the same trends; however, the reduction in zero-load latency is less pronounced as a result of the lower network diameter. We use separable input-first allocation and present detailed results for UR traffic; however, our qualitative observations hold for other switch allocators and synthetic traffic patterns.

Simulation results confirm that all three speculation schemes exhibit similar characteristics for low to medium levels of network load. In particular, they all reduce zero-load latency by 21 % compared to a purely non-speculative router design. With two VCs, speculation improves saturation throughput by 3 %, with minimal differences between the three speculation variants. As the number of VCs is increased to four, the performance of the pessimistic scheme and the priority-based scheme under heavy load approaches that of the non-speculative implementation. This is a result of speculative grants being eliminated by non-speculative requests that later fail to produce a grant. However, the throughput loss compared to the canonical implementation is smaller than 2 %.

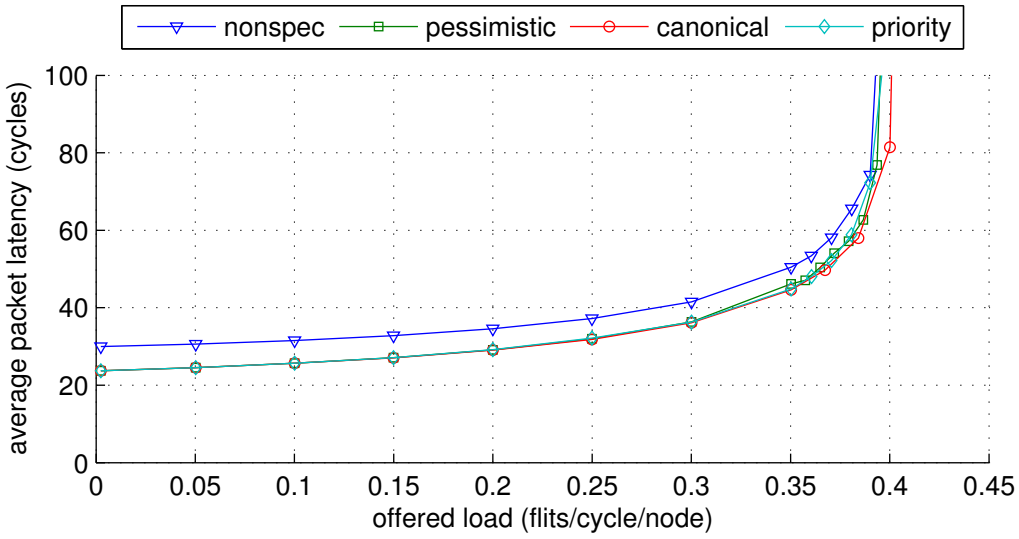
Overall, the improvements in saturation rate by themselves would likely not suffice to justify the complexity increase incurred by speculative switch allocation; however, the substantial reduction in latency at low to medium load is of significant benefit to many common applications of NoCs [81].

Finally, we evaluate the performance of combined VC and switch allocation. Figure 5.9 shows the load-latency diagram for a Mesh with two VCs. We prioritize requests from body and tail flits using the same basic mechanism as described in Section 5.3.3, and we compare the resulting performance to that of a non-speculative implementation, as well as to that of a priority-based speculative switch allocator. All design points are implemented as separable input-first allocators.

At low to medium network load, combined allocation yields the same latency improvement as speculation, as both effectively reduce the pipeline delay incurred at



(a) Mesh, 2 VCs.



(b) Mesh, 4 VCs.

Figure 5.8: Impact of speculative switch allocation on packet latency for UR traffic.

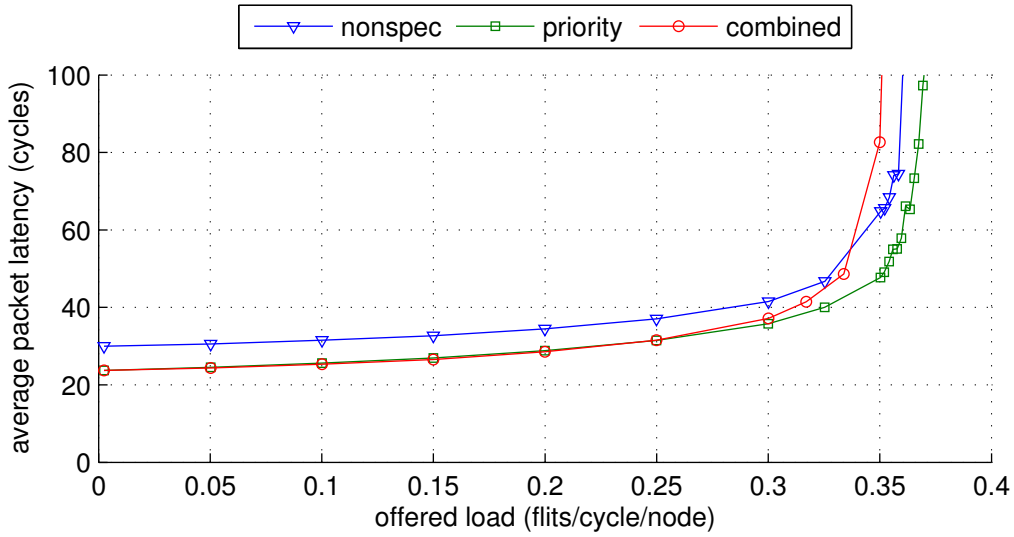


Figure 5.9: Impact of combined allocation on packet latency for Mesh (UR traffic).

each hop by one cycle. However, as network load continues to increase, the quality of allocation degrades and the saturation rate for combined allocation is reduced by 3 % and 5 % compared to the non-speculative and the speculative implementation, respectively.

This throughput degradation is primarily caused by an increase in unproductive requests at high load: Because body and tail flits are given priority over head flits, any newly arriving head flits whose destination port has body and tail flits waiting to be transmitted are queued. Once the last body or tail flit destined for a given output port is forwarded, all queued head flits that are waiting for that port compete for access in the next cycle. However, because a combined allocator does not assign an output VC to a head flit until after it wins allocation, we cannot eliminate requests for which no available downstream VC exists a priori; thus, under heavy load, there is an increased probability that a grant generated for one head flit will have to be discarded even though a suitable VC would have been available for another head flit.

In a canonical implementation that uses separate VC and switch allocators, speculative requests are in principle prone to the same effect. However, in such designs, any

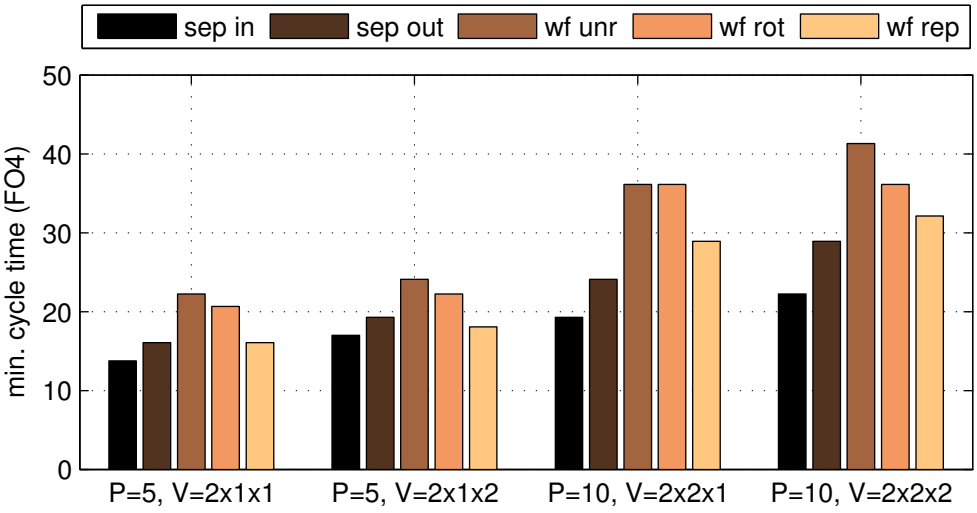


Figure 5.10: Minimum cycle time for switch allocator implementations.

buffered head flit can be assigned an output VC—and thus become non-speculative—without securing crossbar access first. Thus, as a result of over-provisioned VC allocator bandwidth (cf. Section 5.4), the number of speculative—and thus potentially unproductive—requests at high network load is smaller compared to combined allocation.

5.5.4 Delay and Cost

Figure 5.10 compares the minimum cycle time for different switch allocator implementations across a range of design points. We first focus on non-speculative implementations.

As in the case of VC allocation, the separable input-first implementation yields the lowest delay for all configurations, outperforming the separable output-first implementation by 12–23 % and the fastest wavefront-based implementation.

Despite its associated overhead, the replication-based wavefront allocator can operate at a lower minimum cycle time than both the unrolled and the rotation based

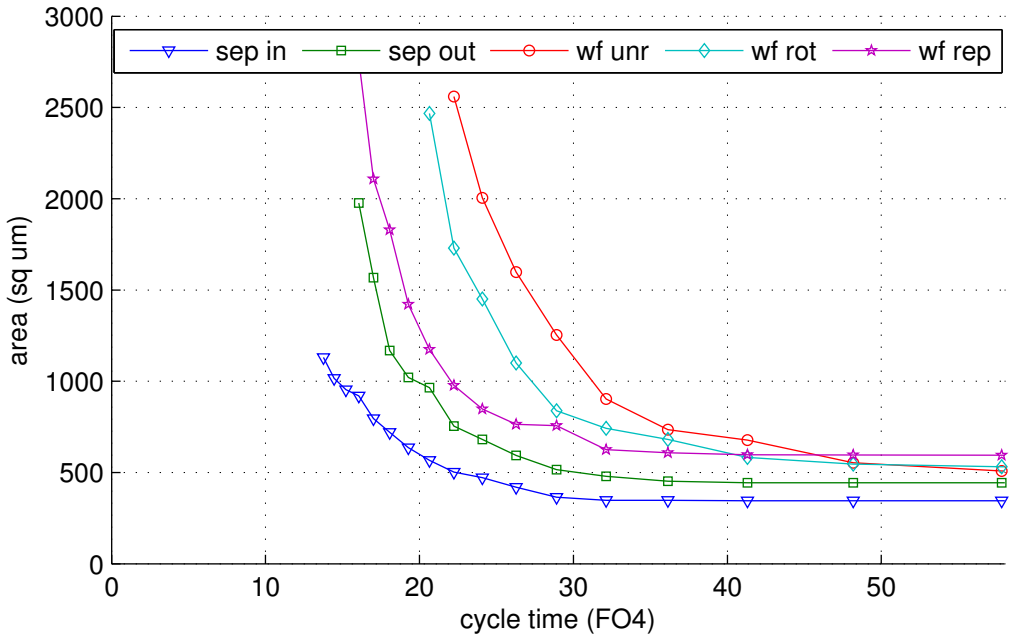


Figure 5.11: Area-delay trade-off for switch allocator implementations.

ones. For the five-port configurations, its delay exceeds that of the separable input-first allocator by 6–17 %; however, due to its inferior scaling behavior, the delay increase grows to 45–50 % when using ten ports. As such, despite its associated improvements in saturation rate, using a wavefront-based switch allocator in an FBfly is only attractive if external constraints impose a sufficiently long cycle time.

The minimum cycle times for both separable implementations closely match those for the corresponding VC allocator implementations if the optimizations described in Section 4.3 are applied. For the wavefront-based implementations, this is only true for the configurations with five ports; for the two larger design points, the VC allocator has significantly higher delay. This is explained in part by the fact that wavefront-based VC allocators cannot exploit restrictions in transitions between resource classes (cf. Section 4.3).

The area-delay trade-off—shown in Figure 5.11 for a five-port configuration with two VCs—largely reflects the trends established by the results for minimum cycle time; in particular, except for target cycle times greater than 45 FO4, the order of

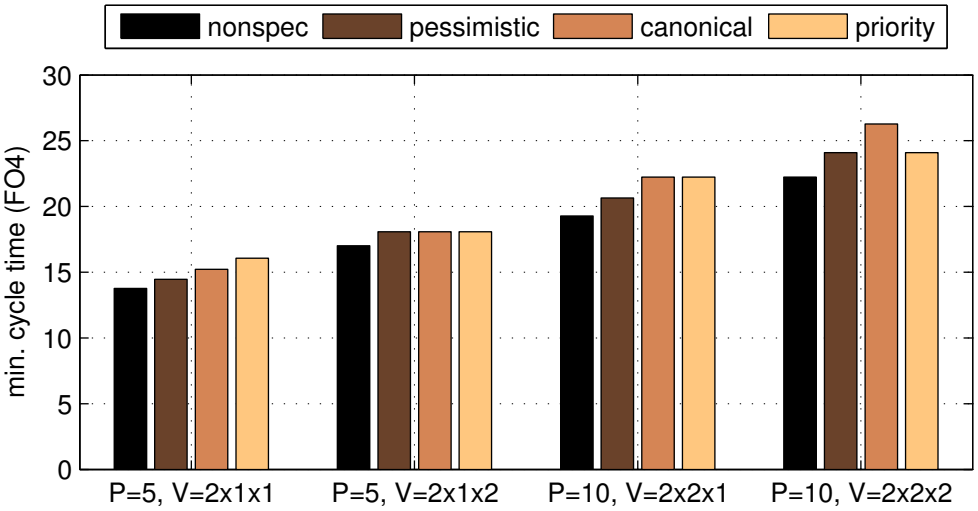


Figure 5.12: Minimum cycle time for speculation implementations.

implementations in terms of area efficiency is the same as that for delay, primarily as a result of gate sizing that synthesis must perform in order to meed timing constraints. Results for larger configurations as well as for Power-Delay Product (PDP) are qualitatively similar.

Figure 5.12 illustrates the delay impact of implementing speculative switch allocation; in particular, we compare implementations based on separable input-first allocation. The results for the non-speculative baseline implementation are identical to the ones shown in Figure 5.10.

Pessimistic speculation yields a delay reduction of up to 8 % compared to the canonical design, and it incurs an average delay increase of 7 % over the non-speculative implementation. Priority-based speculation incurs a slight delay increase at the smallest design point, but otherwise yields equal or better delay compared to the canonical implementation.

Figure 5.13 shows the trade-off between target cycle time and cell area for the five-port configuration with two VCs. While the pessimistic implementation and the canonical implementation exhibit similar characteristics when delay constraints are

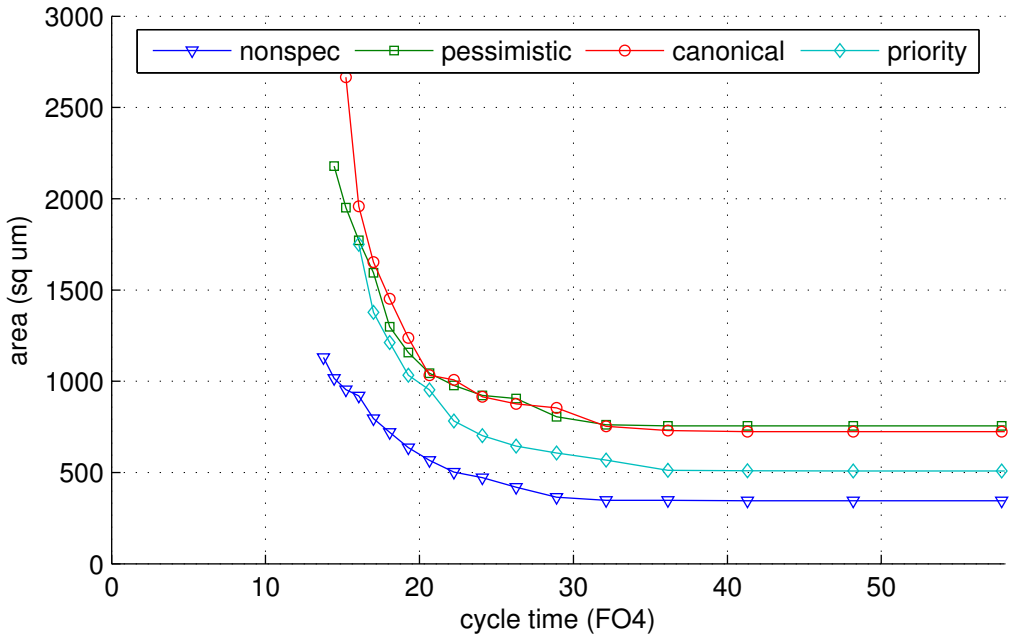
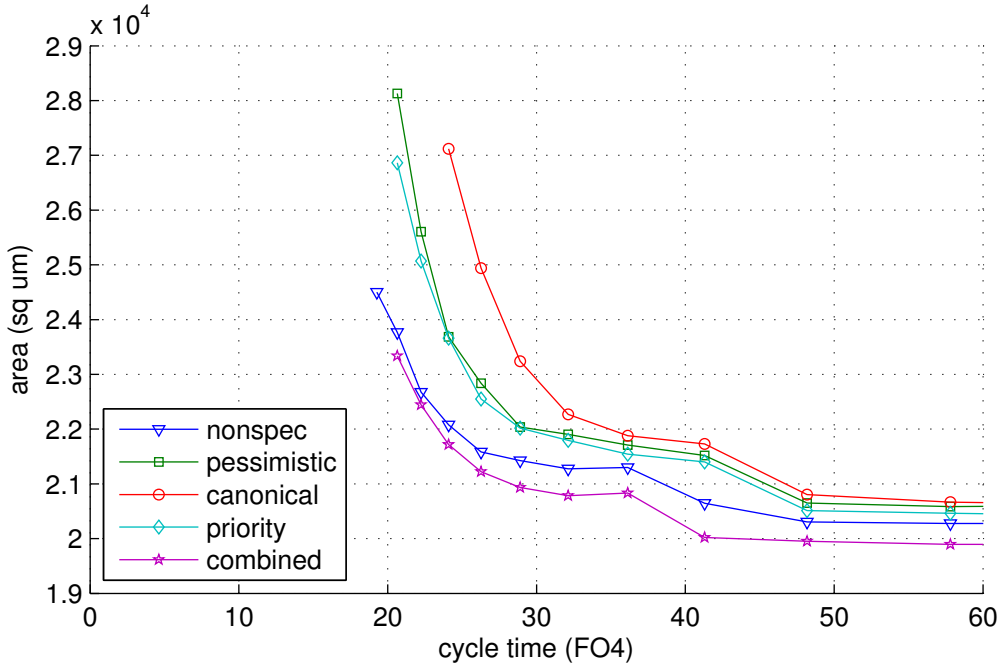


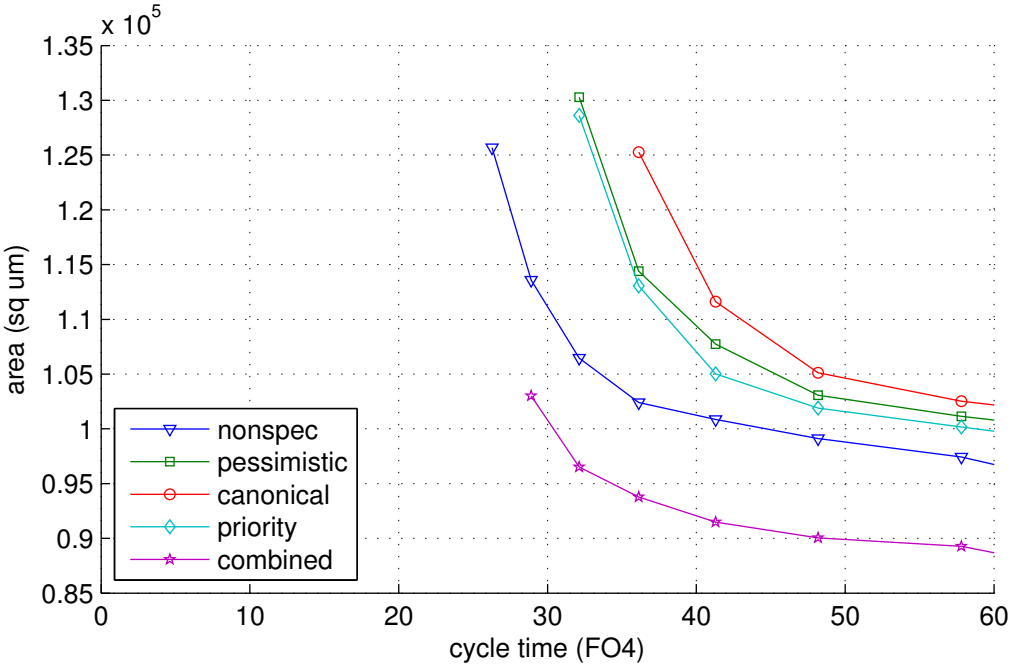
Figure 5.13: Area-delay trade-off for speculation implementations.

sufficiently loose to allow for the use of minimum-size gates, the pessimistic implementation requires less gate resizing to achieve more stringent delay constraints, and thus gradually becomes more area-efficient than the canonical implementation as the target cycle time decreases. Both require substantially more area than the priority-based implementation, as arbiter state is essentially replicated for the speculative and the non-speculative parts of the design. At target cycle times exceeding 25 FO4, sharing arbiter state allows the priority-based implementation to reduce area by 30 % compared to the canonical and pessimistic designs; compared to the non-speculative baseline, on the other hand, priority-based speculation incurs an area overhead of around 50 %. For more aggressive cycle times, the necessary gate resizing to meet timing constraints causes the cell area of the priority-based implementation to approach that of the pessimistic one.

To conclude our evaluation, we synthesize complete router instances. In addition to enabling us to evaluate combined VC and switch allocation, this allows us to more accurately capture the effects of peripheral logic required for generating requests and



(a) Mesh, 2×1×1 VCs.



(b) FBfly, 2×2×1 VCs.

Figure 5.14: Area-delay trade-off for complete router instances.

updating VC state in response to grants, and to consider interactions between VC and switch allocation.

Figure 5.14 shows the area-delay trade-off for a non-speculative baseline router, router instances using the three variants of speculative switch allocation described in Section 5.3, as well as a router using combined VC and switch allocation as described in Section 5.4. All implementations—including VC allocators where applicable—are based on separable input-first allocators. We consider two configurations: A five-port router with two VCs—one per message class—as used in typical Mesh networks, as well as a router configuration with ten ports and four VCs that is representative of FBfly networks. As in the case of individual allocator instances, PDP results are largely consistent with the qualitative observations for area.

For both design points, pessimistic and priority-based speculation are both faster and more area-efficient than the canonical implementation. Priority-based speculation has a slight area advantage because fewer registers are used to maintain arbiter state; however, the difference between the two designs is small compared to the total area of the router. In contrast, by eliminating the dedicated VC allocator, the use of combined VC and switch allocation substantially reduces the router's overall cell area.

For the Mesh, pessimistic and priority-based speculation both improve delay by 14 % compared to the canonical implementation. Combined allocation achieves the same minimum delay while reducing overall router area by 13 % and 17 %, respectively. All three cases incur a 7 % delay penalty compared to the non-speculative implementation.

In the FBfly configuration, combined allocation allows the router to achieve clock frequencies of up to 1 GHz; the corresponding cycle time (29 FO4) represents a 10 % reduction compared to the fastest speculative design overall, and a 36 % reduction compared to the fastest speculative design with the same area. Furthermore, at the maximum target frequency, combined allocation reduces the router's overall cell area by 9 % compared to the non-speculative baseline implementation.

Due to the VC allocator's scaling behavior (cf. Section 4.3), increasing the number of VCs by assigning additional VCs to each packet class further increases the delay

and area benefits afforded by combined allocation; we omit detailed results in the interest of brevity.

5.6 Related Work

Speculative switch allocation as a means of improving network performance was originally proposed by Peh and Dally [76, 77]. In particular, these contributions describe the canonical implementation outlined in Section 5.3.1. The authors present an analytical delay model for a separable implementation, but do not investigate area or power; since the model does not account for wire delay, its results are overly optimistic for modern process technologies.

Mukherjee et al [61] compare the performance of several switch allocator implementations in the context of a system-level interconnection network. Their study assumes that routers are deeply pipelined, and two of the allocators considered require multiple cycles to produce each matching; both factors would be undesirable in latency-sensitive NoCs. Furthermore, the different allocators are compared purely in terms of network performance; in particular, the analysis does not consider area or power, both of which represent first-class design considerations in NoCs.

Mullins et al. [62] reduce the pipeline latency of a VC router by precomputing arbitration decisions in a separable allocator one cycle in advance. While this scheme effectively removes the switch allocation stage from the router's critical path for buffered flits, it is less effective in the absence of congestion, as conflicts between newly arriving flits can result in unused crossbar time slots.

The row/column decoupled router introduced by Kim et al. [46] features an efficient mirror allocation scheme; however, this scheme is not directly applicable to generic router designs.

Kumar et al. [47] describe a scheme for combined VC and switch allocation that dynamically transitions between input- and output-first operation based on network load. Their design explicitly prioritizes subsequent flits from the same packet, and uses FIFO queues to implement both input-side arbitration and VC selection.

Park et al. [74] introduce a mechanism that prioritizes flits traveling along frequently used paths in the network, as well as a method to allow such flits to bypass the switch allocation pipeline stage entirely by sending arbitration requests ahead of time.

Seo and Thottethodi [83] implement maximum-size allocation in minimally routed Mesh networks using a lookup table of precomputed matchings. While table size can be reduced by filtering requests and exploiting routing restrictions, the proposed approach quickly becomes infeasible as the radix of the router grows. Furthermore, an additional mechanism is required to mitigate starvation issues as described in Section 3.4.

More recently, Ahn et al. [1] and Michelogiannakis et al. [57, 58] have proposed schemes that improve matching quality for separable switch allocators by reusing crossbar connections for multiple successive flits, effectively allowing efficient matchings to build up incrementally. These efforts are complementary to the work presented in this chapter.

5.7 Summary

In the present chapter, we have evaluated three exemplary switch allocator implementations and investigated several approaches for reducing the router's pipeline delay.

Following the evaluation methodology established in Chapter 4, we compare matching quality for individual allocator instances and find that wavefront allocation yields superior matching in the presence of congestion, particularly for configurations with large numbers of ports and VCs; in contrast to VC allocation, such congestion can be sustained for indefinite periods of time if network load is high. Nevertheless, simulation results for network-level performance indication that the improved matching quality afforded by wavefront allocation does not translate to significant performance improvements for Mesh networks. In contrast, for higher-radix networks like the FBfly, wavefront allocation measurably improves throughput for benign traffic if a sufficiently large number of VCs is used; in such cases, the use of a wavefront allocator is particular attractive if externally imposed timing constraints mask its

comparatively high delay.

In evaluating the network-level performance impact of speculative switch allocation, we find that a substantial reduction in latency stands in contrast to a merely marginal improvement in saturation throughput. Based on the realization that the impact of speculative switch allocation is most significant at low to medium network load, we have developed two modified speculation mechanisms that improve delay, area and energy efficiency compared to the canonical implementation at the cost of a slight reduction in saturation throughput.

Combined allocation affords the same latency improvements as speculative switch allocation, but reduces the router's overall area requirements and critical path delay by eliminating the dedicated VC allocator. However, this is achieved at the cost of a further reduction in saturation throughput caused by allocation inefficiencies at high load. Nevertheless, due to its low cost and delay, combined allocation—implemented using separable input-first allocators with integrated priority support—represents an attractive design choice for a wide variety of network configurations.

Chapter 6

Buffer Management

6.1 Overview

Proper buffer sizing and organization are essential to achieving optimal network performance [40, 66, 80]. At the same time, input buffers account for a large fraction of the overall area and power budget of typical Network-on-Chip (NoC) routers [15, 39, 96, 98]. Consequently, buffer resources must be utilized efficiently in order to achieve good cost-performance trade-offs. To this end, prior research has proposed dynamic buffer management schemes in which a pool of buffer slots is shared between Virtual Channels (VCs) [47, 66, 75, 90].

In the present chapter, we explore the key design parameters pertaining to the organization and management of router input buffers and evaluate their effect on network performance. We discuss important considerations in slicing and sizing input buffers and compare different buffer management mechanisms in terms of their impact on the router's cost and performance.

6.2 Buffer Organization

We first discuss the high-level design trade-offs involved in choosing the appropriate number and depth of VCs that the buffer is divided into and briefly outline how these choices affect network performance.

6.2.1 Number of VCs

VCs serve dual purposes in interconnection networks: On the one hand, they enable flits from different packets—even within a single flow—to bypass each other, e.g. when one of them becomes blocked. Doing so reduces Head-of-Line (HoL) blocking and improves channel utilization as well as overall network throughput. On the other hand, different flows of traffic can be constrained in which subset of a network's VCs they are allowed to travel on; this provides a means of separating different flows of traffic, which in turn can be used as the basis for implementing deadlock avoidance schemes or Quality-of-Service (QoS) policies.

By enabling flits from one packet to pass other packets' flits, VCs provide performance benefits in the same way that adding dedicated turn lanes to an intersection can improve the flow of traffic in street networks; in particular, they allow flits that have become blocked due to contention elsewhere in the network to be bypassed by other flits which are not directly affected by this contention, enabling them to make use of channel bandwidth that would otherwise be left idle [17].

The number of VCs also directly determines the number of requests that are exposed to the VC and switch allocator. The degree to which this affects network performance varies with the topology and the characteristics of the routing function. For example, in Flattened Butterfly (FBfly) networks [45], packets make turns at each hop. Consequently, different packets stored in a given input buffer are likely to request different output ports, and therefore increasing the number of VCs increases the average number of candidate output ports that a given input port can use. For allocators whose matching efficiency improves with the number of exposed requests—e.g. wavefront allocators (cf. Chapter 3)—the additional requests provide more freedom in finding an optimal matching between inputs and outputs. However, this effect reaches a point of diminishing returns once the number of active VCs becomes larger than the number of output ports.

On the other hand, in Mesh networks using Dimension-Order Routing (DOR), packets are more likely to continue traveling in the same direction than to take a turn at any given router. Consequently, requests from different packets in a given input buffer tend to be highly correlated, and therefore exposing additional requests is less

likely to lead to an increase in the number of candidate output ports.

Different types of packets which are subject to interdependencies at the edge of the network—e.g. request packets and the reply packets that they generate at their destination—must be isolated from one another in order to prevent protocol deadlock [20]. This is readily achieved by preventing the different types of packets from using the same VCs, effectively assigning VCs to different *message classes*. The required number of messages classes—and, by extension, of VCs—depends on the specifics of the communication protocol as well as implementation details of the network interface.

Similarly, cyclic resource dependencies within the network, which can form e.g. as a result of routing, can be prevented by further grouping VCs into different *resource classes*, with transitions between the latter being restricted such as to enforce a partial order of resource acquisition. As an example, this approach is used for deadlock-free dateline routing in torus networks [20] and multi-phase routing algorithms such as Valiant's algorithm [94] or Universal Globally Adaptive Load-Balanced (UGAL) [87]. The required number of resource classes depends on the network topology and the routing algorithm.

While increasing the number of VCs beyond the minimum number required to achieve the desired deadlock avoidance properties generally leads to better performance, it also increases router cost as each VC requires control logic, state and—in the case of statically managed buffers—additional buffer space to satisfy minimum capacity requirements. The number of VCs also directly affects the complexity of VC allocation and therefore the router's cost and, potentially, cycle time, as we explored in Chapter 4.

6.2.2 Maximum VC Capacity

Network latency under modest load represents a critical performance metric for many typical NoCs applications [81]. To minimize latency in the absence of significant congestion, it is important to avoid other causes for stalls. in particular, in order to avoid credit stalls,it is important for individual VCs to have enough buffer space

available to cover the credit round-trip time. This allows flits from a single packet of arbitrary length to be transmitted back-to-back without introducing fragmentation.

In cases where packet size is limited and known a priori, the same effect can be achieved by making individual VCs deep enough to be able to hold an entire packet; however, while this avoids credit stalls when transmitting a single packet at a time, such stalls can still occur when transmitting a burst or continuous stream of back-to-back packets. Unless a large enough number of VCs is available to cover the credit round-trip delay with their aggregate capacity, this both increases packet latency and reduces the achievable throughput.

As network load—and therefore congestion—increases, the capacity of individual VCs becomes less important, as congestion leads to an increase in overall latency, and therefore reduces the impact of fragmentation. Furthermore, in the presence of multiple VCs with pending flits, any credit stalls incurred by one of them are unlikely to affect throughput, as there is a high probability that another VC is ready to go; thus, full throughput can be achieved as long as all VCs collectively have sufficient capacity to cover the credit round-trip delay.

The total capacity across all VCs also limits the number of in-flight flits, and thus the amount of contention that can be tolerated before back pressure must be applied to the upstream router. Higher aggregate capacity allows temporary contention to be contained at the local router, and thus reduces the incidence of tree saturation in the network [78]. However, the same effect also reduces the stiffness of the flow control feedback loop, which is undesirable for adaptive routing algorithms that rely on this feedback for their routing decisions. Furthermore, for any given network load, there is a point of diminishing returns beyond which additional increases in total capacity simply reduce average buffer utilization without improving performance.

6.2.3 VC Reallocation Policy

Another key design parameter for buffer management is the policy governing the reuse of VCs. Canonical router designs with statically partitioned input buffers typically allow a VC to be reused by the next packet as soon as the previous packet's tail flit

has been sent. As a result, multiple packets may be queued up behind one another in each VC, up to the limit of the VC's capacity. This enables buffer configurations with relatively few VCs to support a large number of in-flight packets. On the other hand, because only the first packet in each VC participates in allocation, this approach can increase HoL blocking by preventing subsequent packets from making forward progress if the first packet becomes blocked.

In order to minimize HoL blocking, we can limit reuse such that each VC can hold at most one packet at any given time [66]; this is referred to as *atomic VC allocation*. In practice, atomic VC allocation is easily enforced by requiring all of a VC's credits to be returned to the upstream router before making it available for re-allocation. The disadvantage of this scheme is that it inherently limits the number of in-flight packets at each router input and output port to the number of VCs. Consequently, it is best suited for configurations with many VCs.

Certain routing functions, notably those that rely on escape VCs for deadlock avoidance, require VCs to be allocated atomically.

6.3 Static Buffer Management

As a result of the stringent timing constraints that the NoC environment typically imposes, much prior work has opted to implement low-complexity buffer management schemes which statically divide the available buffer space among all VCs.

While such schemes minimize control logic complexity, they are prone to buffer under-utilization when network load is not evenly distributed across VCs. Furthermore, in order to avoid credit stalls, each VC individually must be assigned at least enough buffer space to be able to cover the credit round-trip delay, causing buffer requirements to scale linearly with the number of VCs while further reducing the average utilization factor. Consequently, such implementations typically only support a modest number of VCs.

Statically managed buffers are readily implemented as circular buffers, with management logic essentially comprising a pair of index registers per VC that point to the buffer locations which will be read from and written to next, respectively.

For efficiency reasons, it is typically beneficial to consolidate storage for all VCs into one large buffer structure instead of using a separate buffer for each VC. In a standard cell implementation, this does not affect the delay for the buffer write and read paths, as flits for all VCs at a given input port arrive through the same input register and depart through the same crossbar port; as such, the larger buffer merely subsumes external demultiplexing and multiplexing logic that is necessary in the case of separate buffers per VC.

Credit tracking at the router's output ports can be implemented as a set of counters that track the number of occupied buffer entries for each VC; counters are incremented whenever a flit destined for a given VC wins switch allocation and decremented in response to incoming credits from the downstream router. If the counter for a given output VC reaches its maximum value, no further flits can be sent to that VC until one or more credits return.

6.4 Dynamic Buffer Management

In practice, network load is typically not evenly distributed across all VCs. For example, with request-reply traffic [4], both packet classes consist of one short—read requests and write replies—and one long—read replies and write requests—packet type; however, many common workloads generate more read transactions than write transactions. As a result, the VCs assigned to the packet class used by replies tend to see a higher effective load than those carrying requests. Similarly, for multi-phase routing algorithms like UGAL [87] or dateline routing, network traffic tends to be unevenly distributed across the corresponding resource classes.

Dynamic buffer management schemes improve buffer utilization in such scenarios by allowing available buffer space to be shared among VCs. This offers several key advantages over static approaches:

- Assigning buffer space to VCs based on demand increases the average buffer utilization and thus facilitates more efficient use of expensive buffer resources.

- Furthermore, by decoupling flit storage from VCs, the differential cost of increasing the number of VCs is reduced. As a result, a dynamically managed input buffer of a given size can support more VCs—as many as one per buffer entry—than would be feasible with static partitioning.

- Finally, dynamic buffer management allows the effective number and depth of VCs to vary based on network conditions. Specifically, at low network load, a dynamically managed buffer can provide a small number of active VCs that are each sufficiently deep to cover the credit round-trip delay, whereas at high load, the same buffer can accommodate a multitude of shallow VCs that collectively reduce HoL blocking [66, 80].

Together, these benefits allow a dynamically managed buffer to either improve performance for a given buffer size or achieve comparable performance to a statically partitioned buffer at lower cost; for example, Nicopoulos et al. [66] report up to 25% higher performance or up to 50% reduction in buffer size for a 64-node mesh network.

6.4.1 Implementation

Prior research has proposed several implementation alternatives for dynamically managed router input buffers. From a functional perspective, all variants exhibit the same overall behavior: They manage multiple variable-length queues—one per VC in our case—and allow flits to be removed from the head or added to the tail of each. However, the individual variants differ in how the order of flits is preserved within each queue. Practical designs largely fall into three major categories:

Linked-list based [90]: Tamir and Frazier initially proposed a design that uses a hardware linked-list implementation to track the order of flits and their assignment to different queues in the buffer. In such implementations, each queue maintains a head and tail pointer into the buffer, and flit order is established by adding a successor pointer to each buffer entry[1]. A separate list tracks the set of currently unused buffer slots which are dispensed to arriving flits on demand.

[1] In practice, these successor pointers are typically stored in a separate array, as appending them to the flit buffer entries directly necessitates the inclusion of a second write port.

Table-based [47, 66]: In order to avoid the control logic necessary to update successor pointers, a second implementation variant stores pointers to each VC's individual in-flight flits in a packet table; unoccupied buffer entries are tracked using a bit mask. While this approach simplifies flit insertion and removal, it increases the total number of pointer registers, as each table entry must include enough pointers to accommodate the maximum capacity for an individual VC[2]. Conversely, the fixed table entry size limits buffer occupancy for individual VCs, reducing overall flexibility.

Self-compacting buffers [52, 65, 75]: A third approach eliminates the need to maintain pointers to all buffer entries by storing each individual VC's flits in a contiguous buffer region in order of arrival and moving unoccupied entries to the top of the buffer. Every time a flit arrives at (departs from) the buffer, all occupied buffer entries above the point of insertion (removal) are shifted up (down) to ensure that the buffer regions assigned to individual VC remain contiguous. Under high load, when buffer occupancy is high, this leads to substantial dynamic power overhead; as a result, self-compacting buffers generally do not represent an attractive design choice for the power-constrained NoC environment, and we exclude them from further consideration in the remainder of this chapter.

If buffer space is shared freely, we can track credits using a single counter at each output port. However, as we will see in Section 6.5, practical implementations typically impose additional constraints on the distribution of buffer slots among queues. In such cases, it is generally necessary to provide individual occupancy counters for each downstream VC as in the statically partitioned case, as well as a separate counter for the total number of shared buffer slots that are currently occupied.

Table 6.1: Buffer management implementation cost.

Description	Static	Linked-list	Packet table
Buffer control			
VC pointers	$2{\times}V{\times}\lceil log_2(\lfloor B/V \rfloor)\rceil$	$2{\times}V{\times}\lceil log_2(B)\rceil$	$2{\times}V{\times}\lceil log_2(P)\rceil$
Flags	V	V	V
Free pointers	–	$2{\times}\lceil log_2(B)\rceil$	–
Free mask	–	–	B
Buffer pointers	–	$B{\times}\lceil log_2(B)\rceil$	–
Output VC state			
Busy/empty flags	$2{\times}V$	$2{\times}V$	$2{\times}V$
VC counters	$V{\times}\lceil log_2(\lfloor B/V \rfloor)\rceil$	$V{\times}\lceil log_2(B)\rceil$	$V{\times}\lceil log_2(P)\rceil$
Free counter	–	$\lceil log_2(B)\rceil$	$\lceil log_2(B)\rceil$

6.4.2 Overhead

Table 6.1 shows a breakdown of the implementation overhead for the different buffer management schemes as a function of the buffer size (B), the number of VCs (V) and the maximum packet length (P). In addition to the cost of the buffer control logic itself, we include the cost of the output-side VC state and credit tracking mechanism.

As a concrete example, Figure 6.1 compares the overhead for a 16-entry buffer with 64-bit flits and a maximum packet length of six flits; the white segments at the bottom of each bar correspond to the number of registers required for the actual flit buffer.

With a maximum packet length of six flits, the configuration with two VCs does not represent a feasible design point for the table-based buffer management scheme: The packet table only provides enough pointers to map twelve of the sixteen available buffer entries, leaving four entries unusable. For the remaining feasible design points, on the other hand, the linked-list based implementation consistently incurs less overhead; as such, we will use the latter mechanism throughout the remainder of this dissertation.

[2] Note that the primary benefit of dynamic buffer management is to allow each individual VC to use more than its fair share of the buffer's total capacity.

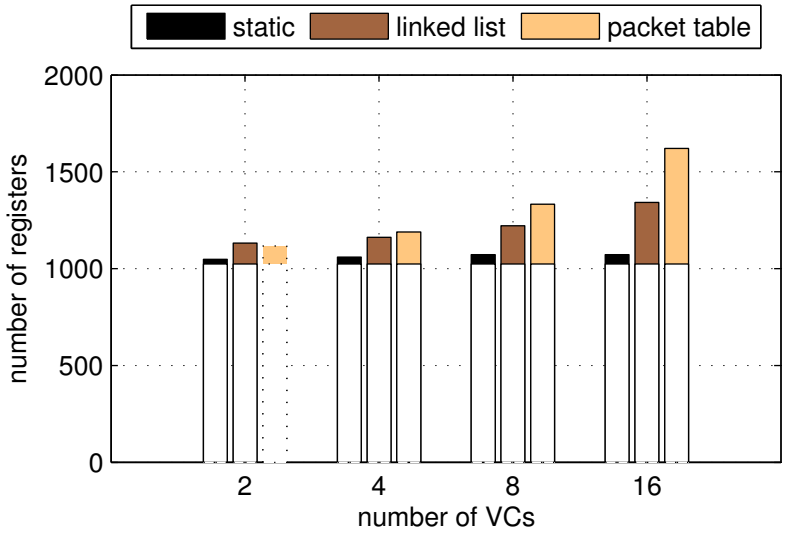


Figure 6.1: Buffer management overhead for a 16-entry buffer.

6.5 Deadlock Avoidance

Without additional restrictions, buffer sharing can allow cyclic dependencies to form between VCs and buffer slots when multiple packets are interleaved, resulting in network deadlock. Figure 6.2 shows an example of a deadlocked configuration involving two routers. For simplicity, we assume that each router input buffer has a capacity of two flits and can support up to two VCs; however, similar examples can be constructed for most reasonable network configurations with larger buffers or more VCs. In the depicted example, the flits from packets Ⓐ and Ⓑ have traveled from the upper router's western and eastern input, respectively, to the lower router's northern input, acquiring VCs on the way as indicated by the inscribed circles. Subsequently, packet interleaving has allowed flits from packets Ⓒ and Ⓓ to each acquire a VC at the upper router and to fill up all available buffer slots there, preventing additional flits from packets Ⓐ and Ⓑ from entering this router. Even after the two initial flits from packets Ⓐ and Ⓑ leave the lower router and relinquish the buffer at its northern input, packets Ⓒ and Ⓓ are unable to proceed as both VCs at the lower router are still occupied by packets Ⓐ and Ⓑ. The remaining flits from packets

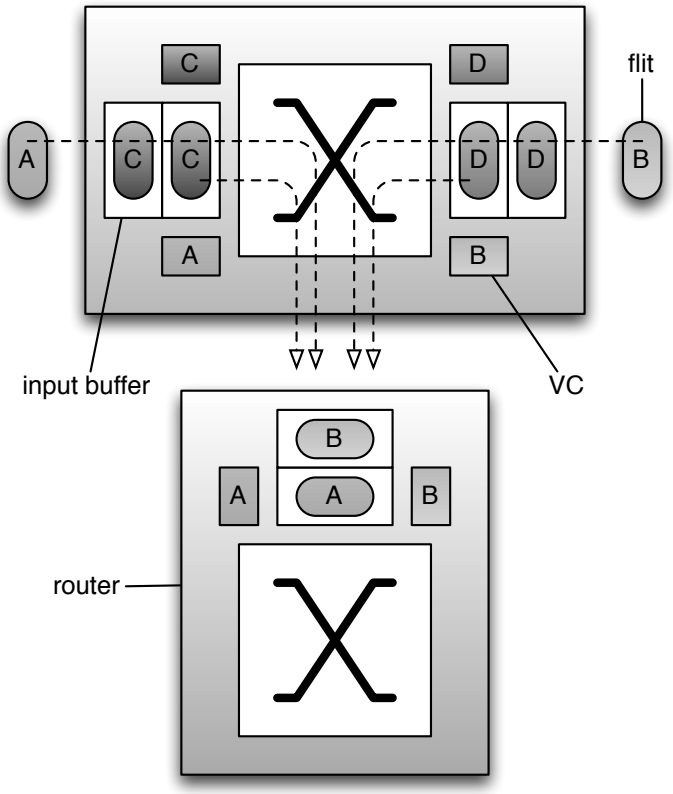


Figure 6.2: Buffer sharing causes interleaving deadlock.

Ⓐ and Ⓑ are also unable to make progress, as all buffer space at the upper router is occupied by packets Ⓒ and Ⓓ. Since the VCs at the lower router can only be freed after all remaining flits from packets Ⓐ and Ⓑ have passed through, a cyclic dependency has formed and the network has become deadlocked.

We can avoid interleaving deadlock in a straightforward manner by statically reserving at least one buffer slot for each VC. This guarantees that body and tail flits from one packet can bypass flits from other packets in order to catch up to their respective predecessor flit. Since each VC is able to make forward progress, protocol and routing deadlock avoidance can be implemented in the same way as for a statically managed buffer.

However, for configurations with large numbers of VCs, statically reserving buffer slots for all VCs significantly reduces the amount of buffer space that is available for sharing, relinquishing some of the utilization benefits that dynamic buffer management provides. For such configurations, we can avoid interleaving deadlock in a more space-efficient manner by exploiting the fact that buffer space needs to be reserved for a given packet if and only if some but not all of its flits have been sent downstream. Consequently, interleaving deadlock can be prevented by dynamically reserving a buffer slot for a given VC whenever a head flit is sent to it and releasing it once the corresponding tail flit is sent. In this way, buffer space is only reserved for as long as a packet is in flight, and otherwise remains available to other VCs.

When employing this dynamic reservation scheme, a VC that does not currently have any buffer space assigned is unable to accept any new flits if all shared buffer slots are in use by other VCs. Thus, in order to avoid protocol and routing deadlock, we must not only constrain the set of VCs that a given packet is allowed to use based on its message and resource class as outlined in Section 6.2.1, but also ensure that each class has buffer space available at all times. This is easily achieved by statically reserving one buffer entry for the range of VCs assigned to each message and resource class. Because the number of such classes required for deadlock avoidance in practice tends to be much smaller than the number of available buffer entries, such reservations do not significantly reduce the amount of buffer space that can be shared among VCs.

6.6 Evaluation

6.6.1 Experimental Setup

As in previous chapters, we evaluate network performance for different buffer configurations using a modified version of the BookSim 2.0 interconnection network simulator.

We model memory traffic that comprises request and reply packets for both read and write transactions. Read requests and write replies consist of a single head flit, followed by an additional flit which carries the memory address being accessed. Read replies and write requests additionally carry four flits containing payload data. Nodes generate traffic with a 2:1 ratio of read requests to write requests.

Nodes inject request packets into the network according to a Bernoulli process with configurable arrival rate. Upon reaching their destination node, requests trigger injection of a corresponding reply packet in the next cycle, which then proceeds back to the source node. Injection of replies takes priority over the injection of new request packets.

To evaluate network performance both under benign and under adversarial load conditions, we select destinations for request packets according to a set of synthetic traffic patterns including Bit Complement (BC), Bit Reverse (BR), Nearest Neighbor (NN), Shuffle (SH), Tornado (TO), Transpose (TR) and Uniform Random (UR) [20]. Throughput numbers correspond to the *effective* throughput *for the specified traffic pattern*, which is given by the minimum measured throughput across all source-destination pairs in the pattern. Packet latency is measured from the time at which a packet is placed in the injection queue to the time at which its tail flit is ejected from the network; i.e., reported latencies include source queuing delay.

We consider two exemplary 64-node network topologies: A Mesh network with 8×8 routers, each of which connects to a single node, as well as a two-dimensional FBfly [45] with four nodes per router. Network channels have a delay of a single cycle for the Mesh and between two and six cycles for the FBfly; ingress and egress channels in both topologies have a delay of one cycle. The Mesh uses DOR, while the FBfly uses UGAL routing [87].

Both networks use input-queued routers and VC flow control. The router pipelines comprise two stages: The first pipeline stage implements combined VC and switch allocation as described in Section 5.4, while the second one is reserved for crossbar traversal. Lookahead routing [28] removes the need for a dedicated routing stage. After a credit arrives at a router, processing and internal signal propagation cause an additional two-cycle delay before the associated buffer slot becomes visible to the switch allocator.

We consider the following three buffer management schemes in our evaluation:

Static: The available buffer capacity is evenly divided among all VCs.

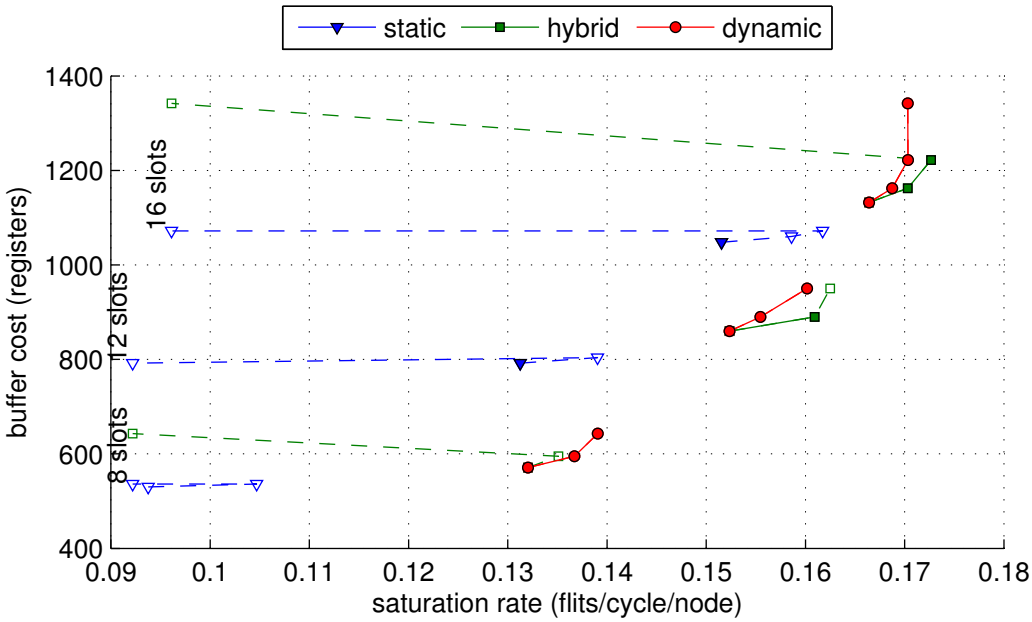
Hybrid: One buffer slot is statically assigned to each VC for deadlock avoidance, while the remaining buffer slots are shared among all VCs. As the reserved buffer slots logically correspond to a statically partitioned subset of the buffer, this effectively represents a hybrid of the two remaining schemes.

Dynamic: Buffer slots are shared among all VCs, and one slot is retained by each active VC as described in Section 6.5. In addition, in order to prevent protocol and routing deadlock, we statically reserve a single buffer slot for the range of VCs assigned to each message and resource class.

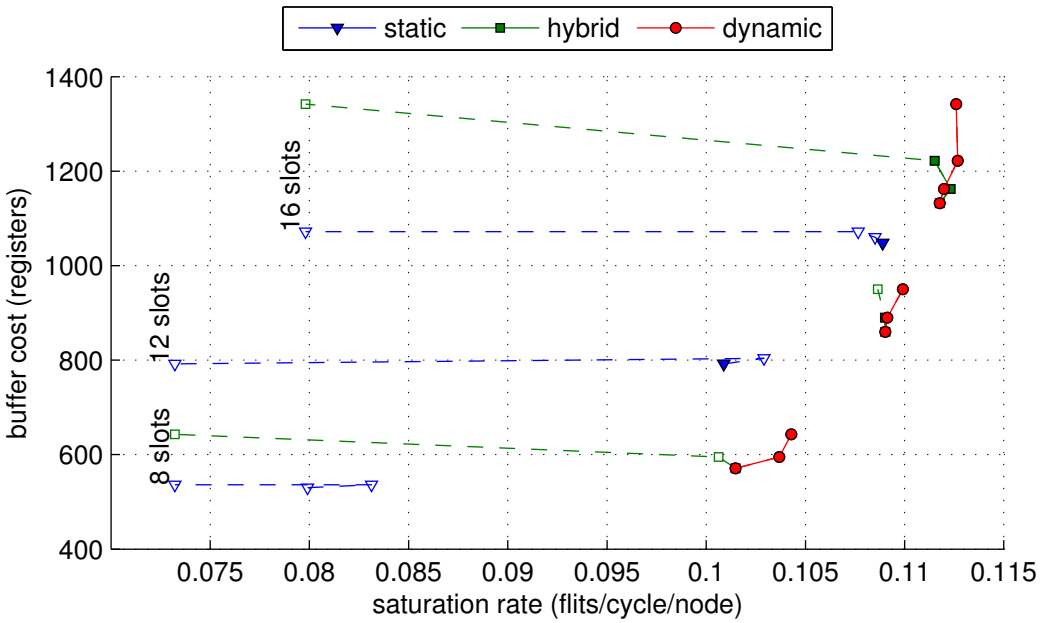
Unless otherwise noted, we allow each VC to be assigned to a new packet as soon as the previous packet's tail flit enters the crossbar traversal stage; Section 6.6.3 explores the performance impact of using atomic VC allocation.

6.6.2 Cost-Performance Trade-offs

Figure 6.3 illustrates the trade-off between buffer implementation cost and saturation throughput. Individual curves correspond to different buffer sizes as annotated. The first data point in each curve corresponds to a configuration with two VCs; each additional data point represents a doubling of the number of VC. Buffer cost is determined by computing the total number of registers required for flit storage and adding buffer management overhead as described in Table 6.1. We show results for UR traffic as an example of a benign traffic pattern (Figure 6.3a), as well as for the



(a) UR traffic.



(b) Harmonic mean across traffic patterns.

Figure 6.3: Cost-performance trade-offs for Mesh.

harmonic mean of saturation rates across all simulated traffic patterns (Figure 6.3b). Results for other individual traffic patterns are largely consistent with our qualitative observations. All reported cost numbers assume a flit width of 64 bits.

In order to minimize latency in the absence of congestion, we are particularly interested in configurations that provide sufficiently large maximum VC capacities to allow individual packets to be transferred without credit stalls; i.e., the maximum capacity for an individual VC must exceed the maximum packet size or the credit round-trip time, whichever is smaller. Design points that fail to meet this requirement are designated by unfilled markers and dashed lines in Figure 6.3; these design points generally exhibit increased latency at low to medium injection rates as a result of packet fragmentation.

Overall, results for UR traffic—shown in Figure 6.3a—and for the mean across all considered traffic patterns exhibit largely similar behavior; in particular, in both cases, the two dynamically managed configurations of a given buffer size yield comparable performance to the next larger statically partitioned buffer configuration: When comparing configurations with the same number of VCs, a 12-entry dynamically managed buffer yields comparable performance to a 16-entry statically partitioned buffer while reducing implementation cost by up to 22 %, while an 8-entry shared buffer outperforms a 39 % more expensive statically managed one with 12 entries.

For buffer implementations with a given capacity, dynamic buffer management is particularly beneficial when the overall buffer size is small. For example, with a capacity of eight entries, dynamic buffer management improves the maximum achievable saturation throughput for UR traffic across VC configurations by 33 % while increasing buffer cost by 20 %; alternatively, by choosing the same number of VCs as in the static configuration, the difference in buffer cost can be reduced to 11 % while maintaining a performance advantage in excess of 30 %.

As the buffer size increases, credit stalls become less frequent and thus have less impact on overall network performance. However, sharing can be attractive even for large buffers if external constraints necessitate the use of a large number of VCs; in particular, dynamic buffer management is beneficial if the maximum capacity for individual VCs—i.e., the buffer size divided by the number of VCs in the case of

a statically partitioned buffer—falls below the value necessary to support back-to-back transmission of flits from the same packet; such configurations are indicated by unfilled markers and dashed lines in Figure 6.3.

For a statically partitioned buffer, increasing the number of VCs for a given buffer size reduces each VC's maximum capacity proportionally; as such, VC configurations represent different trade-offs between zero-load latency—which favors deeper VCs—and saturation throughput—which favors more, shallower VCs. In contrast, for buffer sharing with dynamic reservations, increasing the number of VCs does not incur a latency penalty. However, beyond a relatively small number of VCs, the differential performance gains are generally smaller than the associated increase in buffer cost[3]. The hybrid implementation with static reservations exhibits performance characteristics that are close to those for the fully dynamic implementation when the number of VCs is relatively small. As more VCs are added, the total amount of buffer space that is reserved for individual VCs continues to increase, leaving fewer buffer entries available for sharing among VCs; consequently, the performance approaches that of the statically partitioned implementation[4].

Figure 6.4 and Figure 6.5 illustrate the effect of the number of VCs on zero-load latency for a Mesh with 16-entry input buffers: All configurations for which the effective maximum capacity for each VC suffices to avoid incurring credit stalls under low load—as indicated by the dashed line in Figure 6.4—yield the same zero-load latency. For the static scheme, increasing the number of VCs quickly reduces the per-VC capacity below this level, causing zero-load latency to increase as capacity drops further. The effect is less pronounced for the hybrid scheme, as each additional VC only reduces the number of shared buffer slots by one. In contrast, per-VC capacity—and hence, latency—remains unchanged as the number of VCs is increased in the fully dynamic configuration. Overall, dynamic buffer management thus allows a buffer of a given size to support a larger number of VCs without compromising zero-load latency.

The performance advantages afforded by dynamic buffer management are even

[3] As we discussed in Chapter 4 and Chapter 5, increasing the number of VCs furthermore directly affects the cost and critical path delay of the VC and switch allocation logic.

[4] In the extreme case where $V = B$, the hybrid scheme effectively degenerates into implementing static partitioning, as all buffer slots are reserved for individual VCs.

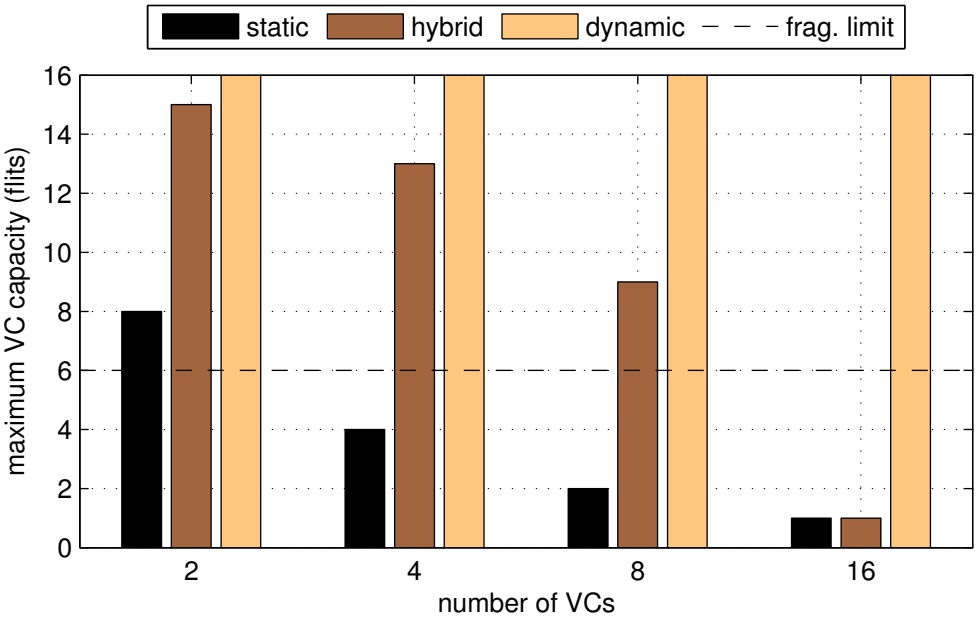


Figure 6.4: Maximum VC capacity for 16-entry buffer.

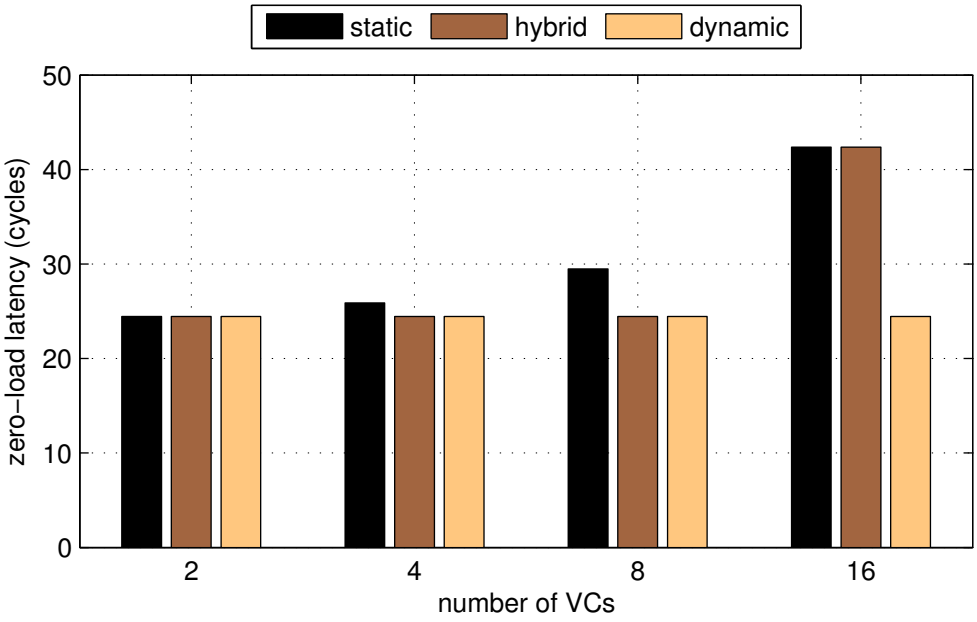


Figure 6.5: Zero-load latency for Mesh (16 buffer slots, UR traffic).

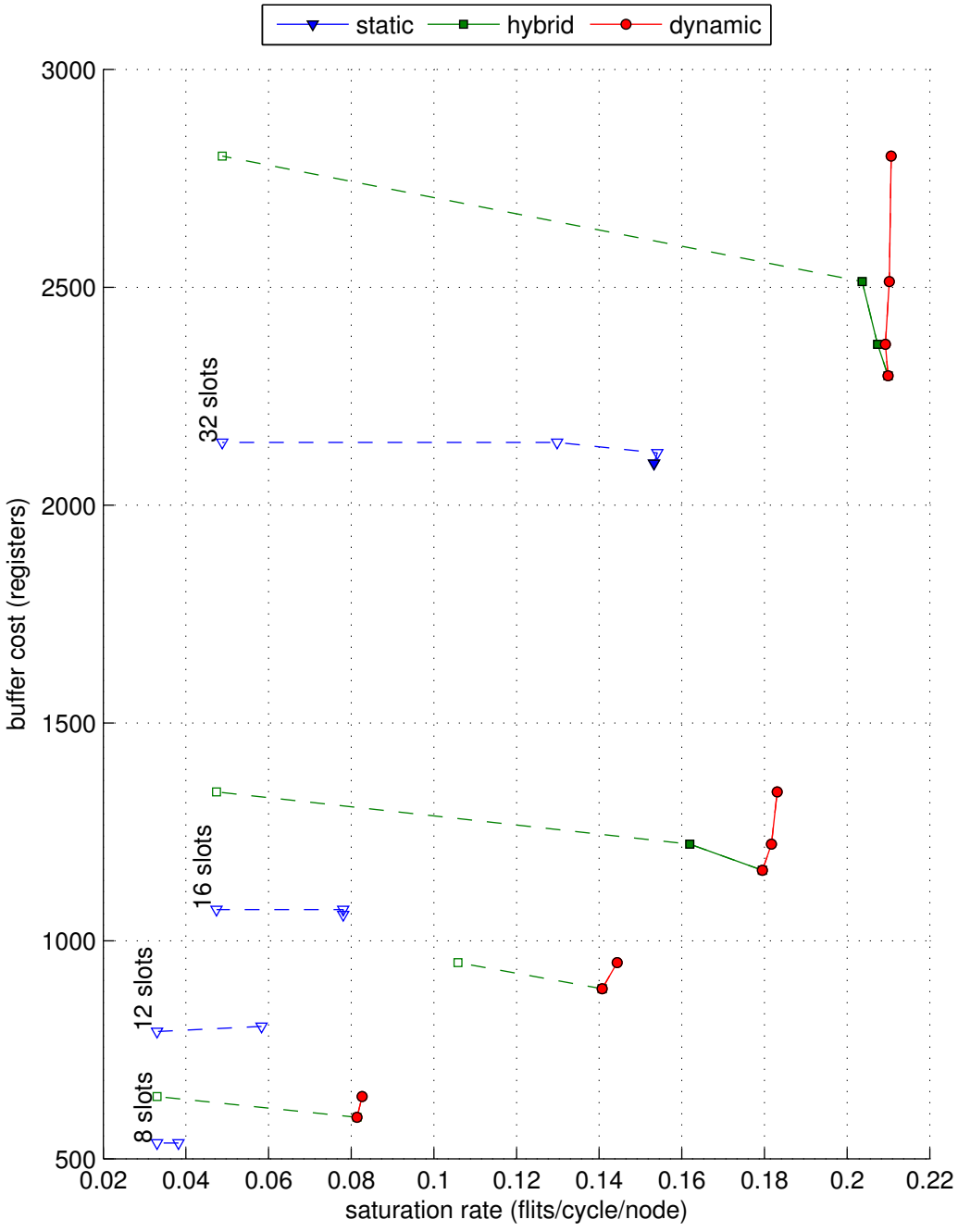
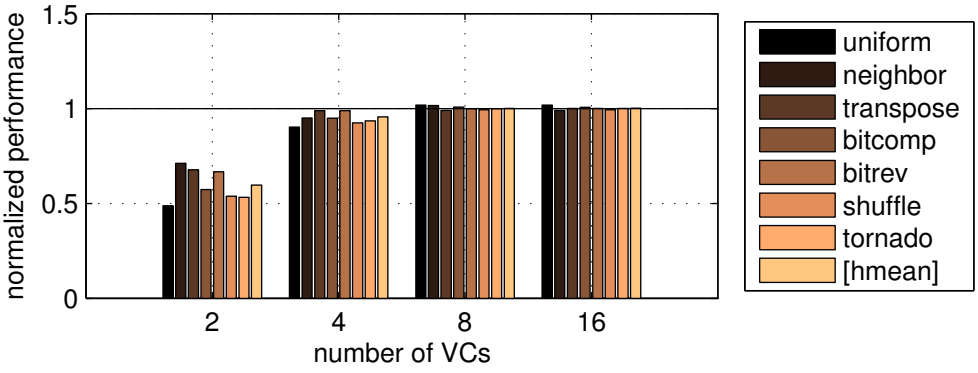


Figure 6.6: Cost-performance trade-offs for FBfly (all traffic patterns).

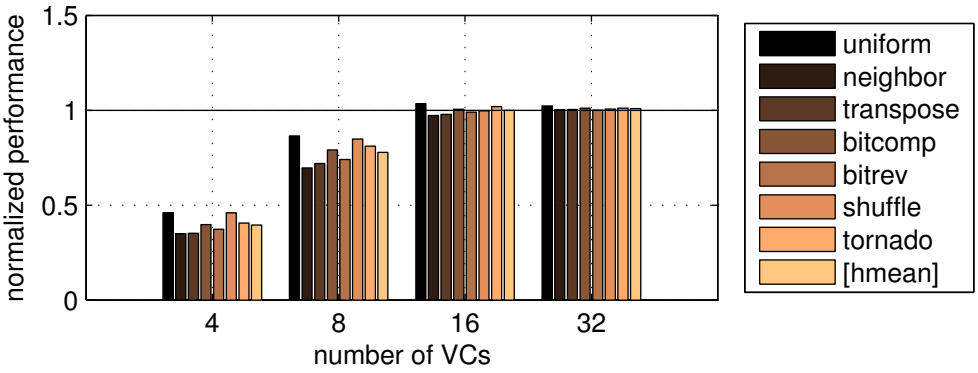
more pronounced in the case of the FBfly network, as shown in Figure 6.6. We report the harmonic mean of the saturation throughput for all simulated traffic patterns; the relative performance of the different buffer management schemes exhibits the same qualitative behavior for most individual traffic patterns. Because we employ UGAL routing on the FBfly network, which requires the use of two resource classes for deadlock avoidance, the first data point in each curve in Figure 6.6 corresponds to four VCs, compared to two in the case of the Mesh. In order to accommodate statically partitioned configurations with the same VC depth as before, we increase the maximum buffer size considered to 32 flits.

Due to the increase in the overall number of VCs, the implementation overhead for the hybrid and fully dynamic schemes is larger than in the Mesh. However, this is offset by substantial performance increases: When comparing configurations with four VCs, buffer sharing enables a 16-entry buffer to outperform a statically partitioned buffer with 32 entries—the only such configuration that has sufficient capacity per VC to allow packets to be transmitted without introducing fragmentation. In particular, the dynamically managed buffer achieves 17 % higher average saturation throughput while simultaneously reducing buffer cost by 45 %. Similarly, an eight-entry buffer using either the hybrid or dynamic scheme achieves 4 % higher performance with 44 % lower cost compared to a 16-entry statically partitioned buffer.

The observed performance improvements are primarily a result of load imbalance between the two resource classes used by UGAL routing: In the absence of congestion, most packets are routed minimally, leaving any buffer space assigned to the non-minimal resource class unused. Even in the presence of congestion, any packets that are routed non-minimally only use the associated resource class while on their way to the randomly selected intermediate router; thus, as long as any packets are routed minimally, network load remains unevenly distributed across the two resource classes. With a statically partitioned buffer, this leaves a fraction of the buffer space assigned to the less heavily loaded class unused; in contrast, buffer sharing allows available buffer space to be distributed among both classes based on demand; in particular, by allowing the more heavily loaded class to use more than its fair share of buffer space, the incidence of tree saturation [78] is reduced, leading to improved overall



(a) Mesh.



(b) FBfly.

Figure 6.7: Performance impact of using atomic VC allocation.

throughput.

As in the case of the Mesh, we find that increasing the number of VCs for a given buffer size does not significantly improve performance, and in fact cause performance degradation for the static and hybrid configurations in most cases as the maximum capacity for any given VC is reduced. While the fully dynamic buffer management scheme consistently yields higher saturation throughput with more VCs, the benefits are generally not commensurate with the associated increase in buffer cost.

6.6.3 Atomic VC Allocation

Figure 6.7 illustrates the impact of using atomic VC allocation on the achievable saturation throughput for different synthetic traffic patterns. We present detailed results for the fully dynamic buffer management scheme; additional simulation runs confirm that our observations hold for the static and hybrid schemes.

For small numbers of VCs, atomic VC allocation severely limits the number of in-flight packets for each router input and output port. This leads to reduced channel utilization and increases the incidence of tree saturation, resulting in substantially degraded overall throughput.

With a maximum packet length of six flits, the smallest configuration for each topology effectively prevents the buffer from being fully utilized, and consequently experiences the most severe performance degradation. While the penalty is reduced as the number of VCs increases, performance remains degraded up to the point where the number of available VCs equals the buffer size divided by the minimum packet length, allowing the buffer to be fully utilized with arbitrary combinations of packets.

In the presence of a sufficiently large number of VCs, atomic VC allocation can reduce HoL blocking by avoiding false dependencies between packets [66]; however, as we explained in Section 6.2.1, the resulting benefits are limited if most packets follow the same direction of travel, as is commonly the case in Mesh network using DOR. Indeed, our results indicate that, even with the maximum possible number of VCs, atomic VC allocation improves performance by at most 2 % for the Mesh and 3 % for the FBfly.

Overall, while atomic VC allocation can be beneficial in situations where a large number of VCs is required as a result of external design constraints, its performance benefits generally do not justify any significant increases in design complexity.

6.7 Related Work

Rezazad and Sarbazi-azad [80] study of the effect of input buffer organization on the performance of Mesh, Torus and Hypercube interconnection networks. In particular,

they evaluate the trade-off between the number and depth of VCs for a fixed buffer size and investigate schemes which assign different numbers of VCs to links in different dimensions. Similarly, Huang et al. [42] propose an approach for optimizing the number of VCs multiplexed onto each physical channel based on traffic characteristics. To this end, they propose an off-line algorithm for computing expected port contention rates for a given target application running on a Mesh network.

Dynamic input buffer management in off-chip interconnection networks was pioneered by Tamir and Frazier in [90]; the authors provide a detailed evaluation of a linked-list based implementation. Pinkston and Choi [16] extend this work by introducing support for VCs. Park et al. [75] and Ni et al. [65] propose alternative implementations that avoid the complexity of linked lists by storing flits belonging to the same queue in contiguous buffer slots inside a linear and circular buffer structure, respectively. Liu and Delgado-Frias [52] adapt the work in [65] to the NoC domain and propose reserving buffer space for each VC in order to avoid deadlock and starvation effects. Lai et al. [49] and Wang et al. [97] propose NoC routers with linked-list based buffer management. In contrast, Nicopoulos et al. [66] and Kumar et al. [47] implement buffer sharing using table-based approaches. Evaluations in these studies focus on Mesh and Torus networks, and generally assume that all packets have the same length.

Other recently proposed methods of reducing buffer cost, including bufferless [25, 60] and Elastic Buffer (EB) [55, 56] flow control, do not support VCs; as such, they must rely on other means—e.g. using multiple parallel networks—to avoid routing deadlock or to support traffic classes. For situations where multiple such networks are required, these approaches are typically less attractive compared to VC flow control.

6.8 Summary

In this chapter, we have investigated the trade-offs involved in choosing input buffer organizations for NoC routers. We have discussed the major design parameters—number and depth of VCs, overall buffer size and static vs. dynamic buffer management schemes—and investigated their respective implications for network cost and

performance.

Simulation results for exemplary 64-node Mesh and FBfly networks show that allowing buffer entries to be shared among VCs improves network performance for a given buffer size; more importantly, it also allows a given level of performance to be achieved at substantially lower cost compared to statically partitioned buffers. Buffer sharing is particularly beneficial for small buffer sizes, where it can provide comparable performance to statically partitioned buffers with 50 % (Mesh) and 100 % (FBfly) more capacity. This is of particular interest in the NoC domain, where area and power budgets are typically subject to tight constraints.

Contrary to popular wisdom, we find that both increasing the number of VCs and using atomic VC allocation yield only modest performance improvements when considering a range of traffic patterns; in particular, we find that the speedup achieved by increasing the number of VCs beyond the minimum imposed by external constraints—e.g., deadlock avoidance requirements—is generally smaller than the associated increase in buffer management overhead.

Chapter 7

Adaptive Backpressure

7.1 Overview

In Chapter 6, we showed that allowing expensive buffer resources to be shared among multiple Virtual Channels (VCs) represents an effective way of improving buffer utilization under benign load conditions, enabling the network designer to achieve better cost-performance trade-offs. However, as we will see in the course of the present chapter, unrestricted sharing can give rise to undesirable behavior under adversarial load conditions. Specifically, buffer sharing allows VCs which experience heavy downstream congestion to monopolize buffer space at the expense of other VCs. This is particularly undesirable in scenarios where multiple workloads with different performance characteristics and Quality-of-Service (QoS) requirements share access to the network, as it can allow an adversarial workload to significantly degrade the performance of a well-behaved one even when both operate on disjoint sets of VCs. Examples of scenarios that are particularly susceptible to such undesired interference effects include networks with heterogeneous endpoints—e.g. general-purpose cores and specialized accelerators—or Chip Multi-Processors (CMPs) executing multiple virtual machines.

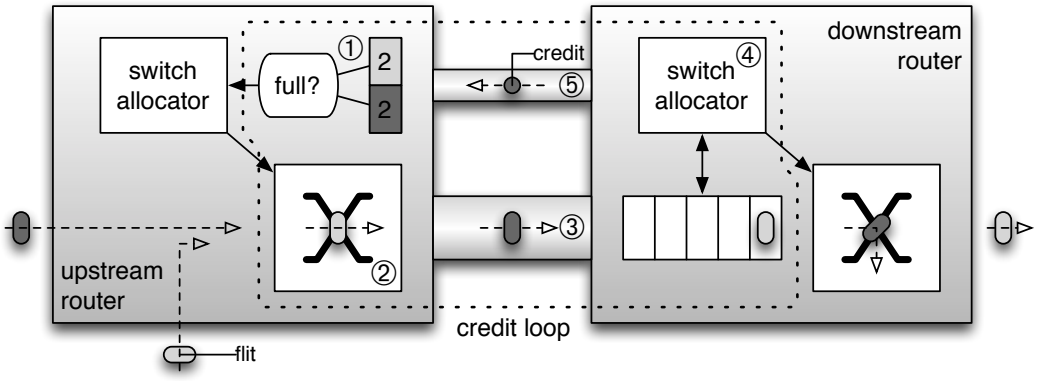
To address these issues, we develop Adaptive Backpressure (ABP), a novel mechanism that heuristically regulates the credit supply for each output VC based on its observed performance characteristics. In doing so, we preferentially assign buffer

space to those VCs that carry well-behaved traffic, and we aim to limit the amount of buffer space that is occupied unproductively by VCs experiencing downstream congestion. ABP is readily implemented as a simple, low-overhead extension to the router's existing output controllers, and it does not require changes to other parts of the router's pipeline or to inter-router signaling.

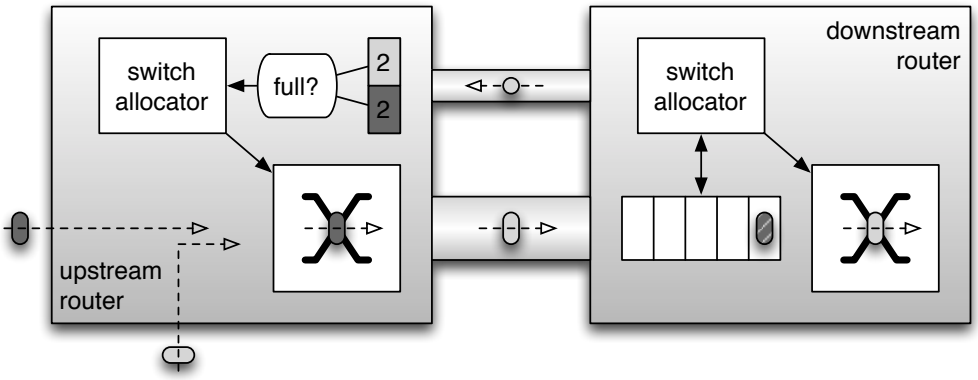
7.2 Motivation

As shown in Chapter 6, sharing buffer space among multiple VCs is attractive in Networks-on-Chip (NoCs) because it improves buffer utilization and thus makes more efficient use of the limited and expensive buffer resources. However, managing buffer entries in a shared pool introduces an additional degree of coupling between VCs, which now have to compete for buffer space in addition to channel bandwidth. As we will demonstrate in the remainder of this section, such coupling can lead to severe performance degradation when buffer space is shared freely among multiple types of traffic with different performance characteristics.

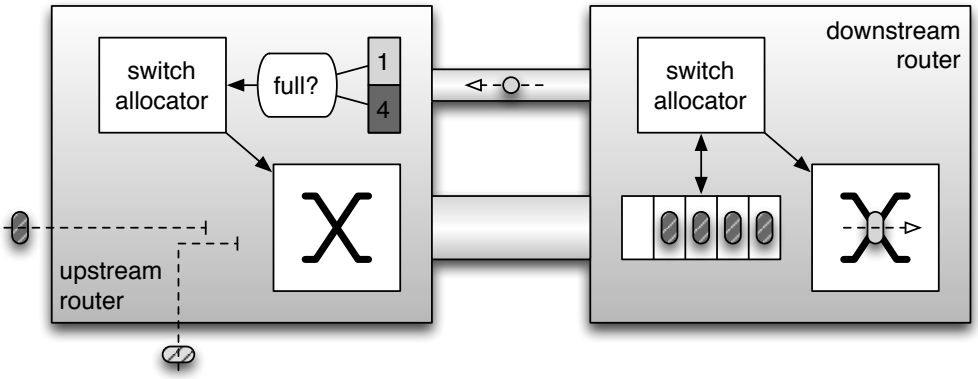
As a motivating example, we consider the situation depicted in Figure 7.1. Figure 7.1a shows a snapshot of the steady state of flits from two different VCs—shaded in light gray and dark gray—arriving at the upstream router from the bottom and left, traversing a network channel, and leaving the downstream router at the right and bottom, respectively. Only the relevant parts of the two routers are shown. The dotted outline marks the credit loop that implements flow control between the two



(a) Uncongested steady-state operation.



(b) The dark gray VC experiences downstream congestion.



(c) Congestion propagates upstream, blocking both VCs.

Figure 7.1: Unrestricted sharing causes congestion to spread across VCs.

neighboring routers:

① A credit is consumed when the switch allocator at the upstream router generates a grant for a waiting flit. The credit is implicitly carried along by the flit as it makes its way towards the next hop.

② In the first cycle after receiving the grant, the flit traverses the upstream router's crossbar.

③ Subsequently, the flit departs the upstream router and traverses the channel towards the downstream router.

④ Upon arrival at the downstream router, the flit is held in the input buffer until it receives a grant from the switch allocator, at which point the implicitly carried credit is sent back upstream.

⑤ The credit traverses the flow control back-channel towards the upstream router, where it indicates to the switch allocator that the corresponding buffer slot is ready to be assigned to a different flit in the next cycle. This closes the credit loop.

In the congestion-free steady state, three implicit credits—located in stages ②, ③ and ④—and one explicit credit—located in stage ⑤—are in flight at any given time. Since each credit effectively corresponds to one particular buffer entry, the input buffer at the downstream router must comprise at least one additional slot—for a total of five slots—in order to ensure that the upstream router always has at least one credit available, allowing it to forward a new flit in every cycle.

We note that out of the four outstanding credits, only one actually corresponds to a downstream buffer slot that is currently occupied by a flit; two credits account for slots that have been reserved but not yet filled, as the corresponding flits are still en route to the downstream router, while the fourth credit corresponds to a buffer entry that has been vacated but remains reserved until the credit returns to the upstream router. This difference between the *perceived* buffer occupancy—measured by the upstream router in the number of outstanding credits—and the *actual* buffer

occupancy at the downstream router is typical for congestion-free operation. All outstanding credits in Figure 7.1a are necessary to support the throughput achieved in this scenario.

In Figure 7.1b, the bottom output at the downstream router becomes congested. As a result, the flit at the head of the downstream router's input buffer becomes blocked—indicated in the figure by a stripe pattern—and remains stationary. Because there are still only four credits in use at that point, the congestion is not yet visible to the upstream router, which therefore continues to forward flits from both VCs as they arrive. The light gray VC's destination output remains uncongested, and its flits therefore continue to be forwarded from the downstream buffer immediately after arrival. As such, any credits the light gray VC consumes continue to be returned upstream after a delay equal to the basic credit round-trip latency, which depends on the router pipeline and the length of the network channels. In contrast, once flits from the dark gray VC reach the downstream router, they are unable to make further progress due to the ongoing congestion. Over time, this results in the flits from the latter VC accumulating in the buffer, preventing their credits from being returned upstream.

Assuming that one buffer slot is reserved for each VC to prevent interleaving deadlock and starvation (cf. Section 6.5), flits from the congested VC eventually fill up all shared buffer space as shown in Figure 7.1c. Exhausting the credit supply, this causes backpressure to reach the upstream router, prompting it to exclude the VC from switch allocation and thus propagating congestion *within the VC* upstream. For the uncongested VC, on the other hand, flits continue to be forwarded immediately after arrival at the downstream router. However, because only its reserved buffer slot remains available, the VC can only send a single flit downstream in each credit round-trip interval; additional flits that arrive at the upstream router during this interval become blocked, as shown in Figure 7.1c. As a result, buffer monopolization allows congestion to spread *across VCs*.

In stark contrast to the scenario depicted in Figure 7.1a, the majority of the outstanding credits in the scenario shown in Figure 7.1c—specifically, all credits held by the congested VC—correspond to flits that occupy slots in the the downstream

buffer; as these flits are blocked, only the credit used by the uncongested VC actually supports data movement. The stationary flits do not contribute to throughput, and therefore represent inefficient use of buffer resources; in fact, by diminishing the upstream router's credit supply, they reduce overall throughput and cause the channel between the two routers to be severely under-utilized.

The given example readily extends to configurations with any number of VCs: If buffer space is shared freely, any VCs that experience downstream congestion will accumulate flits in the downstream router's input buffer over time, reducing the number of credits available to other VCs. Unrestricted buffer sharing therefore facilitates the spread of congestion across VCs, causing the network to become more susceptible to tree saturation [78] and reducing overall performance under load. Furthermore, in scenarios with multiple traffic classes, the described buffer monopolization effect allows misbehaving traffic in one class to significantly degrade the latency and throughput of traffic in other classes even if they are routed on disjoint sets of VCs.

7.3 Detailed Description

In order to mitigate the adverse effects described in Section 7.2 without sacrificing the benefits of dynamic buffer management under benign conditions, we propose a mechanism that regulates sharing by heuristically limiting the number of outstanding credits for each VC based on its observed performance characteristics. VCs that exceed their quota allocation are treated as full until enough credits return for the quota to be satisfied. The goal is to assign quota values in a way that provides individual VCs with enough credits to sustain their observed throughput as in Figure 7.1a while minimizing the amount of buffer space that is occupied unproductively as in Figure 7.1c.

Once a flit occupies a buffer slot, its corresponding credit can only be returned to the upstream router after a grant is received from the switch allocator, indicating that the flit will be forwarded in the next cycle; i.e., the only way to recover outstanding credits is to wait for forward progress at the downstream router. Therefore, a VC can temporarily exceed its quota if an update causes it to drop below the current

number of outstanding credits. Since quota checks are performed in addition to the conventional mechanisms for ensuring credit availability and deadlock avoidance (cf. Section 6.5), this does not interfere with the correct operation of the router; instead, it simply causes the *actual* distribution of credits among VCs to deviate from the *desired* distribution embodied in the quota values. In the absence of deadlock, any flit stored in an input buffer will eventually be forwarded, allowing its corresponding credit to be returned to the upstream router. As this newly returned credit can only be consumed by those VCs that have not exhausted their quota, such deviations from the desired credit distribution tend to self-correct over time.

7.3.1 Quota Computation

In determining quota values for individual VCs, we aim to make credits freely available to those VCs that utilize them efficiently, while being more restrictive in situations that are susceptible to the previously described performance pathologies. To this end, we take advantage of the realization that the number of credits that a given output VC can utilize productively is effectively limited by the throughput it achieves:

Based on our earlier example in Figure 7.1a, we know that if a VC achieves a steady-state throughput of one flit per cycle, the number of outstanding credits required to support this throughput is equal to the basic credit round-trip latency determined by router pipeline and channel length. In the absence of congestion, any additional credits available to the VC beyond the required number will simply remain unused.

If congestion causes stalls at the downstream router, as shown in Figure 7.2, both the number of outstanding credits at the upstream router and the number of occupied buffer slots at the downstream router increase in response to each stall if no quota is imposed. This represents inefficient use of buffer space in the same way as in Figure 7.1c, as the additional flits accumulating in the downstream router's input buffer are not needed to support the resulting effective throughput, and the corresponding credits are unavailable to other VCs.

On the other hand, limiting the number of outstanding credits to less than the

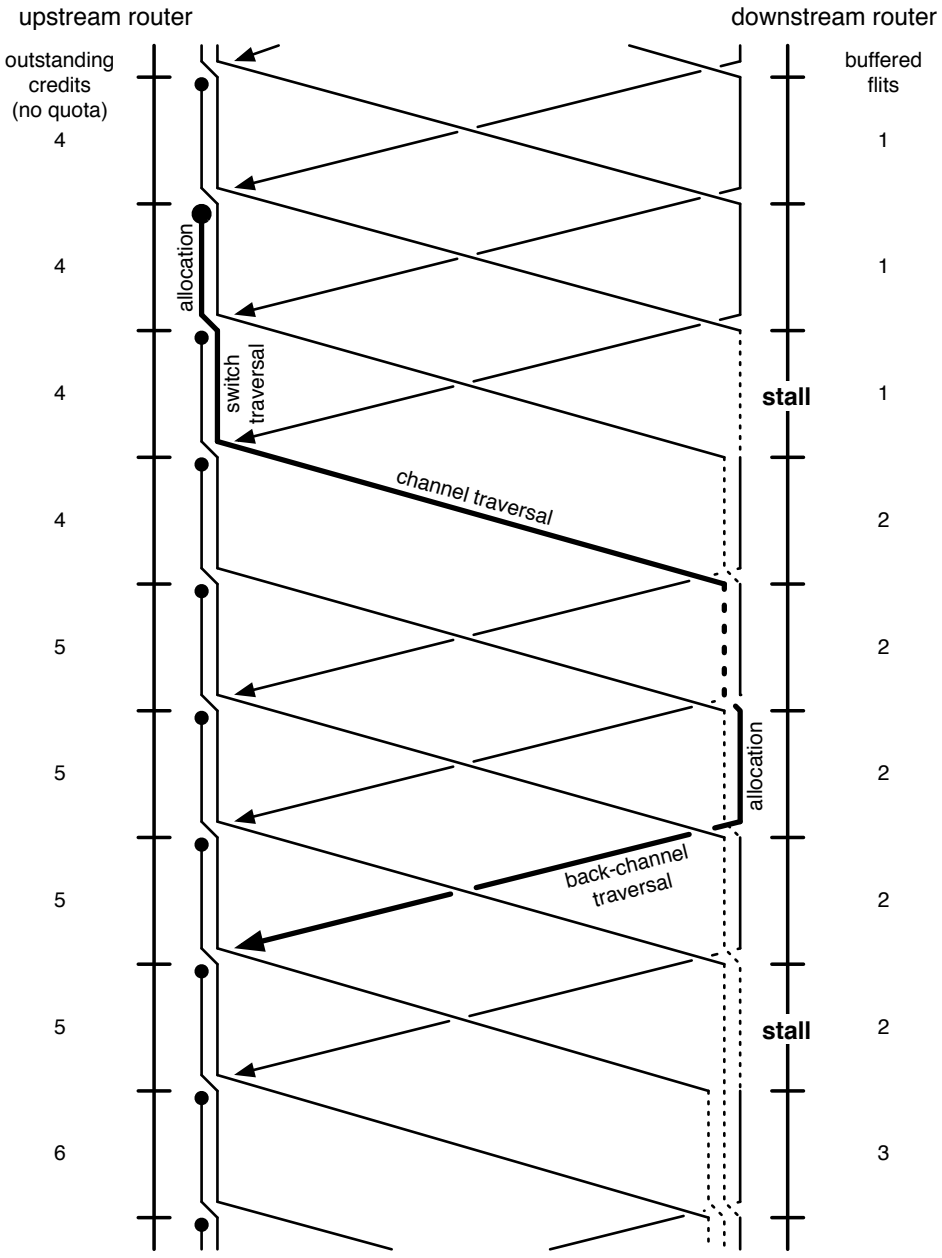


Figure 7.2: With no credit quota, congestion causes flits to accumulate downstream.

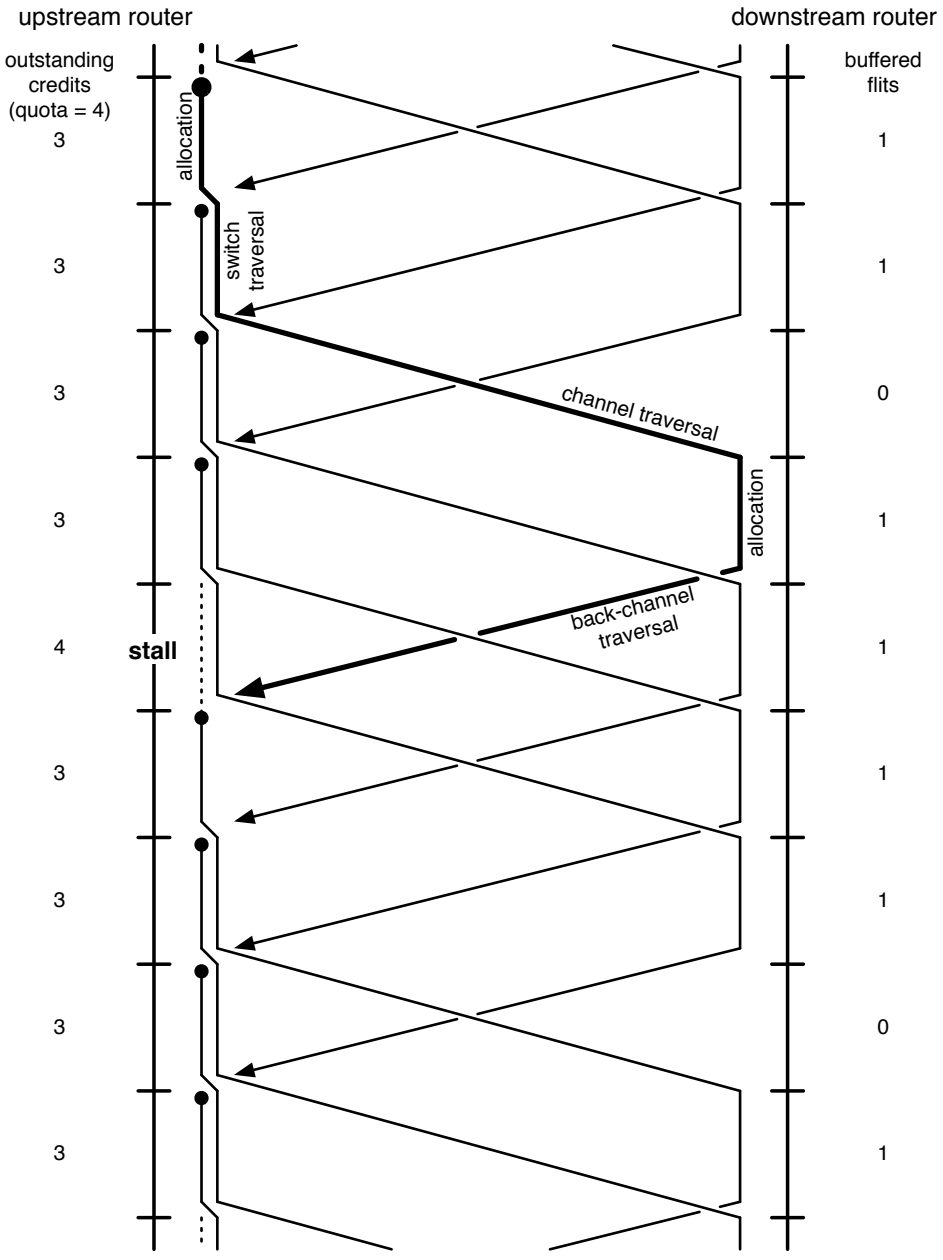


Figure 7.3: Limiting credits reduces the effective downstream throughput.

amount required to cover the basic credit round-trip latency leads to idle cycles in which the downstream router's input buffer is empty, as shown in Figure 7.3: In the given example, once four flits are in flight—and hence, all four allowed credits are outstanding—, the upstream router must suspend transmission until one of the outstanding credits returns. Thus, by imposing a quota on the number of outstanding credits, we can effectively regulate throughput.

We can exploit this ability to regulate throughput by matching the credit quota value to the level of downstream congestion: Figure 7.4 shows the result of applying the downstream stall pattern from Figure 7.2, which causes throughput to be reduced by 20 % at the downstream router, to Figure 7.3, where the credit quota leads to a 20 % reduction in throughput at the upstream router. The resulting effective throughput is the same as in Figure 7.2; however, in contrast to the latter, there is no unproductive accumulation of flits in the downstream router's input buffer. As a result, the congested VC uses fewer credits, leaving more available to other VCs.

In the general case, we can avoid inefficient use of credits and buffer resources in this way by setting a VC's quota value to the product of its effective throughput and the basic credit round-trip latency $T_{crt,base}$. However, in practice, the upstream router cannot easily measure throughput directly. Instead, we compute quota values based on the observed round-trip time $T_{crt,obs}$ for individual credits:

If $T_{crt,obs}$ for a given credit is equal to $T_{crt,base}$, we know that the corresponding flit must have been forwarded immediately at the downstream router, suggesting that the VC it was assigned to is able to achieve unimpeded throughput; consequently, we set its quota value to $T_{crt,base}$ credits.

On the other hand, $T_{crt,obs}$ for a credit may exceed $T_{crt,base}$ if the corresponding flit experienced one or more stall cycles at the downstream router directly, or if it incurred queueing delay as an indirect result of previous stalls as shown by the highlighted transition in Figure 7.2. Each cycle by which $T_{crt,obs}$ exceeds $T_{crt,base}$ requires a subsequent idle cycle in order to avoid unproductive increases in buffer occupancy. By subtracting the difference from the quota value for the congestion-free case, we can ensure that the appropriate number of idle cycles will be generated over the course of the next credit round-trip interval.

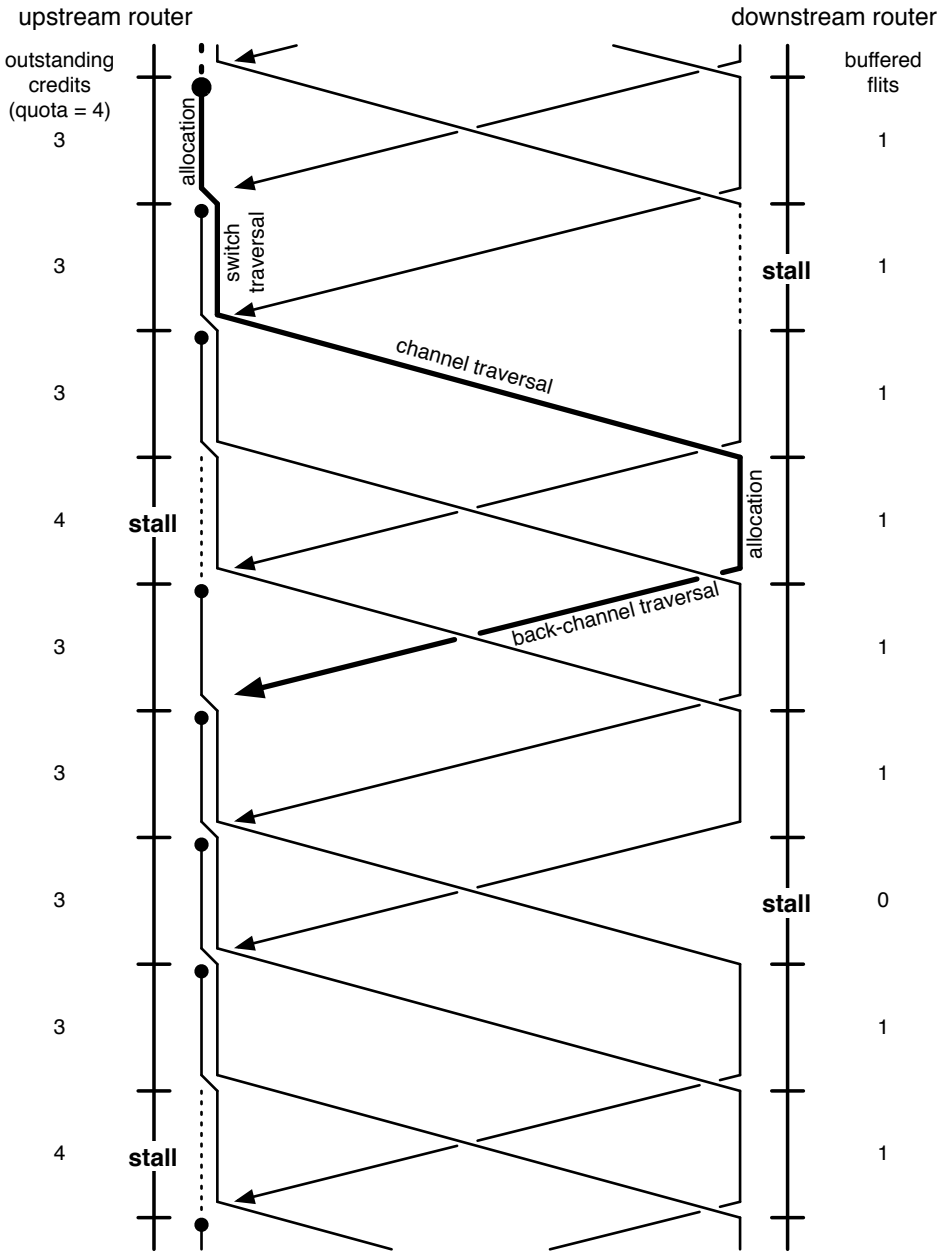


Figure 7.4: Matching quotas to congestion levels prevents accumulation of flits.

Noting that we must allow each VC to use at least one credit in order to guarantee that quota values can continue to be updated, we derive the overall equation for quota updates based on the observed credit round-trip time[1]:

$$\begin{aligned} Q &= max(T_{crt,base} - (T_{crt,obs} - T_{crt,base}), 1) \\ &= max(2 \times T_{crt,base} - T_{crt,obs}, 1) \end{aligned} \tag{7.1}$$

Quotas for all VCs are updated independently, and no explicit effort is made to ensure that the sum of all quotas does not exceed the total capacity of the input buffer. However, as explained at the beginning of Section 7.3, quota values merely represent a *desired* distribution of buffer space among VCs and thus are not required to be conservative in order to ensure correct operation.

Because quota values are adjusted in response to credits that return upstream, ABP can only effectively regulate the credit supply if the time scale at which the level of congestion changes exceeds the credit round-trip delay; in particular, it cannot prevent sporadic periods of congestion that extend for less than a round-trip interval—such as highly transient hot spots—from causing inefficient use of buffer space. The severity of this weakness is mitigated by the fact that credit round-trip times in NoCs typically do not extend beyond a few clock cycles; as a result, the total amount of congestion that can form in the vicinity of such hot spots—and hence, the overall impact on performance—is limited in practice. While it is theoretically possible for an adversarial workload to specifically generate short-term hot spots at arbitrary locations in the network, achieving the desired transient behavior requires tight coordination among multiple traffic sources and is further complicated by interactions with other workloads.

7.3.2 Implementation

From an implementation perspective, ABP comprises two main functional aspects: On the one hand, it needs a mechanism that prevents flits from being forwarded to

[1] In practice, it is beneficial to stabilize quotas by performing updates by means of a moving average rather than using the computed value directly.

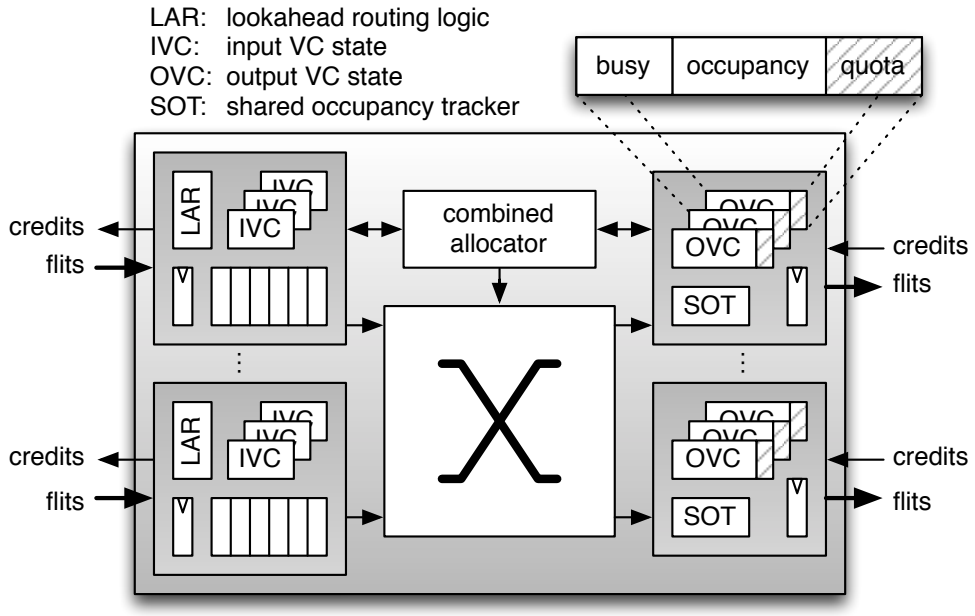


Figure 7.5: Router block diagram with modifications for quota enforcement.

an output VC that has exhausted its quota. On the other hand, it requires a facility that measures credit round-trip times and computes the individual VCs' quota values in response.

In order to enforce quotas in a router design that employs dynamic buffer management (cf. Section 6.4.1), we augment the state of each output VC by adding a register that tracks its current credit quota, as highlighted in Figure 7.5. In combination with the existing VC occupancy counter, this register is used by the allocator to mask output VCs that have exceeded their quota in the same way as if they had run out of buffer space.

In order to minimize implementation overhead, we only measure round-trip delay for a single outstanding credit per output VC at a time. This avoids the need to be able to track a large number of outstanding credits—up to the total number of entries in the input buffer—in parallel; doing so would necessitate the addition of a dynamically managed time stamp buffer at each output port, and the associated overhead would offset a substantial part of the savings achieved by using a dynamically

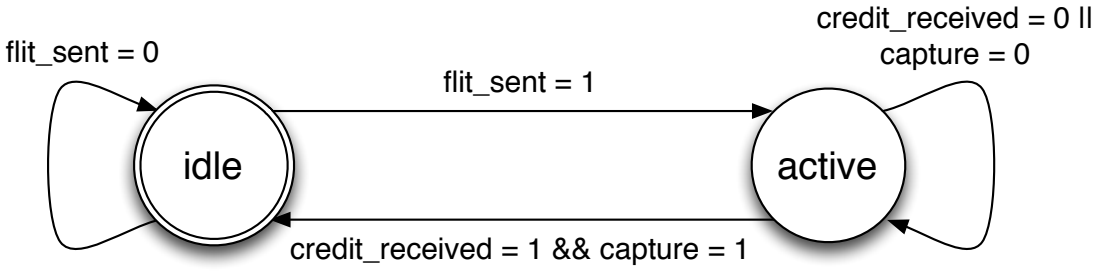


Figure 7.6: State transition diagram for quota computation logic.

managed input buffer in the first place. While restricting measurements in this way can slightly increase the response time of quota updates to the onset of downstream congestion, this does not significantly affect the steady-state performance of ABP.

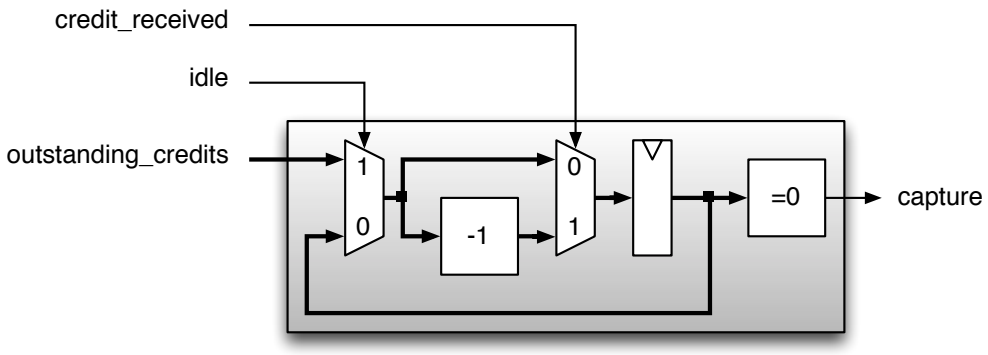
For each output VC, a state variable indicates whether a measurement is currently taking place. As shown in Figure 7.6, all VCs are initially in the *idle* state. When a flit is sent to a VC in this state, it transitions to the *active* state, indicating that a measurement is in progress. It remains in this state until the associated credit returns from the downstream router, at which point the VC updates its quota value according to the observed round-trip delay and transitions back to the *idle* state.

When a VC enters the *active* state, it may already have one or more credits in flight; these must be ignored for the purposes of round-trip time computation. We achieve this by employing a *skip counter* as shown in Figure 7.7a: When a flit is sent to a VC in *idle* state, the skip counter is initialized to the current number of outstanding credits for this VC; subsequently, every time a credit is received, the counter value is decremented. A counter value of zero indicates that the next credit to be received is the one being measured and causes the *capture* signal to be asserted.

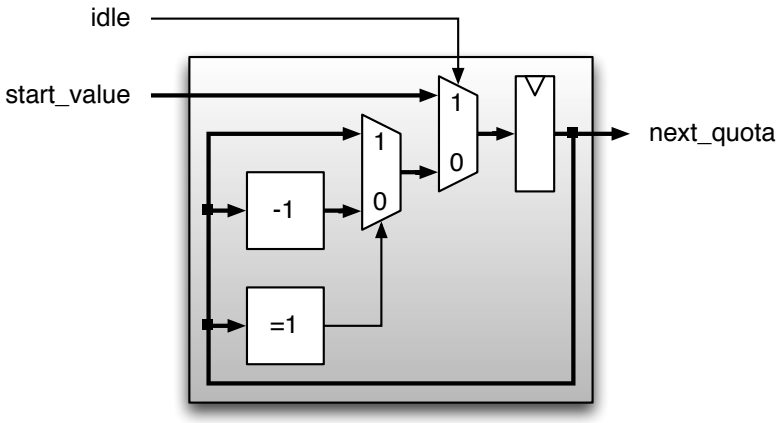
Figure 7.7b sketches the implementation of the counter that is responsible for measuring the actual round-trip time. In *idle* state, the counter is forced to a pre-computed constant:

$$start\_value = 2 \times T_{crt,base} - 1 \tag{7.2}$$

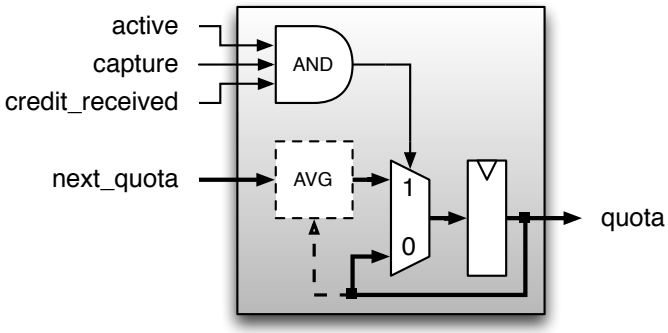
Subtracting one is necessary to properly account for the one-cycle delay between



(a) Skip counter.



(b) Round-trip time counter.



(c) Quota update logic.

Figure 7.7: Implementation sketches for quota computation logic components.

Table 7.1: Storage overhead for ABP.

Description	Cost
Quota registers	$V \times \lceil log_2(T_{crt,base}) \rceil = 12\,bits$
Round-trip timers	$V \times \lceil log_2(2 \times T_{crt,base}) \rceil = 16\,bits$
Skip counters	$V \times \lceil log_2(B) \rceil = 16\,bits$
Total storage overhead per port	$44\,bits$

the *start_value* input and the *next_quota* output. In *active* state, the counter is decremented in every cycle, down to a minimum value of one, as per Equation 7.1.

Finally, Figure 7.7c shows the quota register and the associated update logic: When the VC is in *active* state and the skip counter indicates that no older credits must be discarded, the next credit that is received causes the *next_quota* value computed by the round-trip time counter to be written to the quota register. We can optionally stabilize quota values by performing an additional averaging step, as indicated by the dashed lines in Figure 7.7c. This is achieved by adding *next_quota* to *quota* and discarding the least significant bit of the sum, causing the *quota* output to represent an exponentially weighted moving average of the computed *next_quota* values.

Since measurements and quota updates are computationally simple and can be performed off the critical path, the described mechanism does not adversely affect the router's cycle time.

7.3.3 Overhead

When using a dynamically managed input buffer as described in Section 6.4.1 with a capacity of $B = 16$ flits per input buffer, $V = 4$ VCs and a basic credit round-trip latency of $T_{crt,base} = 8$ cycles, we can compute the total number of registers required for implementing ABP using Table 7.1. Assuming a flit width of 64 bits, the resulting 44 registers represent an overhead of 4.2 % relative to the cost of the flit buffer and its associated management logic (cf. Section 6.4.2).

7.4 Evaluation

7.4.1 Experimental Setup

We evaluate the efficacy of ABP using a customized version of the BookSim 2.0 interconnection network simulator [20]. We conduct simulations on a tiled CMP with 8×8 nodes, connected either as a Mesh, a Concentrated Mesh (CMesh) [4] or a Flattened Butterfly (FBfly) [45]. All network channels are 64 bits wide. Ingress and egress channels have a delay of one cycle, while channels connecting different routers have a delay of one cycle for the Mesh, two cycles for the CMesh and two, four or six cycles—depending on physical distance—for the FBfly. Packets are routed using Dimension-Order Routing (DOR) on the Mesh and CMesh, and using the Universal Globally Adaptive Load-Balanced (UGAL) [87] routing algorithm on the FBfly.

We model input-queued routers with credit-based flow control and two pipeline stages. The first stage performs combined VC and switch allocation as described in Section 5.4 and computes lookahead routing information for the next hop [28], while the second pipeline stage is reserved for switch traversal. We use a separable input-first allocator design with round-robin arbiters. After arriving at a router, credits incur a processing and signal propagation delay of two cycles before the corresponding downstream buffer slot becomes available for allocation again.

Each input buffer has a total capacity of 16 flits, shared among 4 VCs. For experiments with two traffic classes, half of the VCs are statically assigned to each class. One buffer slot is statically reserved for each VC in order to avoid interleaving deadlock and starvation [52]. The *base* configuration does not otherwise restrict sharing. The *abp* and *abp-ma* configurations implement ABP as described in Section 7.3 with immediate and with moving average based quota updates, respectively.

Network terminals maintain a separate, unbounded injection queue per traffic class. Each terminal can can inject a single flit into the network in any given cycle; if multiple traffic classes have pending flits, injection alternates between them in a round-robin fashion. All reported latencies include source queueing delay.

For synthetic traffic, packet arrival times are generated according to a Bernoulli process. Destination addresses are chosen according to a set of traffic patterns

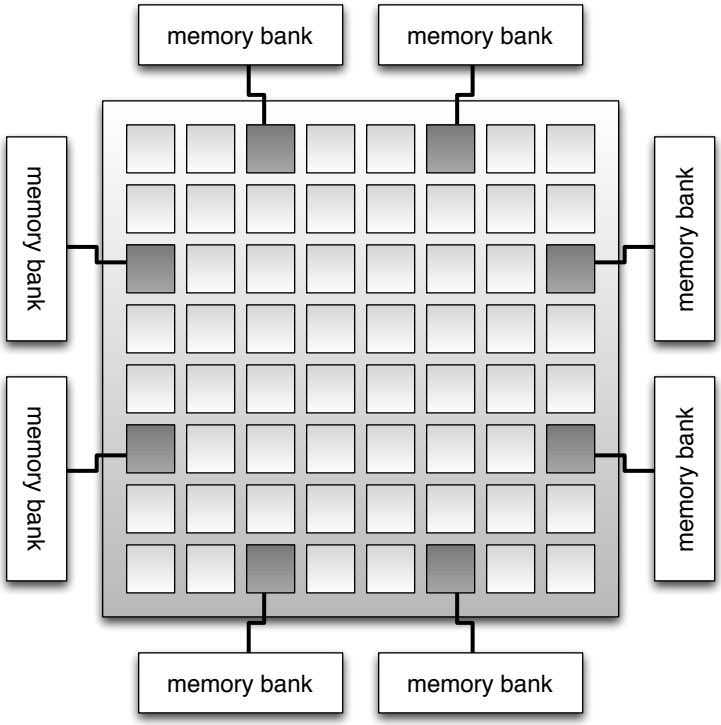


Figure 7.8: Target system for CMP application workloads.

that model communication behavior commonly encountered in network workloads. Throughput numbers correspond to the *effective* throughput *for a particular traffic pattern*; i.e., we report the minimum throughput across all source-destination pairs [20]. Packet lengths follow a bimodal distribution, with 50 % of the packets comprising two and six flits, respectively.

For CMP application workloads, we simulate a target system of 64 tiled nodes with eight attached banks of external memory, as shown in Figure 7.8. Figure 7.9 gives an overview of the individual network nodes' internal structure: Each node comprises a general-purpose processor cores with private L1 instruction and data caches, a slice of a shared L2 cache, as well as an array of throughput-optimized stream processors. Eight of the nodes additionally include a memory controller.

The general-purpose processor cores implement a RISC architecture with in-order execution. Cores and caches operate at four times the network's clock frequency.

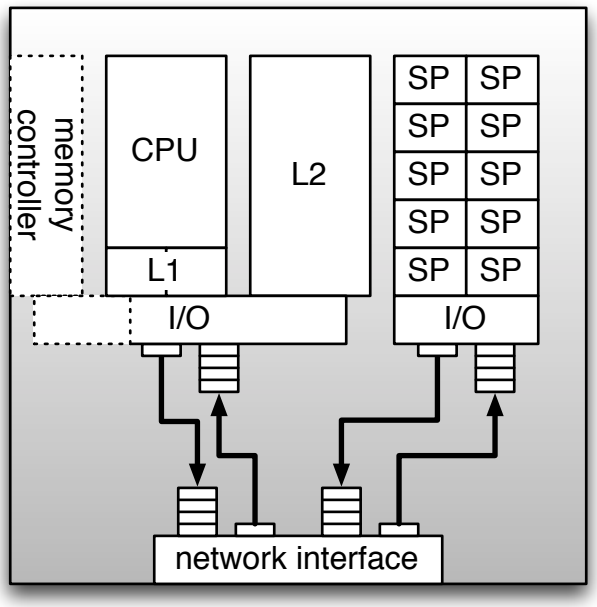


Figure 7.9: Network node configuration.

Table 7.2 lists the key parameters of the cache hierarchy.

With 64-bit wide network channels, memory transactions manifest as traffic with a bimodal packet length distribution: Short packets—e.g., read requests, write responses or invalidation messages—comprise a head flit followed by a flit that carries a 64-bit memory address, while long packets—primarily write requests, writebacks, as well as read and downgrade responses—include eight additional payload flits.

We leverage Netrace [35] to generate application traffic for the general-purpose

Table 7.2: Cache configuration for general-purpose cores.

	L1 cache	L2 cache
Capacity	32 kB + 32 kB	64 × 256 kB
Organization	I + D	shared S-NUCA
Associativity	4-way	8-way
Line size	64 B	64 B
Access time	3 cycles	6 cycles
Coherence	MESI	—

cores based on a set of PARSEC benchmarks [12]. Netrace tracks and enforces timing dependencies between different packets in a trace, enabling us to perform closed-loop simulations without incurring the overhead of using a full-system simulator.

As the co-located array of throughput-optimized cores in each network node supports a large total number of outstanding memory transactions, we can model its aggregate behavior by continuously streaming data to the memory controllers. We assume that data is streamed at cache line granularity, and consequently use the same bimodal packet length distribution as for application traffic. We further assume that the destination addresses for streaming traffic are interleaved such that packets are uniformly spread across all eight memory controllers.

Our evaluation methodology for application traffic is similar to that used by Grot et al. [31,32] to evaluate performance isolation for PARSEC benchmarks; however, in their experiments, streaming traffic is generated by one or more columns of dedicated aggressor nodes added to one side of a Mesh, while in our model, all nodes generate both streaming traffic and PARSEC traffic.

7.4.2 Synthetic Traffic

We first evaluate the efficacy of ABP using synthetically generated traffic. This gives us the freedom to easily exercise the network at a wide variety of load levels without begin restricted to the traffic characteristics of any particular application.

Network Stability

In many practical interconnection networks, once the saturation point is reached, further increases in injection rate will actually *reduce* the effective throughput for the applied traffic pattern. This *throughput instability* is a result of two factors:

On the one hand, higher injection rates cause more flits to be in flight at any given time, leading to increased contention for network resources and facilitating the formation of *tree saturation* [78]. Overall throughput is reduced in response to congestion spreading through the network.

On the other hand, when multiple flows of traffic merge at different points along a

path through the network, the use of *locally fair* allocation policies at the individual routers can lead to an exponential reduction in available network bandwidth for flows that are subject to multiple instances of such merging. For adversarial traffic patterns, this causes one or more sources to become subject to starvation as injection rate increases beyond saturation, reducing the effective throughput for the desired traffic pattern [20].

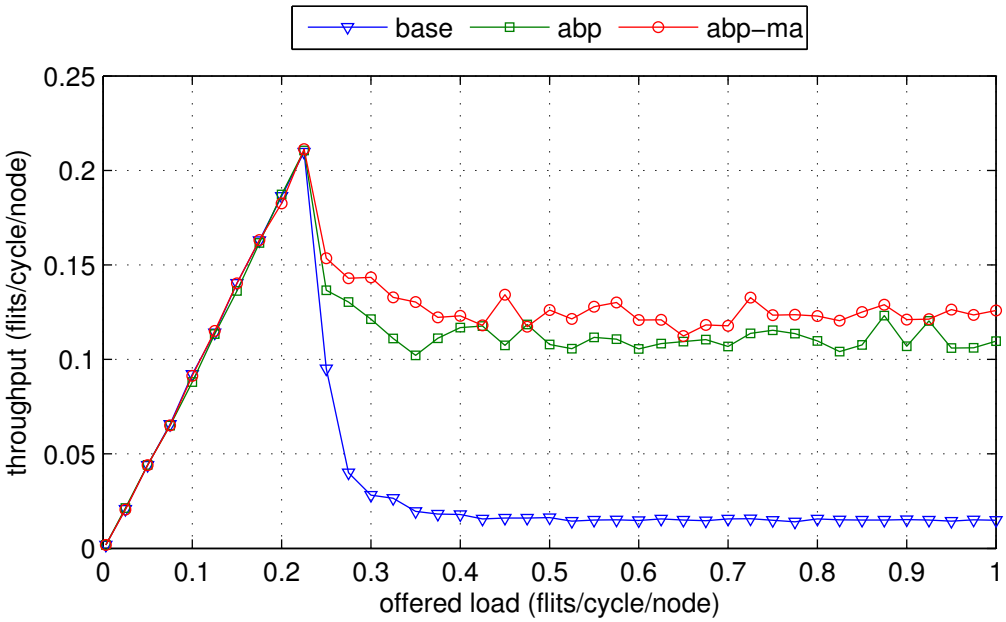
By regulating credit availability and promoting efficient use of buffer resources, ABP can mitigate both causes of network instability: Limiting buffer occupancy for congested traffic flows inhibits the spread of congestion to other flows and therefore inhibits tree saturation. Furthermore, ABP effectively throttles sources that inject traffic at a very high rate, alleviating the starvation effects that cause instability in adversarial traffic patterns.

Figure 7.10 demonstrates throughput instability for the Mesh and CMesh networks with adversarial Tornado (TO) traffic[2]. In both networks, throughput drops sharply once the saturation point is reached and stabilizes at a substantially lower post-saturation value.

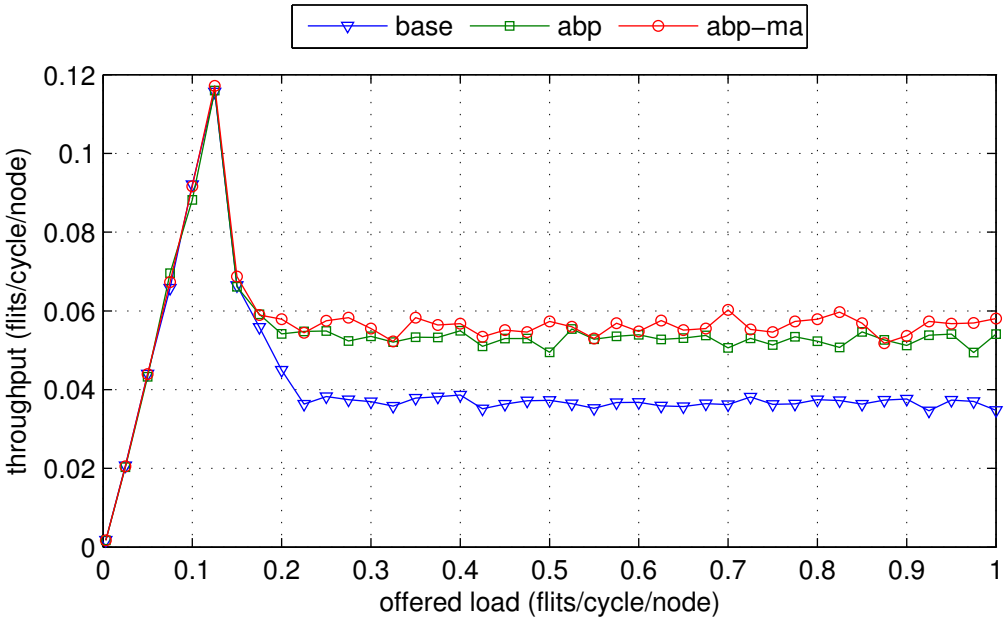
For the Mesh, as shown in Figure 7.10a, extensive merging of traffic flows effected by the topology's large network diameter causes the throughput for the *base* configuration to drop to less than 8 % of the value achieved at saturation. In contrast, post-saturation throughput with ABP stabilizes at 51 % of the saturation throughput. Stabilizing quota values by performing updates via a moving average process further improves the efficacy of ABP, increasing throughput for the *abp-ma* configuration to 60 %; this represents a 7.8-fold improvement over the *base* configuration.

Due to its smaller network diameter, the CMesh is less susceptible to starvation caused by traffic merging than the Mesh; on the other hand, because individual links in the CMesh are shared between a larger number of flows, the effects of tree saturation become more pronounced. This results in an increase in post-saturation throughput for the *base* configuration to 32 % of the peak rate, while post-saturation throughput for that for the *abp* and *abp-base* configurations decreases to 42 % and

[2]The FBfly topology is inherently not susceptible to such instability and is thus not considered here.



(a) Mesh.



(b) Concentrated Mesh.

Figure 7.10: Throughput vs. offered load for TO traffic pattern.

49 % of said value, respectively. Nevertheless, ABP remains beneficial, yielding up to a 1.5-fold improvement over the *base* configuration.

Figure 7.11 shows the throughput at maximum injection rate for different synthetic traffic patterns on the Mesh and CMesh networks, as well as the harmonic mean across traffic patterns. The upper end of the white segment atop each bar marks the saturation rate for a particular combination of configuration and traffic pattern.

With the exception of Uniform Random (UR) traffic, ABP measurably improves post-saturation throughput for all traffic patterns, with an average improvement of 52 % for the CMesh and 259 % for the Mesh. While ABP reduces saturation rate by an average of 2–3 % across traffic patterns, this only results in a performance degradation in a narrow region around the saturation point as throughput at low to medium network load is unaffected.

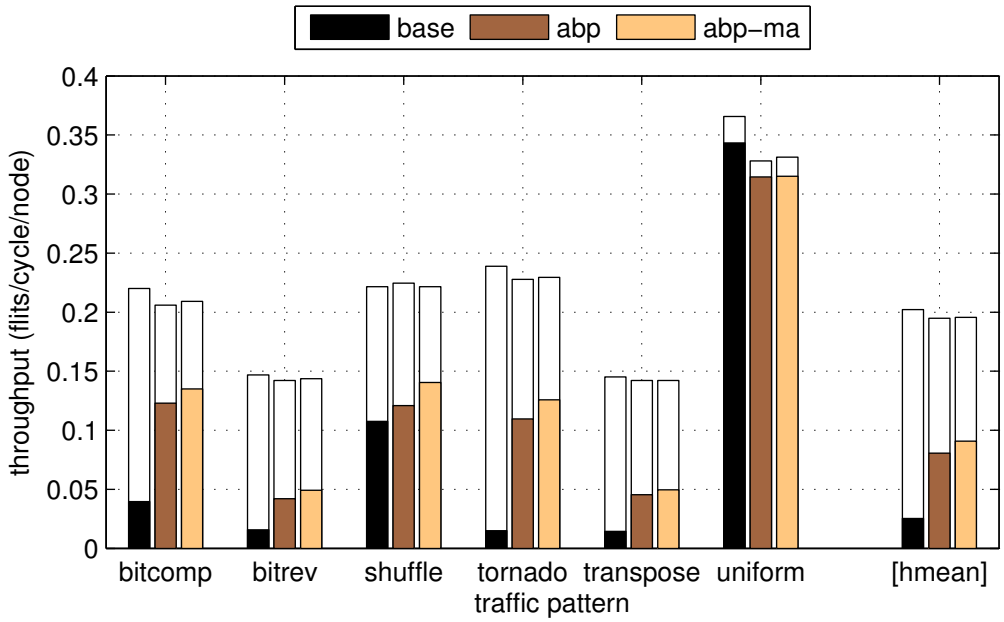
For UR random traffic, ABP actually decreases saturation rate by 10 % and 3 % and post-saturation throughput by 8 % and 5 % for the Mesh and CMesh, respectively. This is because UR traffic reduces correlation between successive packets in a given VC as a result of randomly selected destinations, causing quota values to be less likely to apply across multiple packets[3]. The performance degradation for UR traffic only manifests at high injection rates; in contrast, ABP affords substantial performance improvements for most other traffic patterns starting at significantly lower injection rates.

While practical systems with finite injection queues are inherently self-throttled, and therefore cannot operate in the post-saturation region in steady state, it is possible for the injection rate in such systems to exceed the saturation rate temporarily, e.g. as a result of bursty traffic or the formation of a transient hotspot in the network. In such cases, by maintaining high post-saturation throughput, ABP enables the network to recover and return to steady-state operation more quickly.

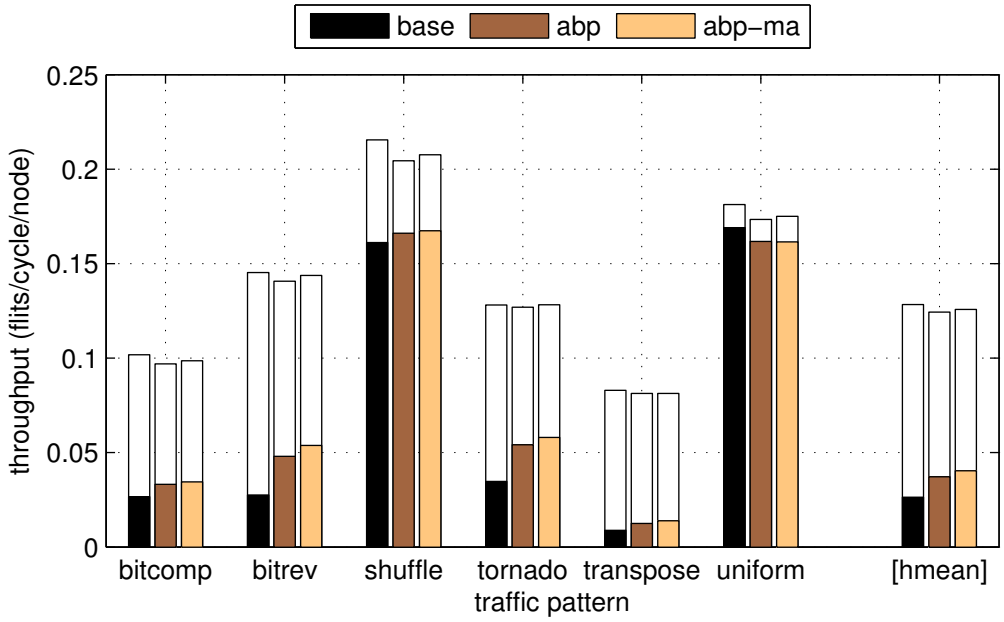
Performance Isolation

Individual processor cores in a typical CMP can generally only support a limited number of outstanding network requests—e.g. L1 cache misses or evictions—at a time;

[3]Concentration mitigates this effect by reducing the number of possible destination routers.



(a) Mesh.



(b) CMesh.

Figure 7.11: Throughput at maximum injection rate (outlines show saturation rate).

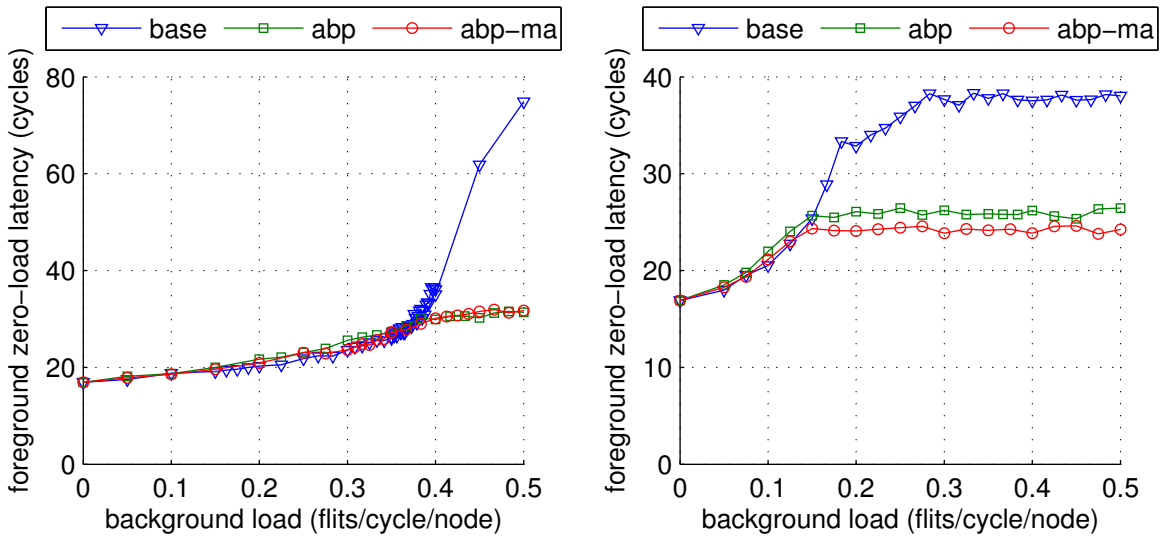
once the maximum number of concurrent in-flight requests is reached, any instruction that would trigger an additional request must block until one or more earlier requests complete. As packet latency in an interconnection network generally increases with offered load, this introduces a negative feedback loop that causes performance in such *self-throttled* systems to be primarily limited by end-to-end latency rather than available bandwidth [81]. In evaluating performance isolation, we therefore first focus our investigation on how latency for a given traffic class of interest is affected by a second class of background traffic.

Figure 7.12 demonstrates how the zero-load latency for a foreground workload consisting of UR traffic varies as increasingly adversarial background traffic—using the Nearest Neighbor (NN), UR, Transpose (TR) and Hot Spot (HS) traffic patterns—is injected into a 64-node CMesh. In all four cases, the latency for the the foreground traffic pattern is initially virtually identical across the three configurations, increasing slowly as background load ramps up.

Once the background traffic reaches its saturation point, buffer occupancy for the *base* configuration increases rapidly, resulting in buffer monopolization as described in Section 7.2 and severely degrading the zero-load latency of the foreground traffic. In contrast, the *abp* configuration effectively limits buffer occupancy for the background workload once it reaches the saturation point, preventing such inefficient use of buffer space. As a result, the latency for the foreground traffic quickly stabilizes at a significantly lower level compared to the baseline implementation.

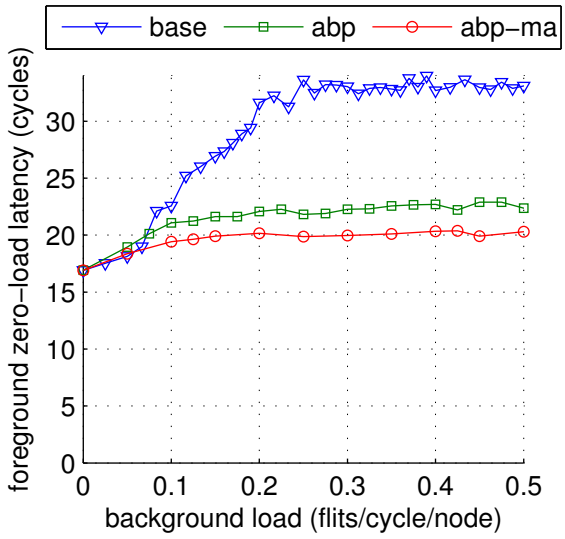
While the inflection point occurs earlier for more adversarial background traffic patterns, the increase in foreground latency is actually more pronounced for benign patterns, both for the baseline configuration and—albeit to a lesser extent—for the two ABP-based ones. Benign traffic patterns saturate at higher injection rates, resulting in a larger number of in-flight flits, which in turn cause the foreground traffic's flits to experience a higher degree of contention for network resources.

Figure 7.13 shows the average zero-load latency in the presence of 50 % UR background traffic across a set of synthetic foreground traffic patterns, including Bit Complement (BC), Bit Reverse (BR), Shuffle (SH), TO, TR and UR, for the three topologies we investigate. The white segment at the bottom of each bar represents

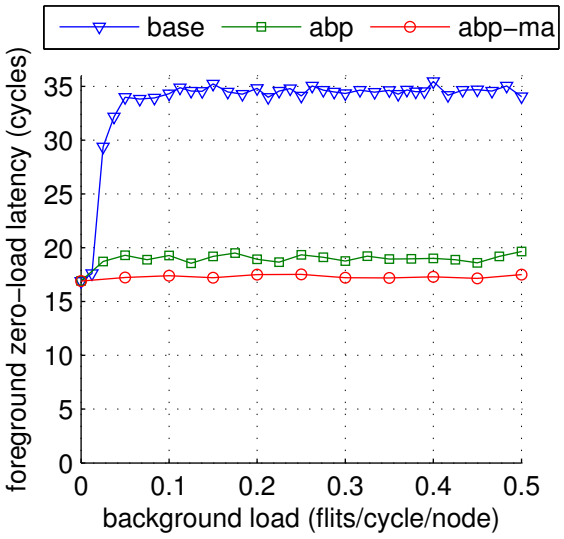


(a) NN background traffic.

(b) UR background traffic.



(c) TR background traffic.



(d) HS background traffic.

Figure 7.12: Foreground zero-load latency for UR foreground traffic on CMesh.

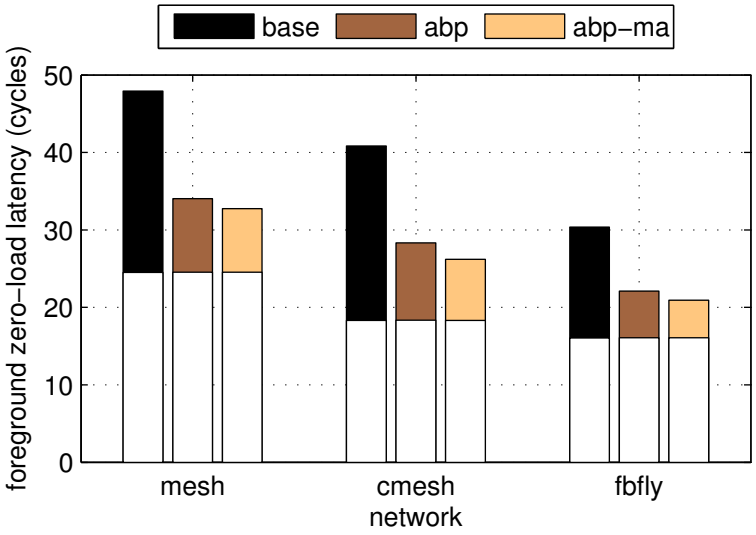
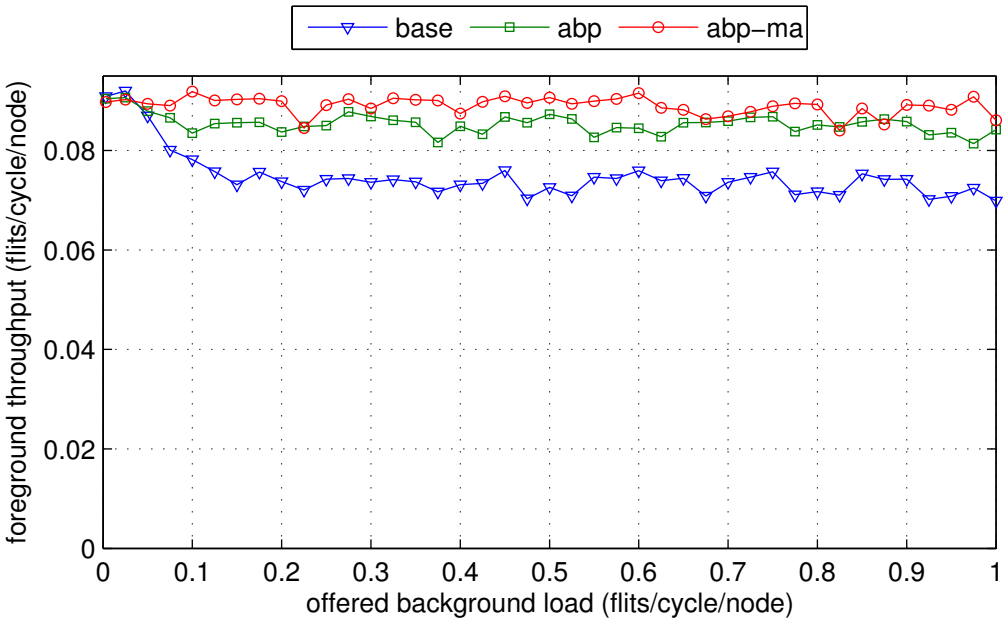


Figure 7.13: Average foreground zero-load latency with 50 % UR background traffic.

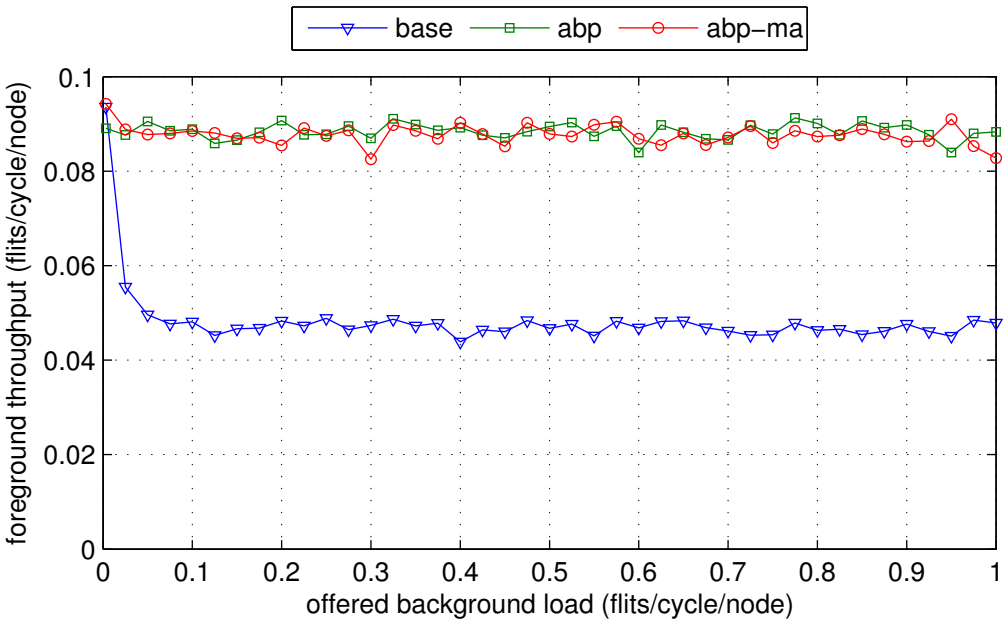
the latency measured in the absence of background traffic; our measurements show that ABP does not adversely affect latency under benign conditions. While the magnitude of foreground latency varies by traffic pattern, we find that the performance difference between the *base*, *abp* and *abp-ma* configurations is consistent across all six patterns for each topology. Overall, the latency improvement is most pronounced for the CMesh network at 36 % and slightly smaller for the Mesh and FBfly at about 31 %.

In addition to increasing zero-load latency, adversarial background traffic can also negatively impact the throughput achieved by benign foreground traffic. Figure 7.14 shows the effective throughput when injecting UR foreground traffic into a CMesh at a target injection rate of 10 %. As with latency, increased buffer occupancy causes foreground performance for the *base* configuration to degrade as the background traffic enters its saturation region[4]. In contrast, both ABP-based configurations allow foreground throughput to remain stable beyond the background traffic's saturation

[4]Note that the higher foreground injection rate causes the background traffic to saturate earlier than in Figure 7.12.



(a) TR background traffic.



(b) HS background traffic.

Figure 7.14: Throughput degradation for UR foreground traffic on CMesh.

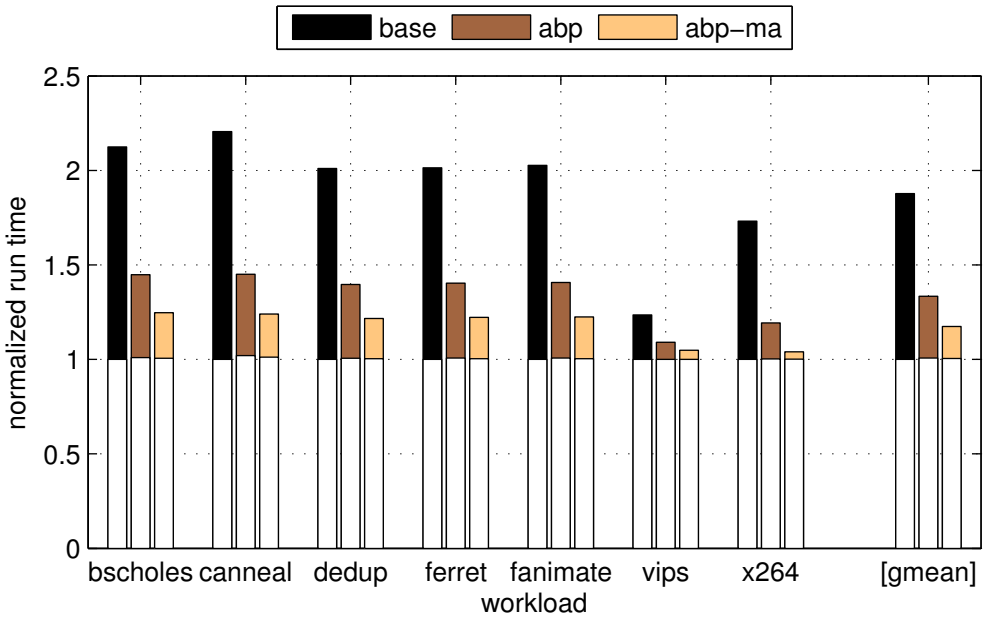
point, resulting in minimal overall performance degradation. ABP effectively throttles traffic with undesirable performance characteristics, mitigating its impact on the network, while leaving well-behaved traffic largely unaffected.

Performance on the Mesh is qualitatively similar if we double the foreground injection rate to 20 % in order to account for the difference in bisection bandwidth. For the FBfly, we observe qualitatively similar behavior for HS background traffic; however, even at an injection rate of 20 %, foreground throughput for UR traffic remains virtually unaffected by TR background traffic.

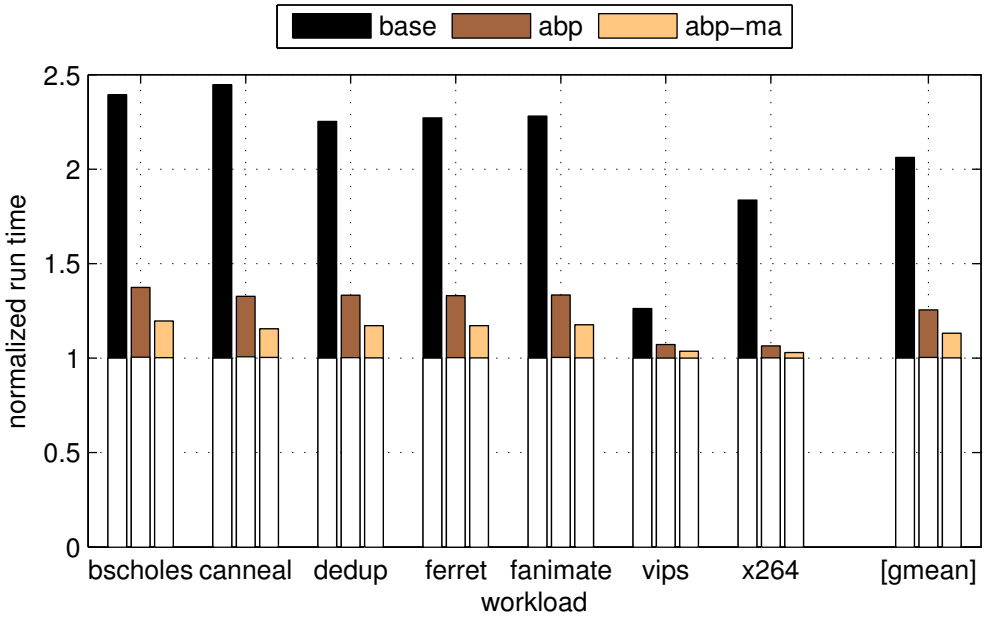
7.4.3 Application Performance

In order to validate the results of our experiments with synthetic traffic, we conduct measurements on a simulated 64-node CMP as described in Section 7.4.1. Specifically, we consider a setup where network nodes are heterogeneous, comprising both a general-purpose processor and an array of throughput-optimized stream processing cores. Individual general-purpose cores represent self-throttling traffic sources as per our earlier description and as such are primarily sensitive to network latency. In contrast, each array of throughput cores in aggregate supports a large number of outstanding memory transactions; as such, their combined traffic is largely limited by available network bandwidth. The conflicting performance characteristics of the two types of traffic can lead to undesired interference effects when both are transported across the same NoC.

Figure 7.15 shows the normalized execution time for individual PARSEC benchmarks running on the general-purpose cores in the presence of streaming background traffic as described in Section 7.4.1, as well as the geometric mean across all benchmarks. For each PARSEC application, we measure the time it takes to deliver the first one million packets from the benchmark's Region of Interest (ROI) across the network. All results are normalized to the execution time measured for the *base* configuration when no background traffic is injected into the network. Results are shown for the CMesh and FBfly topologies; the behavior for the Mesh is qualitatively similar to that for the CMesh, but average slowdown is reduced by about 10 % across



(a) CMesh.



(b) FBfly.

Figure 7.15: Application slowdown for general-purpose cores.

configurations.

The white segment at the bottom of each bar in Figure 7.15 corresponds to the execution time without background traffic for a particular combination of configuration and benchmark. As in our earlier experiments with synthetic traffic, we see that using ABP does not adversely affect the performance in the benign case.

Once streaming background traffic is injected, all three configurations experience significant performance degradation. This slowdown is primarily a result of increased packet latency: As the simple in-order cores modeled in our experiment support only a single outstanding memory transaction, any additional network delay incurred by memory traffic directly results in stall cycles.

The reasons for the latency increase are two-fold: On the one hand, additional contention delay is incurred as a result of failed allocation attempts as both types of traffic compete for channel bandwidth. On the other hand, dynamically managed input buffers can allow the streaming background load to monopolize buffer space as described in Section 7.2, effectively limiting the application traffic's credit supply. In contrast to contention-induced delay, the effects of this *indirect* throttling cannot be mitigated by prioritizing latency-sensitive traffic during allocation.

In limiting buffer occupancy for adversarial traffic, ABP reduces packet latency and improves throughput for the PARSEC traffic generated by the general-purpose cores, resulting in a 34 %, 38 % and 45 % reduction in execution time for the Mesh, CMesh and FBfly, respectively, compared to the baseline configuration. Across topologies, the benefit of employing ABP is least pronounced (15–17 %) for the *vips* benchmark; this is because *vips* uses coarse-grain parallelism with comparatively little sharing and data exchange between different nodes, making this benchmark less sensitive to network performance. In contrast, the *canneal* benchmark generates significantly higher network load and is thus much more sensitive to network performance, resulting in improvements of 39–53 %.

7.5 Related Work

The *ViChaR* scheme introduced by Nicopoulos et al. [66] regulates the number of active VCs at each router input based on network load; however, it does not impose limits on the amount of credits that individual VCs can hold. While the use of atomic VC allocation and a fixed, short packet length prevent *individual* VCs from monopolizing buffer space in this scheme, *groups* of VCs with similar performance characteristics—e.g. those assigned to a particular traffic class—can in aggregate still cause significant performance degradation for other VCs.

Banerjee and Moore [5] show that resource utilization in NoCs can be improved by grouping packets into flows—e.g. based on their destination—and allocating VCs to flows instead of individual packets. Shim et al. [85] propose a similar approach that statically maps flows to specific VCs at design time. Both approaches prevent blocked flows from acquiring more than a single VC at each input buffer, but neither limits the amount of buffer space occupied by that VC. As such, when using dynamic buffer management, they are complementary to our proposed mechanism.

Lai et al. [49] propose a scheme in which each router predicts congestion levels at each neighboring routers' output ports and prioritizes those packets during switch allocation that will be forwarded to uncongested outputs. Similarly to ABP, this causes fewer flits to be sent to downstream VCs which are subject to congestion; however, because congestion levels are estimated at port granularity, this approach cannot prevent interference between multiple flows of packets destined for the same output.

Network-level congestion control schemes based on source throttling [6, 92] inherently mitigate buffer monopolization effects by reducing the incidence of congestion in the network. However, because such schemes only perform coarse-grained traffic regulation at the network boundary, they tend to be pessimistic and slow to react to localized changes in network behavior. Furthermore, they typically only consider the aggregate behavior across all VCs and thus do not address interference between concurrent workloads.

Finally, prior research has explored various schemes for providing QoS guarantees

and isolation between workloads in NoCs [31, 32, 50, 70] These mechanisms generally assume that buffer space is statically partitioned and include no provisions to avoid interference effects caused by buffer sharing; consequently, such approaches are complementary to the ABP scheme developed in this chapter.

7.6 Summary

In this chapter, we have developed ABP, a novel scheme for regulating occupancy in dynamically managed router input buffers. By heuristically limiting each VC's credit supply based on its observed performance characteristics, the proposed scheme aims to minimize unproductive buffer occupancy and to prevent VCs that experience downstream congestion from monopolizing shared buffer space at the expense of other VCs' performance. ABP maintains the utilization and performance benefits that dynamic buffer management provides under benign load conditions, and it is readily implemented as a simple, low-overhead extension to the existing flow control logic.

Simulation results for three exemplary 64-node NoCs show that ABP effectively improves performance isolation, reducing zero-load latency in the presence of background traffic by up to 36 % compared to a state-of-the-art implementation with unrestricted sharing. ABP also improves network stability, increasing throughput under heavy load by an average of 52 % and 259 % across a set of six synthetic traffic patterns for the Mesh and CMesh, respectively. Finally, we present simulation results for PARSEC benchmarks running on a heterogeneous CMP and show that ABP can reduce execution time in the presence of streaming background traffic by an average of 34 %, 38 % and 45 % across benchmarks for the Mesh, CMesh and FBfly networks, respectively.

Overall, ABP enables networks to satisfy more stringent QoS requirements while capitalizing on the performance and cost benefits of dynamic buffer management.

Chapter 8

Conclusion

8.1 Summary

With the end of Dennard scaling, diminishing returns from traditional approaches to increasing single-threaded performance, and the rise of energy efficiency as a primary design concern, continuing increases in processing power will rely on the development of efficient large-scale Chip Multi-Processors (CMPs). Networks-on-Chip (NoCs) have emerged as a promising approach for satisfying the communication requirements of such designs. The latency and throughput characteristics of the network have a direct impact on the CMP's performance; likewise, the cost of communication directly affects its energy efficiency.

While high-level design parameter—topology, routing and flow control—set the framework for the network's overall performance and cost, an efficient network must be composed of efficient channel and router implementations. In the present dissertation, we have investigated implementation aspects and microarchitectural design trade-offs for efficient high-performance NoC routers.

In Chapter 2 and Chapter 3, we have first discussed practical implementation aspects for elementary arbiters and allocators, respectively, and conducted a detailed evaluation of standard-cell designs in a commercial 45nm process. To this end, we have investigated several approaches for building wavefront allocators that are free from combinational loops. Comparing delay, area and energy efficiency, we have

found that matrix arbiters, which have frequently been cited as the preferable design choice in the context of interconnection networks, are simultaneously less efficient and slower than round-robin arbiters at typical sizes encountered in NoC routers. Furthermore, we have shown that our synthesis-friendly wavefront allocator designs yield lower delay and cost than a previously proposed loop-free implementation.

Based on the elementary allocator designs, we have investigated practical Virtual Channel (VC) allocator implementations in Chapter 4. Because the effective load on the VC allocator is low in practice, we have found that differences in matching quality between different implementation variants do not translate into significant differences in network-level performance; as such, the optimal choice of VC allocator is primarily determined by delay and cost considerations. In practice, this favors separable input-first implementations.

We have furthermore introduced sparse VC allocation, a scheme that reduces VC allocator complexity by exploiting restrictions on the possible transitions between VCs assigned to different packet classes. Synthesis results show that sparse VC allocation yields substantial improvements in delay, area and energy efficiency and thus increases the allocator's scalability.

In Chapter 5, we have similarly investigated switch allocator implementations. We have found that a wavefront allocator's superior matching quality only translates to network-level performance improvements in cases where request matrices tend to be densely populated. In particular, waterfront allocation can substantially improve saturation rate for Flattened Butterfly (FBfly) networks with many VCs, but generally yields little benefit for Mesh networks with Dimension-Order Routing (DOR). Even in cases where performance is improved, wavefront allocation is primarily attractive if its comparatively high delay can be masked by external timing constraints.

While speculative switch allocation yields a substantial reduction in latency at low to medium network load, we have found that it provides only marginal gains in saturation throughput. Based on this realization, we have developed two modified speculation mechanisms—pessimistic speculation and priority-based speculation—that improve delay, area and energy efficiency compared to the canonical implementation at the cost of further reducing the performance gains under heavy load.

By obviating the need for a dedicated VC allocator, combined VC and switch allocation achieves significant cost reductions while providing the same latency benefits as speculative switch allocation under low to medium load. However, we have found that this is achieved at the cost of a slight degradation in saturation throughput. Despite this, its low cost and delay make combined allocation an attractive design choice for many network configurations.

We have investigated buffer organization trade-offs and evaluated buffer management schemes in Chapter 6. Simulation results show that dynamic buffer management improves saturation throughput for a given buffer size or, conversely, allows a desired saturation throughput to be achieved with a smaller buffer. We have found that buffer sharing is particularly attractive for FBfly networks with Universal Globally Adaptive Load-Balanced (UGAL) routing, as it reduces tree saturation by allowing buffer space to be distributed among resource classes based on demand. Furthermore, we have found that the associated overhead generally makes increasing the number of VCs for a fixed buffer size beyond the minimum necessary to support the desired set of traffic classes unattractive, and that atomic VC allocation yields only modest performance improvements even with large numbers of VCs.

While allowing buffer space to be shared among VCs improves performance for well-behaved traffic, we have demonstrated in Chapter 7 that unrestricted sharing can lead to pathological performance and undesired interference between traffic classes in the presence of downstream congestion. In order to avoid such adverse effects without degrading performance in the benign case, we have developed Adaptive Backpressure (ABP), a low-overhead scheme that continuously regulates each VC's credit supply based on its observed performance characteristics. This effectively counteracts unproductive buffer occupancy and thus prevents VCs that are subject to downstream congestion from monopolizing buffer space and degrading the performance of other VCs. As such, ABP allows network designers to take advantage of the cost and performance benefits of unrestricted buffer sharing under benign load without incurring the associated loss of isolation between traffic classes.

8.2 Future Work

The work described in this dissertation presents numerous opportunities for additional research:

In Chapter 5, we found that the effect of matching quality on network-level performance is limited by sparsely populated request matrices. In particular, differences between allocator implementations only manifest in the presence of substantial congestion. As such, it may prove beneficial to utilize different allocator implementations at low network load, where latency is most critical, than at high load, where throughput is more important. In particular, we can imagine using a latency-optimized allocator—perhaps simple collision detection—for newly arriving flits, and a more complex allocator—possibly requiring multiple cycles to generate a matching—for buffered flits.

We furthermore found that the use of DOR in Mesh networks leads to an uneven distribution of requests across output ports—effectively further reducing the density of the request matrix—because packets only turn once per dimension; it would be interesting to investigate how our results change when using fully adaptive routing [24].

While Chapter 6 showed that dynamic buffer management schemes lead to better cost-performance trade-offs than static schemes, the performance benefits are partially offset by the associated pointer overhead. We can reduce such overhead by sharing buffer space at a coarser granularity; e.g., in a linked-list based implementation, we can reduce the number of storage elements required for holding buffer pointers and free pointers—and thus a substantial fraction of the overall overhead—by more than 50 % by distributing buffer slots among VCs in pairs[1]. To determine whether this is a good trade-off, additional experiments will have to quantify the associated loss in performance.

As we saw in Chapter 7, the efficacy of ABP can be improved by updating quota values based on a moving average of credit round-trip times. Other filter functions and alternative quota heuristics—perhaps based on expected traffic characteristics supplied by the application, runtime or operating system—would be worth exploring.

[1] Note that this increase in granularity only affects the amount of buffer space that is assigned to a VC; in particular, flow control is still performed at the granularity of individual buffer entries.

ABP was shown to substantially reduce the throughput degradation experienced with adversarial traffic patterns when a Mesh network is operated beyond the saturation point; to completely eliminate such effects, efficient mechanisms for enforcing *global* fairness—ideally without incurring the overhead associated with age-based allocation—will have to be developed.

In general, much of our evaluation of ABP focused on steady-state performance metrics. Further research will be necessary to evaluate the dynamic behavior of ABP and its performance in the presence of rapidly changing traffic conditions in more detail. In particular, it would be interesting to explore whether an adversarial workload can render ABP ineffective by injecting traffic with specifically tuned temporal characteristics.

Due to limitations of our simulation infrastructure, the streaming background load in our application traffic experiments was synthetically generated; it would be insightful to repeat these simulations and replace the synthetic background workload with actual stream processing workloads, e.g. in the form of CUDA kernels.

The router RTL developed in support of this dissertation provides an extensive set of parameters; while this allows many aspects of the router to be freely configured, the fact that each added parameter leads to an exponential increase in the number of possible configurations renders finding the optimal parameters for a given set of constraints through exhaustive design space exploration infeasible. As a result, it would be of considerable benefit it the router could be integrated into an optimization framework like the one described in [3].

Finally, it stands to reason that the router's code base would benefit from the use of a higher-level elaboration framework like Genesis2 [84] in place of the built-in *parameter* and *generate* constructs available in Verilog-2001. As a considerable part of the router code's complexity stems from the extensive use of these language features that was necessary to achieve the desired levels of configurability and generality, this should make the code both easier to understand and to modify, and thus encourage more researchers to validate new microarchitectural ideas down to the level of RTL.

Appendix A

Router RTL Overview

In the course of the work described in this dissertation, we developed a parameterized RTL implementation of a state-of-the-art Virtual Channel (VC) router. In doing so, we followed three primary design goals:

- To provide a generic, flexible router implementation that, through extensive use of parameterization, is able to support a wide variety of configurations in terms of router radix, number of VCs, allocators and other key design parameters, enabling rapid design space exploration.
- To allow for individual configurations to be synthesized into reasonably efficient hardware implementations using industry-standard design flows, enabling detailed cost and performance evaluations.
- To provide a modular design that facilitates extensibility by interested third parties, enabling them to leverage the existing code base for microarchitectural research. The router model is implemented in the industry-standard Verilog hardware description language, and includes programmable pseudo-random traffic generators and signature analysis modules, which facilitate the setup of simple test networks.

The router RTL was used as the basis for conducting evaluations of delay, area and energy efficiency in Chapters 2, 3, 4 and 5; it has since found use in a number of other research efforts at Stanford and beyond [10, 11, 44, 53, 55, 56, 59, 63, 64, 72, 73].

Table A.1: Source tree for router RTL.

Directory	Description
src/	This directory contains the implementation of the individual router components, as well as the top-level wrappers that tie them together into a complete router instance.
src/clib/	This directory contains a library of generic components that are used in many places throughout the router.
verif/	This directory contains verification testbenches, traffic generators, as well as modules that monitor the traffic that flows through each router instance and check for incorrect behavior.
verif/router	A testbench with a single router instance. This testbench is particularly useful for quickly evaluating different router designs for arbitrary topologies.
verif/mesh_3x3	A testbench with a 3×3 Mesh of routers. This testbench allows the routers to be tested with more realistic traffic; however, it is inherently limited to a simple 3×3-node Mesh topology with a single node attached to each router.

Table A.1 shows an overview of the router source tree. The typical mode of using the router implementation is by instantiating the *router_wrap* wrapper module that is provided in the *src/* directory. Router instances can be customized individually by explicitly passing parameter assignments upon instantiation, or in bulk by editing the *parameters.v* file, which is included by *router_wrap*. The testbenches included in the source tree can assist in debugging microarchitectural modifications and in verifying the correct operation of the router; they can also be used to quickly gather simple performance metrics.

Table A.2 provides a brief overview of the parameters of the *router_wrap* module. Additional documentation is available in the form of comments throughout the source code of the router.

Table A.2: Design parameters for router RTL.

Parameter	Description
topology	Selects a network topology. This determines the configuration of the routing logic and the number of ports that connect to neighboring routers.
buffer_size	Selects the total input buffer size per port in flits.
num_message_classes	Selects the number of message classes (cf. Section 4.3).
num_resource_classes	Selects the number of resource classes (cf. Section 4.3).
num_vcs_per_class	Determines the number of VCs that are assigned to each packet class (cf. Section 4.3).
num_nodes	Configures the total number of nodes in the network.

Continued on next page

Table A.2 – *Continued from previous page*

Parameter	Description
num_dimensions	Selects the number of dimensions for the network topology; this directly corresponds to the n parameter in [20], and it determines the k parameter in combination with *num_nodes* and *num_nodes_per_router*.
num_nodes_per_router	Selects the concentration factor; i.e., the number of network nodes attached to each router.
packet_format	Selects a packet encoding.
flow_ctrl_type	Selects a flow control scheme; only credit-based flow control is supported.
flow_ctrl_bypass	Determines whether incoming credits are accounted for immediately as they arrive or whether they merely update the credit count for the next cycle. Enabling this can affect critical path delay.
max_payload_length	Selects the maximum number of payload flits per packet. Ignored when using explicit head and tail bits.
min_payload_length	Selects the minimum number of payload flits per packet. Ignored when using explicit head and tail bits.
router_type	Selects a router implementation variant. In particular, this parameter selects whether to use separate VC and switch allocators or combined VC and switch allocation (cf. Section 5.4).

Continued on next page

Table A.2 – *Continued from previous page*

Parameter	Description
enable_link_pm	If enabled, each network channel includes an additional activity indicator signal that allows the receiving logic at the downstream router to be clock-gated.
flit_data_width	Determines the width of each network channel, excluding control wires.
error_capture_mode	Enables and configures error checking logic inside the router.
restrict_turns	Enables synthesis optimizations based on routing restrictions.
routing_type	Selects the type of routing logic to generate; the only supported implementation uses one or more phases of Dimension-Order Routing (DOR).
dim_order	Selects the order in which dimensions are traversed.
input_stage_can_hold	For wormhole routers only, allows the input pipeline stage to be used as part of the input buffer.
fb_regfile_type	Selects an implementation variant for the register file used to implement input buffers.
fb_mgmt_type	Selects a buffer management scheme for the input buffer (cf. Section 6.3 and 6.4).

Continued on next page

Table A.2 – *Continued from previous page*

Parameter	Description
fb_fast_peek	If enabled, attempt to improve the timing for reading the next packet's header information from the input buffer. This leads to increased cost.
disable_static_reservations	*Reserved.*
explicit_pipeline_register	If disabled, modify the timing of control signals to the input buffer's read port such that its output can connect to the crossbar directly, rather than through an explicit pipeline stage. This saves area and energy, but can lead to increased critical path delay.
gate_buffer_write	If enabled, attempt to clock-gate the input buffer if bypassing succeeds. This can increase the critical path.
dual_path_alloc	If enabled, use separate allocators for buffered and newly arriving flits. Only supported when using combined allocation.
dual_path_allow_conflicts	If enabled, resolve output conflicts when using dual-path allocation via arbitration; otherwise, simply perform collision detection.
dual_path_mask_on_ready	When using dual-path allocation, only mask requests from newly arriving flits if a conflicting VC has flits ready to go. This can lead to increased delay.

Continued on next page

Table A.2 – *Continued from previous page*

Parameter	Description
precomp_ivc_sel	If enabled, attempt to pre-compute input arbitration decisions; i.e., pre-select a winning input VC for each input port one cycle ahead. Only supported when using combined allocation.
precomp_ip_sel	If enabled, attempt to pre-compute output arbitration decisions; i.e., pre-select a winning input port for each output port one cycle ahead. Only supported when using combined allocation.
elig_mask	Determines when VCs become available for re-allocation. In particular, this parameter can be used to enable atomic VC allocation (cf. Section 6.2.3).
vc_alloc_type	Selects the VC allocator implementation variant (cf. Section 4.2). Not supported when using combined allocation.
vc_alloc_arbiter_type	Selects which type of arbiter to use for implementing VC allocation.
vc_alloc_prefer_empty	Determines whether VC allocation should prioritize VCs that are currently empty. This can increase the critical path delay.
sw_alloc_type	Selects a switch allocator implementation variant. Not supported when using combined allocation.

Continued on next page

Table A.2 – *Continued from previous page*

Parameter	Description
sw_alloc_arbiter_type	Selects which type of arbiter to use in implementing switch allocation (cf. Section 5.2).
sw_alloc_spec_type	Enables and configures speculative switch allocation (cf. Section 5.3). Not supported when using combined allocation.
crossbar_type	Selects an implementation variant for the crossbar.
reset_type	Determines whether to use synchronous or asynchronous reset.

Bibliography

[1] Minseon Ahn and Eun Jung Kim. Pseudo-Circuit: Accelerating Communication for On-Chip Interconnection Networks. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[2] Mani Azimi, Donglai Dai, Akhilesh Kumar, Andres Meija, Dongkook Park, Roy Saharoy, and Aniruddha S Vaidya. Flexible and Adaptive On-Chip Interconnect for Tera-Scale Architectures. *Intel Technology Journal*, 13(4), 2009.

[3] Omid J Azizi. *Design and Optimization of Processors for Energy Efficiency: A Joint Architecture-Circuit Approach*. PhD thesis, Stanford University, August 2010.

[4] James Balfour and William J Dally. Design Tradeoffs for Tiled CMP On-Chip Networks. In *Proceedings of the 20th annual International Conference on Supercomputing*, pages 187–198, 2006.

[5] Arnab Banerjee and Simon W Moore. Flow-Aware Allocation for On-Chip Networks. In *Proceedings of the Third International Symposium on Networks-on-Chip*, pages 183–192, 2009.

[6] Elvira Baydal, Pedro López, and José Duato. A Family of Mechanisms for Congestion Control in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(9):772–784, 2005.

[7] Daniel U Becker and William J Dally. Allocator Implementations for Network-on-Chip Routers. In *Proceedings of the 2009 ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis*, 2009.

[8] Daniel U Becker, Nan Jiang, George Michelogiannakis, and William J Dally. Adaptive Backpressure: Efficient Buffer Management for On-Chip Networks. In *Proceedings of the 30th International Conference on Computer Design*, 2012.

[9] Luca Benini and Giovanni de Micheli. Networks on Chip: A New Paradigm for Systems on Chip Design. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 418–419, 2002.

[10] Kshitij Bhardwaj, Koushik Chakraborty, and Sanghamitra Roy. An MILP-based aging-aware routing algorithm for NoCs. In *Proceedings of the 2012 Design Automation & Test in Europe Conference & Exhibition*, pages 326–331, 2012.

[11] Kshitij Bhardwaj, Koushik Chakraborty, and Sanghamitra Roy. Towards Graceful Aging Degradation in NoCs Through an Adaptive Routing Algorithm. In *Proceedings of the 49th Annual Design Automation Conference*, pages 382–391, 2012.

[12] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[13] Shekhar Borkar. Thousand Core Chips—A Technology Perspective. In *Proceedings of the 44th Design Automation Conference*, pages 746–749, 2007.

[14] James Chen. *Self-Calibrating On-Chip Interconnects*. PhD thesis, Stanford University, March 2012.

[15] Xuning Chen and Li-Shiuan Peh. Leakage Power Modeling and Optimization in Interconnection Networks. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 90–95, 2003.

[16] Yungho Choi and Timothy Mark Pinkston. Evaluation of Queue Designs for True Fully Adaptive Routers. *Journal of Parallel and Distributed Computing*, 64(5):606–616, 2004.

[17] William J Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, 1992.

[18] William J Dally and Charles L Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, 36(5):547–553, 1987.

[19] William J Dally and Brian Towles. Route Packets, Not Wires: On-Chip Inteconnection Networks. In *Proceedings of the 38th Conference on Design Automation*, pages 684–689, 2001.

[20] William J Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2004.

[21] José G Delgado-Frias and Girish B Ratanpal. A VLSI Wrapped Wave Front Arbiter for Crossbar Switches. In *Proceedings of the 11th Great Lakes Symposium on VLSI*, pages 85–88, 2001.

[22] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), October 1974.

[23] Giorgios Dimitrakopoulos, Nikos Chrysos, and Kostas Galanopoulos. Fast Arbiters for On-Chip Network Switches. In *Proceedings of the 2008 IEEE International Conference on Computer Design*, pages 664–670, 2008.

[24] José Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, 1993.

[25] Chris Fallin, Chris Craik, and Onur Mutlu. CHIPPER: A Low-complexity Bufferless Deflection Router . In *Proceedings of the 17th International Symposium on High-Performance Computer Architecture*, 2011.

[26] Lester R Ford Jr and Delbert R Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[27] Joshua Friedrich and Brad Heaney. Designing High Performance Systems-on-Chip (Wednesday keynote address). In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference*, 2012.

[28] Mike Galles. Spider: A High-Speed Network Interconnect. *IEEE Micro*, 17(1):34–39, 1997.

[29] Ganesh Gopalakrishnan. Developing Micropipeline Wavefront Arbiters. *IEEE Design & Test of Computers*, 11(4):55–64, 1994.

[30] Paul Gratz, Changkyu Kim, Robert McDonald, Stephen W Keckler, and Doug Burger. Implementation and Evaluation of On-Chip Network Architectures. In *Proceedings of the 2006 International Conference on Computer Design*, pages 477–484, 2006.

[31] 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*, pages 401–412, 2011.

[32] 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 IEEE/ACM International Symposium on Microarchitecture*, pages 268–279, 2009.

[33] Pierre Guerrier and Alain Greiner. A Generic Architecture for On-Chip Packet-Switched Interconnections. In *Proceedings of the 2000 Design, Automation and Test in Europe Conference and Exhibition*, pages 250–256, 2000.

[34] R Curtis Harting, Vishal Parikh, and William J Dally. Energy and Performance Benefits of Active Messages. Technical Report 131, Stanford University, Concurrent VLSI Architecture Group, February 2012.

[35] Joel Hestness and Stephen W Keckler. Netrace: Dependency-Tracking Traces for Efficient Network-on-Chip Experimentation. Technical Report TR-10-11, The University of Texas at Austin, Department of Computer Science, May 2011.

[36] Ron Ho, Ken Mai, and Mark Horowitz. Efficient On-Chip Global Interconnects. In *Digest of Technical Papers of the 2003 Symposium on VLSI Circuits*, pages 271–274, 2003.

[37] Ron Ho, Tarik Ono, Robert David Hopkins, Alex Chow, Justin Schauer, Frankie Y Liu, and Robert Drost. High-Speed and Low-Energy Capacitively-Driven On-Chip Wires. *IEEE Journal of Solid-State Circuits*, 43(1):52–60, January 2008.

[38] Raymond R Hoare, Zhu Ding, and Alex K Jones. A Near-optimal Real-time Hardware Scheduler for Large Cardinality Crossbar Switches. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[39] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. A 5-GHz Mesh Interconnect for a Teraflops Processor. *IEEE Micro*, 27(5):51–61, 2007.

[40] Jingcao Hu and Radu Marculescu. Application-Specific Buffer Space Allocation for Networks-on-Chip Router Design. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, pages 354–361, 2004.

[41] Chung-Hsun Huang, Jinn-Shyan Wang, and Yan-Chao Huang. Design of High-Performance CMOS Priority Encoders and Incrementer/Decrementers Using Multilevel Lookahead and Multilevel Folding Techniques. *IEEE Journal of Solid-State Circuits*, 37(1):63–76, 2002.

[42] Ting-Chun Huang, Umit Y Ogras, and Radu Marculescu. Virtual Channels Planning for Networks-on-Chip. In *Proceedings of the Eigth International Symposium on Quality Electronic Design*, pages 879–884, 2007.

[43] James Hurt, Andrew May, Xiaohan Zhu, and Bill Lin. Design and Implementation of High-Speed Symmetric Crossbar Schedulers. In *Proceedings of the 1999 IEEE Conference on Communications*, pages 1478–1483, 1999.

[44] Andrew B Kahng, Bill Lin, and Siddhartha Nath. Explicit Modeling of Control and Data for Improved NoC Router Estimation. In *Proceedings of the 49th Annual Design Automation Conference*, pages 392–397, 2012.

[45] John Kim, James Balfour, and William J Dally. Flattened Butterfly Topology for On-Chip Networks. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[46] Jongman Kim, Chrysostomos Nicopoulos, Dongkook Park, Vijaykrishnan Narayanan, Mazin S Yousif, and Chita R Das. A Gracefully Degrading and Energy-Efficient Modular Router Architecture for On-Chip Networks. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 4–15, 2006.

[47] Amit Kumar, Partha Kundu, Arvind Singh, Li-Shiuan Peh, and Niraj K Jha. A 4.6Tbits/s 3.6GHz Single-cycle NoC Router with a Novel Switch Allocator in 65nm CMOS. In *Proceedings of the 25th International Conference on Computer Design*, 2007.

[48] Rakesh Kumar, Victor Zyuban, and Dean M Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 408–419, 2005.

[49] Mingche Lai, Zhiying Wang, Lei Gao, Hongyi Lu, and Kui Dai. A Dynamically-Allocated Virtual Channel Architecture with Congestion Awareness for On-Chip Routers. In *Proceedings of the 45th Conference of Design Automation*, pages 630–633, 2008.

[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*, pages 89–100, 2008.

[51] Michael M Lee, John Kim, Dennis Abts, Michael R Marty, and Jae W Lee. Probabilistic Distance-based Arbitration: Providing Equality of Service for Many-core CMPs. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[52] Jin Liu and José G Delgado-Frias. DAMQ Self-Compacting Buffer Schemes for Systems with Network-On-Chip. In *Proceedings of the 2005 International Conference on Computer Design*, pages 97–103, 2005.

[53] Sheng Ma, Natalie Enright Jerger, and Zhiying Wang. Whole Packet Forwarding: Efficient Design of Fully Adaptive Routing Algorithms for Networks-on-Chip. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture*, 2012.

[54] Nick McKeown. The iSLIP Scheduling Algorithm for Input-Queued Switches. *IEEE/ACM Transactions on Networking*, 7(2):188–201, 1999.

[55] George Michelogiannakis, James Balfour, and William J Dally. Elastic-Buffer Flow Control for On-Chip Networks. In *Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, pages 151–162, 2009.

[56] George Michelogiannakis, Daniel U Becker, and William J Dally. Evaluating Elastic Buffer and Wormhole Flow Control. *IEEE Transactions on Computers*, 2010.

[57] George Michelogiannakis, Nan Jiang, Daniel U Becker, and William J Dally. Packet Chaining: Efficient Single-Cycle Allocation for On-Chip Networks. *Computer Architecture Letters*, 10(2):33–36, 2011.

[58] George Michelogiannakis, Nan Jiang, Daniel U Becker, and William J Dally. Packet Chaining: Efficient Single-Cycle Allocation for On-Chip Networks. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 83–94, 2011.

[59] George Michelogiannakis, Daniel Sanchez, William J Dally, and Christos Kozyrakis. Evaluating Bufferless Flow Control for On-Chip Networks. In *Proceedings of the Fourth International Symposium on Networks-on-Chip*, pages 9–16, 2010.

[60] Thomas Moscibroda and Onur Mutlu. A Case for Bufferless Routing in On-Chip Networks. In *Proceedings of the 36th annual International Symposium on Computer Architecture*, pages 196–207, 2009.

[61] Shubhendu S Mukherjee, Federico Silla, Peter Bannon, Joel S Emer, Steve Lang, and David Webb. A Comparative Study of Arbitration Algorithms for the Alpha 21364 Pipelined Router. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–234, 2002.

[62] Robert D Mullins, Andrew West, and Simon W Moore. Low-Latency Virtual-Channel Routers for On-Chip Networks. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 188–197, 2004.

[63] Yohei Nakata, Yukihiro Takeuchi, Hiroshi Kawaguchi, and Masahiko Yoshimoto. A Process-Variation-Adaptive Network-on-Chip with Variable-Cycle Routers. In *Proceedings of the 14th Euromicro Conference on Digital System Design*, pages 801–804, 2011.

[64] Vivek S Nandakumar and Malgorzata Marek-Sadowska. A Low Energy Network-on-Chip Fabric for 3-D Multi-Core Architectures. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2(2):266–277, June 2012.

[65] Nan Ni, Marius Pirvu, and Laxmi Bhuyan. Circular Buffered Switch Design with Wormhole Routing and Virtual Channels. In *Proceedings of the 1998 International Conference on Computer Design: VLSI in Computers and Processors*, pages 466–473, 1998.

[66] Chrysostomos Nicopoulos, Dongkook Park, Jongman Kim, Vijaykrishnan Narayanan, Mazin S Yousif, and Chita R Das. ViChaR: A Dynamic Virtual

Channel Regulator for Network-on-Chip Routers. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 333–346, 2006.

[67] Wladek Olesinski, Hans Eberle, and Nils Gura. PWWFA: The Parallel Wrapped Wave Front Arbiter for Large Switches. In *Proceedings of the 2007 IEEE Workshop on High Performance Switching and Routing*, 2007.

[68] Wladek Olesinski, Nils Gura, Hans Eberle, and Andres Mejia. Low-Latency Scheduling in large switches. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communication Systems*, pages 87–96, 2007.

[69] Kunle Olukotun, Lance Hammond, and James Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[70] Jin Ouyang and Yuan Xie. LOFT: A High Performance Network-on-Chip Providing Quality-of-Service Support. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[71] Gupta Pankaj and Nick Mckeown. Designing and Implementing a Fast Crossbar Scheduler. *IEEE Micro*, 19(1):20–28, 1999.

[72] Michael K Papamichael and James C Hoe. CONNECT: Re-Examining Conventional Wisdom for Designing NoCs in the Context of FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 37–46, 2012.

[73] Michael K Papamichael, James C Hoe, and Onur Mutlu. FIST: A Fast, Lightweight, FPGA-Friendly Packet Latency Estimator for NoC Modeling in Full-System Simulations. In *Proceedings of the Fifth IEEE/ACM International Symposium on Networks-on-Chip*, pages 137–144, 2011.

[74] Dongkook Park, Reetuparna Das, Chrysostomos Nicopoulos, Jongman Kim, Vijaykrishnan Narayanan, Ravishankar K Iyer, and Chita R Das. Design of a

Dynamic Priority-Based Fast Path Architecture for On-Chip Interconnects. In *Proceedings of the 15th Symposium on High Performance Interconnects*, pages 15–20, 2007.

[75] Jonhoo Park, Brian W O'Krafka, Stamatis Vassiliadis, and José G Delgado-Frias. Design and Evaluation of a DAMQ Multiprocessor Network With Self-Compacting Buffers. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 713–722, 1994.

[76] Li-Shiuan Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks*. PhD thesis, Stanford University, August 2001.

[77] Li-Shiuan Peh and William J Dally. A Delay Model and Speculative Architecture for Pipelined Routers. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 255–266, 2001.

[78] Gregory F Pfister and V Alan Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, 34(10), 1985.

[79] Thomas B Preußer, Martin Zabel, and Rainer G Spallek. About Carries and Tokens: Re-Using Adder Circuits for Arbitration. In *Proceedings of the 2005 IEEE Workshop on Signal Processing Systems Design and Implementations*, pages 59–64, 2005.

[80] Mostafa Rezazad and Hamid Sarbazi-Azad. The Effect of Virtual Channel Organization on the Performance of Interconnection Networks. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pages 264–271, 2005.

[81] Daniel Sanchez, George Michelogiannakis, and Christos Kozyrakis. An Analysis of Interconnection Networks for Large Scale Chip-Multiprocessors. *ACM Transactions on Archictecture and Code Optimization*, 7(1), 2010.

[82] Daeho Seo, Akif Ali, Won-Taek Lim, Nauman Rafique, and Mithuna Thottethodi. Near-Optimal Worst-Case Throughput Routing for Two-Dimensional Mesh Networks. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 432–443, 2005.

[83] Daeho Seo and Mithuna Thottethodi. Table-lookup based Crossbar Arbitration for Minimal-Routed, 2D Mesh and Torus Networks. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 2007.

[84] Ofer Shacham. *Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms*. PhD thesis, Stanford University, May 2011.

[85] Keun Sup Shim, Myong Hyon Cho, Michael Kinsy, Tina Wen, Mieszko Lis, G Edward Sug, and Srinivas Devadas. Static Virtual Channel Allocation in Oblivious Routing. In *Proceedings of the Third ACM/IEEE International Symposium on Networks-on-Chip*, pages 38–43, 2009.

[86] Eung S Shin, Vincent J Mooney III, and George F Riley. Round-robin Arbiter Design and Generation. In *Proceedings of the 15th International Symposium on System Synthesis*, pages 243–248, 2002.

[87] Arjun Singh. *Load-Balanced Routing in Interconnection Networks*. PhD thesis, Stanford University, March 2005.

[88] Ivan E Sutherland, Bob Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1999.

[89] Yuval Tamir and Hsin-Chuo Chi. Symmetric Crossbar Arbiters for VLSI Communication Switches. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):13–27, 1993.

[90] Yuval Tamir and Gregory L Frazier. High-Performance Multi-Queue Buffers for VLSI Communication Switches. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 343–354, 1988.

[91] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the RAW Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 2–13, 2004.

[92] Mithuna Thottethodi, Alvin R Lebeck, and Shubhendu S Mukherjee. Self-Tuned Congestion Control for Multiprocessor Networks. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 107–118, 2001.

[93] Yonggang Tian, Xiaodong Tu, Li Wen, Kai Wang, and Yashe Liu. PPAwFE: A Novel High-Speed Crossbar Scheduling Algorithm. In *Proceedings of the 2005 International Conference on Communications, Circuits and Systems*, pages 673–677, 2005.

[94] Leslie G Valiant and Gordon J Brebner. Universal Schemes for Parallel Communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, 1981.

[95] Sriram Vangal, Arvind Singh, Jason Howard, Saurabh Dighe, Nitin Borkar, and Atila Alvandpour. A 5.1GHz 0.34mm2 Router for Network-on-Chip Applications. In *Proceedings of the 2007 IEEE Symposium on VLSI Circuits*, pages 42–43, 2007.

[96] Hangsheng Wang, Li-Shiuan Peh, and Sharad Malik. Power-driven Design of Router Microarchitectures in On-chip Networks. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–116, 2003.

[97] Ling Wang, Jianwen Zhang, Xiaoqing Yang, and Dongxin Wen. Router with Centralized Buffer for Network-on-Chip. In *Proceedings of the 19th Great Lakes Symposium on VLSI*, pages 469–474, 2009.

[98] Terry Tao Ye, Giovanni de Micheli, and Luca Benini. Analysis of Power Consumption on Switch Fabrics in Network Routers. In *Proceedings of the 39th Design Automation Conference*, pages 524–529, 2002.

[99] Min Zhang and Chiu-Sing Choy. Low-Cost Allocator Implementations for Networks-on-Chip Routers. *VLSI Design*, 2009.