The Dissertation Committee for Qinzhe Wu
certifies that this is the approved version of the following dissertation:

Architectural Support for Message Queue Task Parallelism

Committee:

Lizy John, Supervisor

Jonathan Beard

Andreas Gerstlauer

Vijay Garg

Simon Peter

Architectural Support for Message Queue Task Parallelism

by

Qinzhe Wu

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August **2023**

Dedication

Dedicated to my family and friends for their consistent encouragement and support.

Acknowledgments

I would like to express my heartfelt gratitude to my supervisor, Professor Lizy John, for her invaluable guidance, constant patience, and encouragement during the course of my PhD study. Professor John imparted upon me the skills to work as a researcher, and also taught me how to effectively present my work to others. From the early stages of my graduate school journey, she graciously let me explore various directions and discussed diverse research topics with me. During moments when I faced difficulties in advancing my research and felt unsure about the way forward, she provided unwavering support and rekindled my determination to persevere.

I want to acknowledge Jonathan Beard for his tremendous help during our collaboration, which commenced during my internship at Arm in 2019. He has spent countless hours on meeting with me, patiently addressing my questions, and offering insightful suggestions. It is undeniable that he contributed a lot to my accomplishments, and this dissertation, along with my other research papers, has greatly benefited from his meticulous review and meticulous polishing. Furthermore, I would like to extend my sincerest appreciation to other members of my dissertation committee: Professors Andreas Gerstlauer, Vijay Garg, and Simon Peter. Their constructive feedback and comments have played a pivotal role in the overall refinement of my work.

I feel so fortunate to have had the companionship and support of numerous brilliant and amiable graduate students at UT. I first want to thank Reena Panda, Shuang Song, and Jiajun Wang for leading me towards the path of research. Furthermore, I am deeply thankful to Ruihao Li, Ashen Ekanayake, Bagus Hanindhito, and Snehil Verma for their availability during moments when I sought discussion and assistance with my experiments. I would also like to sincerely appreciate Zhigang Wei, Aman Arora, Zachary Susskind, Steven Flolid, Siyuan Ma and Mugdha Jadhao for providing feedback in my presentations at LCA group meetings. Moreover, I also

want to extend my gratitude to Zhuoran Zhao, Shijia Wei, Wenqi Yin, Mochamad Asri, and Tianhao Zheng for bringing immense joy and camaraderies to our shared days and nights at campus. Lastly, I must express my thanks to Zixuan Jiang for alleviating the monotony of working from home during the COVID pandemic and for including me in various engaging events with his friends.

Most of all, I am forever grateful to my dear family and girlfriend. I am indebted to my parents for their unwavering support, continuous love, and belief in my decision to embark on and successfully complete my PhD journey.

Abstract

Architectural Support for Message Queue Task Parallelism

Qinzhe Wu, PhD
The University of Texas at Austin, 2023

SUPERVISOR: Lizy John

The scaling of threads is an attractive way to exploit task-level parallelism and boost performance. From the perspective of software programming, many applications (e.g., network package processing, SQL queries) could be composite of a set of small tasks. Those tasks are arranged in a data flow graph and each task is undertaken by some threads. Message queues are often used to coordinate the tasks among the threads. On the other side, thread scaling is in favor of the hardware advancing trend that there are more Processing Elements (PE) in modern Chip Multiprocessors (CMP) than ever before. This is because single PE cannot simply run faster due to power and thermal limitations; instead architects have to use more transistors for increasing number of PEs, in order to improve the overall computing power of a processor.

Unfortunately, this paradigm using message queues to drive parallel tasks sometime leads to diminishing performance returns due to issues lying in the architecture and system design. Particularly, the conventional coherent shared-memory architectures let task-parallel workloads suffer from unnecessary synchronization overhead and load-to-use latency. For instance, when passing messages through queues, multiple threads could contend for the exclusivity of the cacheline where the shared queue data structure stays. The more threads, the more severe the contention is,

because every transition upgrading a cacheline from shared to exclusive state needs to invalidate more copies in the private caches of other cores [1], and waits for the acknowledgements from more cores. Such a overhead hurts the scalability of threads synchronizing via message queues. Adding to the coherence overhead, the load-to-use latency (from a consumer requesting data until the data being moved to the consumer to use) is often on the critical path, slowing down the computation. This is because the cache hierarchy in modern processors creates some layers of local storage to buffer data separately for different cores. Therefore, serving message queue data in an on-demand manner incurs longer load-to-use latency. It is also challenging to schedule message-driven tasks to use cores efficiently when arrival rate and service rate mismatch. It wastes CPU cycles if a runtime system leaves tasks blocked on full/empty message queues, while switching tasks has additional scheduling overheads. Diverse system topologies further complicate the problem, as the scheduling also needs to take data locality into consideration.

This dissertation explores architectural supports for enhancing the scalability of message queue task parallelism, reducing the load-to-use latency, as well as avoiding blocking. Specifically, this dissertation designs and evaluates a message queue architecture that lowers the overhead of synchronization on shared queue states, a speculation technique to hide the load-to-use latency, as well as a locality-aware message queue runtime system with low overhead on scheduling and buffer resizing.

The first contribution of the dissertation is *Virtual-Link* scalable message queue architecture (*VL*). Instead of having threads access the shared queue state variables (i.e., head, tail, or lock) atomically, *VL* provides configurable hardware support, providing both data transfer and synchronization. Unlike other hardware queue architectures with dedicated network, *VL* reuses the existing cache coherence network and delivers a virtualized channel as if there were a direct link (or route) between two arbitrary *PE*s. *VL* facilitates efficient synchronized data movement between *M:N*

[1] cores are used interchangeably as Processing Elements (*PE*) in this dissertation.

producers and consumers with several benefits: (i) the number of sharers on synchronization primitives is reduced to zero, eliminating a primary bottleneck of traditional lock-free queues, (ii) memory spills, snoops, and invalidations are reduced, (iii) data stays on the fast path (inside the interconnect) a majority of the time.

Another contribution of the dissertation is *SPAMeR* speculation mechanism. *SPAMeR* has the capability to speculatively push messages in anticipation of consumer message requests. With the speculation, the latency of moving data from the source to the consumer that needs the data could be partially or fully overlapped with the message processing time. Unlike pre-fetch approaches which predict what addresses to fetch next, with a queue we know exactly what data is needed next but not *when* it is needed; *SPAMeR* proposes algorithms to learn from queue operation history in order to predict this.

Finally the dissertation contributes *ARMQ* locality-aware runtime. *ARMQ* collects a set of approaches that avoids message queue blocking, ranging from the most general yielding, to dynamically resizing the buffer, and to spawning helper tasks. On one hand, *ARMQ* minimizes the overheads (e.g., wasteful polling, context switch, memory allocation and copying etc.) with a few techniques (e.g., userspace threading, chunk-based ringbuffer etc.) On the other hand, *ARMQ* schedules the message-driven tasks precisely and opportunely, in order to maximize the data locality preserved (in favor of cache) and balance the resource allocation.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

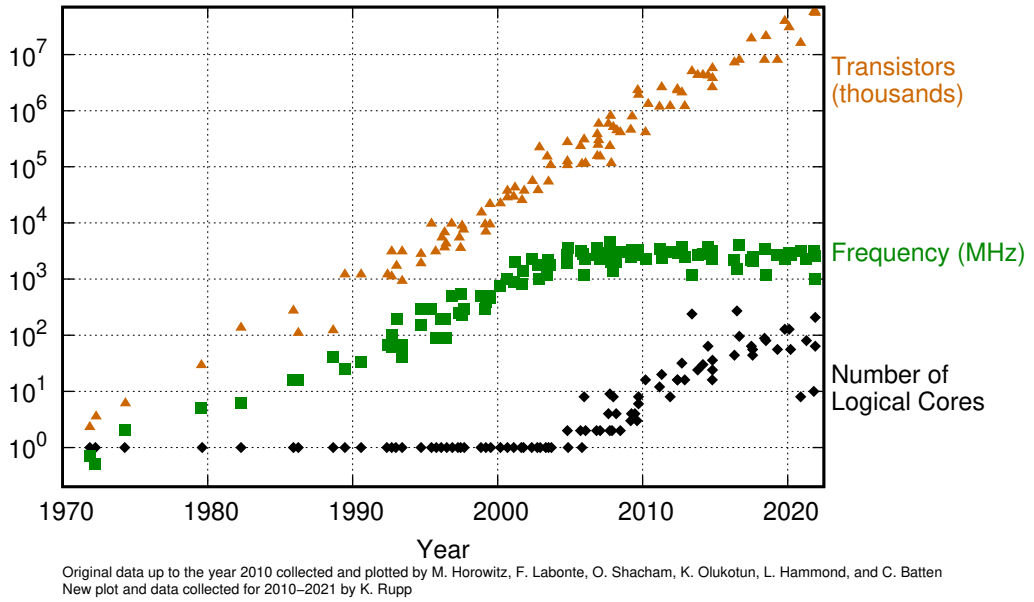
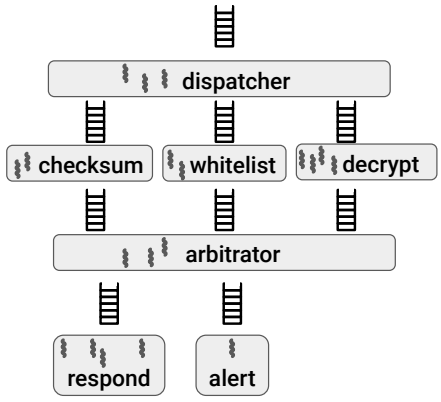


Figure 1.1: 50 years of microprocessor trend. The number of cores rises after the frequency reaches the upper bound.

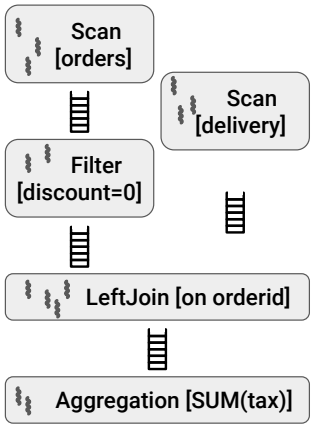
The design of microprocessors has diverged towards more cores instead of higher speed in past decades (as shown in Figure 1.1 [72]). Microprocessors are getting more transistors thanks to the technology advancements, and were used to putting the more transistors available on building stronger and faster uniprocessors. Unfortunately, after hitting the power and thermal limitations, microprocessors cannot gain more performance by simply rising the frequency any more. The trend has changed to integrating more Processing Elements (PE) in Chip Multiprocessors (CMP) to add overall computing power. Reacting to that change, people find task parallelism in many applications in order to make use of PEs in the way that tasks are assigned to different PEs and processed in parallel. The tasks could be as fine-grain

as hundreds of instructions and there are compiler techniques invented to extract them automatically [65, 9]. Figure 1.2a and 1.2b show network package processing and SQL query, respectively, as two examples of task parallel workloads. Each block represents a task, and is undertaken by one or multiple threads. Some of the tasks have no dependencies between each other and can be parallelized freely, for example, calculating checksum, checking whitelist, and payload decryption in Figure 1.2a. For tasks in pipelines, packaged or sliced data flows through stages, and different tasks could be performed on different part of data concurrently (e.g., scan, filter, join, and aggregation in Figure 1.2b). In most applications, tasks are usually pre-defined by the programmers, while sometimes tasks could also be dynamically established according to the user input, such as the SQL query.




```
while (!stream.eof()) {
  future<bool> corrupted = async checksum(pkt);
  future<bool> blocked = async whitelist(pkt);
  future<uint8_t*> payload = async decrypt(pkt);
  if (corrupted.get() || blocked.get()) {
    alert(pkt);
  } else {    respond(payload.get());  }
}
```

(a) network package processing




```
SELECT
  orders.id, SUM(tax)
FROM orders LEFT JOIN delivery
  ON orders.id = delivery.orderid
WHERE discount = 0
GROUP BY orders.orderid
```

(b) SQL query

Figure 1.2: Examples of applications with task parallelism. Diagrams are visualization of the pseudo-code below. Blocks in the diagrams represent tasks, and darker curly lines as threads taking the task. Message queues connect the tasks.

Message queue is a useful data structure for task parallel computing. Messages entering the queue could carry the data needed for computation, or wrap the infor-

mation that is necessary for the computation task, such as pointer to the payload. No matter what the message contains, the delivery of the message itself serves as a signal. For example in Figure 1.2, there are message queues between producer/upstream threads and consumer/downstream threads for data delivery as well as task synchronization. The producer/upstream thread pushes the message into the queue after finishing its task, and the consumer/downstream thread waits until popping the message out from the queue to start its task. This dissertation refers to such program paradigm as message queue task parallelism.

Message queues usually comprise of buffers and metadata (e.g., head, tail pointers). The buffer capacity of a message queue could be either fixed or adjustable on demand. There are some basic operations of message queues, such as checking whether the queue is empty or full (if fixed capacity), push/enqueue, and pop/dequeue a message. The push and pop operations could be either blocking, meaning waiting until the operation is fulfilled, or non-blocking, that the program moves forward anyways handling both the success and failure case of the operation. Because the basic operations of message queues (e.g., checking capacity, push/enqueue and pop/dequeue) would involve access to the buffer and metadata, those underlying structures become shared among threads (unless a single thread is both the only producer and the only consumer of a message queue).

1.1 Problem Description

Since the emergence of *CMP*, shared-memory has become the de facto solution to synchronize data across cores. It is convenient to implement message queues atop of shared-memory, but it is difficult to scale due to the concurrent access to the shared queue states.

For instance, Figure 1.3 shows the operation latency of popular software queue implementation [21] with lock-free optimization increases along with the number of

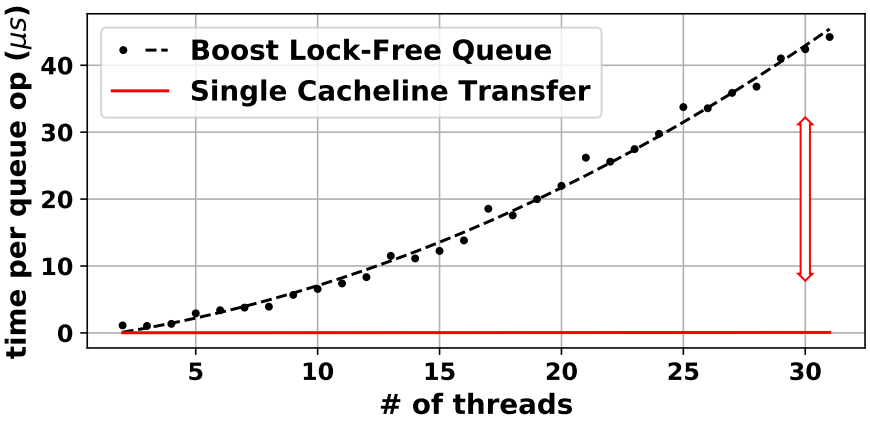


Figure 1.3: The overheads in a popular software queue implementation, Boost Lock-Free Queue increase with number of threads. The red solid line at the bottom shows the latency to transfer queue data without synchronization overhead.

threads that perform queue operations concurrently (black dots and the projected dashed lined). The performance gap enlarges comparing the queue operation latency with the latency of simply moving data across-core (red solid line). The is because the shared queue states (i.e., head, tail pointers, lock) are stored at the same memory location for both the producers and the consumers, the coherence overhead of the queue operation would increases with the number of threads. As Figure 1.4 illustrates, when updating the shared states or data, cache coherence protocol (e.g., MESI) makes sure the value is updated properly across cores through a set of invalidation requests and acknowledgement transactions on the Network-on-Chip (NoC). The more threads using the queue, the more coherence traffic occur and slower for a thread to get the exclusive access of the queue states and finish the operations. The state-of-the-art hardware queue designs[15, 69, 51, 55, 90] achieves better scalability at the considerable hardware cost, and they have limited flexibility to support multi-producer multi-consumer queues.

Another problem arises with message queue task parallelism in multi-core systems is the load-to-use latency. In modern CMP, the cores performing dependent tasks in parallel will access their own private cache for data, which means for message queue task parallel workloads, data would often move from one place to another

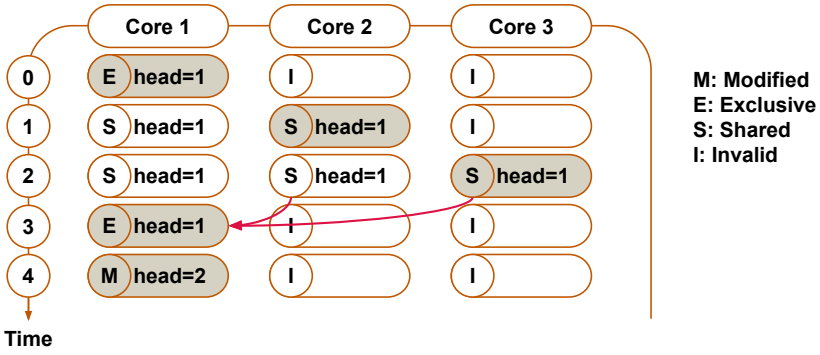
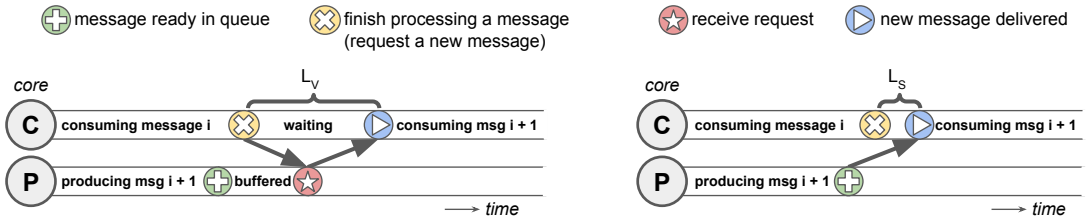


Figure 1.4: Example of updating a shared state under MESI cache coherence protocol. Each row shows the states of cachelines in three cores at a moment. The arrows from Time 2 to Time 3 represent the invalidate-acknowledge traffic that must occur before Core 1 can update the shared state.



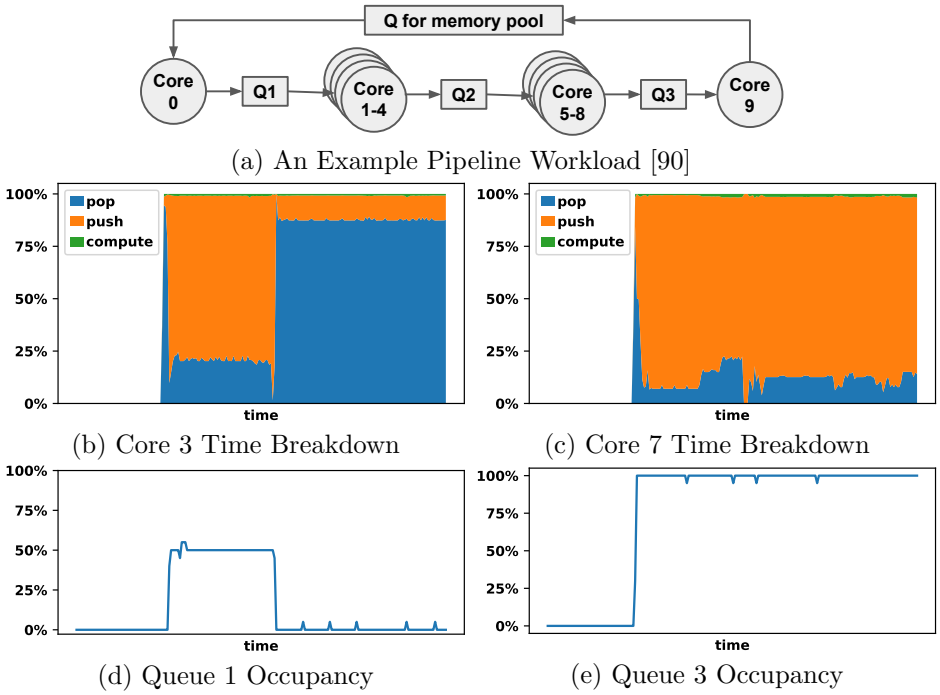
(a) consumer-driven on-demand

(b) producer-driven speculation

Figure 1.5: Two styles of message delivery. Speculation is able to hide the load-to-use latency as well as waive the requesting traffic while on-demand cannot.

across cores. There come two design options regarding how the data should be delivered: either it is driven by the consumer requests in an on-demand manner (Figure 1.5a), or the producer side could speculatively send the message (Figure 1.5b). The latter could help a consumer to reduce the time spent on requesting and waiting (i.e., load-to-use latency), and improve the overall performance especially when a busy consumer becomes the bottleneck. However, it is unclear how the data source could make accurate predictions on when and where to push the data to. Many prefetching techniques [23, 30, 50, 62, 3, 47, 48, 52, 47, 48, 52] are based merely on consumer-size information and could pollute the consumer cache with useless data. The existing producer-driven data movement speculation mechanisms [31, 86] are

software-controlled, so they are not applicable to hardware queue buffers.



(a) An Example Pipeline Workload [90]

(b) Core 3 Time Breakdown

(c) Core 7 Time Breakdown

(d) Queue 1 Occupancy

(e) Queue 3 Occupancy

Figure 1.6: An example message queue task parallel workload, pipeline, that suffers from blocking. Cores are wasting a lot of time trying push/pop on full/empty queues.

Other than the scalability and latency issues mentioned above, message queue task parallelism could suffer from blocking due to task throughput mismatch. When there is a mismatch between the upstream arrival rate and the downstream service rate, the queue buffer will often exist as either empty, or full (buffer has to be bounded otherwise grows infinitely, exhausting limited hardware resources then performance degrades, while it is not easy to determine the optimal buffer size [14, 6]). An empty queue makes the downstream tasks starve, and a full queue throttles the upstream tasks. Both empty queues and full queues block threads from performing useful computation; wasting execution cycles with no progress made to advance the program. Figure 1.6 illustrates the blocking behavior with an example pipeline workload [90]. Based on the tracing data, Figure 1.6b, 1.6c attribute cycles into queue operations and computation, and Figure 1.6d, 1.6e visualize the queue occupancy changes over time.

When blocked on an empty queue, Core 3 spends nearly 90% of its time on popping (shade in blue). Core 7, on the other hand, is pushing to Queue 3 together with three other cores. Because Core 9 as the only consumer of Queue 3, is slower, Queue 3 quickly becomes full and blocks Core 7 from doing useful computation. This blocking issue could be overcome by a runtime scheduler. A beneficial runtime scheduler must take scheduling overheads and data locality into consideration. There have been a great number of novel scheduling techniques [29, 28, 66, 45, 93, 22, 36] tested to be effective on task parallel workloads. Message queue task parallel workloads are more or less different from the applications for which those scheduling techniques are designed, but are likely able to employ those scheduling techniques to address the blocking issue.

In short, the questions this dissertation tries to answer are: **Can we improve the scalability of message queue task parallelism on multi-core architectures? Is the data flow in message queue predictable, so that we can hide the data movement latency with speculation? Can we prevent message-driven tasks from being blocked without causing too much scheduling overhead or losing data locality?**

1.2 List of Contributions

This dissertation proposes architecture and system designs to support message queue task parallelism from the following three aspects:

1. ***Virtual-Link*: A Scalable Multi-Producer Multi-Consumer Message Queue Architecture for Cross-Core Communication**: *Virtual-Link* [94] proposes a configurable hardware queue architecture to support scalable cross-core message queue synchronization. *Virtual-Link* embeds a routing device (*VLRD*) in the cache coherence network of the multi-core processor to take over the shared queue states, which would otherwise trigger cache coherence overheads if reside in shared memory. *VLRD* also reuses the Network-on-Chip

(*NoC*) to transfer message data across core at cacheline granularity, so that threads accomplish their queue operation by merely accessing L1 private cache a majority of the time. On a set of diverse queuing benchmarks, *Virtual-Link* demonstrates speedups over software queue, ranging from 1.10× to 11.36×.

2. ***SPAMeR*: Speculative Push for Anticipated Message Requests in Multi-Core Systems**: *SPAMeR* [95] is a hareware-driven speculation mechanism hiding load-to-use latency. *SPAMeR* offers several adaptive algorithms to predict when and where the message is needed based on the queue operation history. The message is then speculatively delivered by *SPAMeR* in order to reduce the request traffic from the consumers and overlap the data movement latency with message processing time. Adding *SPAMeR* to *Virtual-Link* obtains on average 1.33× speedup.

3. ***ARMQ*: A Locality-Aware Runtime for Message Queues in Multi-Core Multi-Task Parallel Systems**: *ARMQ* provides a runtime system to avoid message queue blocking with low scheduling overhead and better locality. *ARMQ* optimizes the scheduling at the blocking time by employing a few state-of-the-art techniques, such as userspace resource management, dependent task fusion, per-application scheduling customization and so on. *ARMQ* also avoids the buffer resizing overheads with a chunk-based ringbuffer design.

1.3 Thesis Statement

The performance of message queue task parallel workloads could be improved by scalable message queue architecture, message push speculation, and locality-aware runtime system that addresses blocking with low overhead.

1.4 Dissertation Organization

The organization of the dissertation is as follows. Chapter 2 summarizes the prior work in cross-core message queue synchronization, data movement speculation, and low-overhead task scheduling. Chapter 3 describes the system setups, workloads and tools used generate the experimental results throughout the dissertation. Chapter 4 presents a scalable multi-producer multi-consumer message queue architecture design (*Virtual-Link*). Chapter 5 explores a producer-driven data movement speculation mechanism (*SPAMeR*) to hide load-to-use latency under computation. Chapter 6 proposes a runtime system (*ARMQ*), which avoids blocking in message queue task parallelism with low scheduling overhead and achieves better data locality. Finally Chapter 7 concludes the dissertation.

Chapter 2: Related Work

2.1 Software Message Queues

To serve the concurrent access from multiple threads, software message queues would use some synchronous primitives. First option is using locks, either at coarse grain (lock for the entire queue), or at fine grain (lock for every node in the queue). Any thread needs to wrap the queue operations with additional locking steps (i.e., acquire, release). Michael and Scott et. al. [61] proposed an algorithm that uses read-modify-write atomic instructions to directly update the queue structure. Many lock-free queues (e.g., the lock-free queue in Boost library) adapts this algorithm. Lee et. al. [53] noticed the cache inefficiency when queue data are enqueued/dequeued frequently on different cores, so implemented MCRingBuffer, a single-producer single-consumer lock-free queue, to minimize the frequency of reading and writing the shared control variables. ZeroMQ [42] is another popular industry software queue solution, which provides a uniform interface over many different transport protocols (e.g., IPC, TCP) and features like asynchronous I/O. Bershad and Levy et. al. [16, 17] optimized Remote Procedure Call (RPC) to communicate "server" and "client" with pair-wise mapped memory. The pre-allocated memory region has one writer at a time, while one or multiple readers poll a flag bit until the writer marks the message is complete. The idea was then adapted to synchronize concurrent processes/threads in modern multi-core processors [12, 66], where critical optimizations are made to handle cache coherence protocol carefully: cacheline-aligned buffer, sequential writes, and polling at tail. These software message queue implementations more or less would have concurrent accesses to the shared queue states, triggering cache coherence traffic, while the architecture proposed in this dissertation reduces the number of sharers on synchronization primitives to zero.

2.2 Cross-Core Communication with Hardware Queues

Network processing processors such as TILE64 [15], and the QorIQ DPAA [69] provide channel operators or primitives to send data from *PE*-to-*PE* through *DMA* engines and dedicated *NoC*. Carbon [51] builds physical task queues in the cache hierarchy to accelerate a broader set of fine-grain task parallel applications in general. HAQu [55] introduces a new functional unit and storage per-core to accelerating local queuing operations. HAQu also backs the cached queue states with application memory, so that context switch could be handled easily without operating system changes. CAF [90] and Intel DLB [59] centralize the queue management in a device attached to coherence network in order to facilitate multi-producer multi-consumer queues. Compared to these prior works, the hardware queue architecture designed by this dissertation reuses some existing hardware resources in cache hierarchy for less cost and higher efficiency.

2.3 Data Movement Speculation

	Consumer-Driven	**Producer-Driven**
SW	compiler [50, 62, 3] helper threads [47, 48, 52]	pre-send [31] pre-push [86]
HW	prefetchers [23, 30] runahead [71, 64]	*SPAMeR* [95]

Figure 2.1: Taxonomy of data movement speculation.

Producer-Driven: Mostly, data movement is done in an on-demand manner, while with the context of queue, there have been studies on moving data from producers without consumer requests. Fotohi et. al. [31] proposed pre-send, a software

controlled data forwarding technique to reduce the cache misses and latency in multi-threaded programs. The similar software technique is also evaluated by Varoglu et. al. [86], and they also extended the cache coherence protocol with a push exclusive action to further reduce the coherence traffic.

Consumer-Driven: On the other side, many prefetching techniques help on getting the data in place earlier. Prefetching could be done by purely in software (with prefetch instructions or helper threads), or with hardware prefetchers [23, 30] usually implemented in cache. Some compilers techniques [50, 62, 3] have been invented to automate the insertion of prefetch instructions for variety of workloads. In order to avoid crowding the processor pipeline and deal with irregular memory access patterns (e.g., chasing chain, hashed index), a helper thread [47, 48, 52] could execute in parallel to the main threads and brings in the data earlier. Similarly, the runahead techniques [71, 64] extract the instructions related to long-latency memory accesses from the instruction sequence of the original program, in order to issue the precise data requests with perfect timing.

Unlike existing speculation approaches, the mechanism presented in this dissertation falls into the untrodden quadrant (producer-driven hardware speculation as Figure 2.1 shows), and the prediction is not only about where move the data to, but also when to move.

2.4 Task Scheduling

Task scheduling is a well-studied topic. Many techniques have been invented, more than a section can cover. This section discusses the related prior works that influence *ARMQ* design (in Chapter 6) the most.

Dependent Task Scheduling: *SWITCHES* [29, 28] is a light-weight runtime that optimizes the scheduling of dependent *OpenMP* [27] tasks across loops. *SWITCHES*

achieves lower scheduling overhead by identifying and dispatching "Cross-Loop-Task" to distributed schedulers at compile time. It is difficult if not impossible to transform message queue task parallel workload to be structured like *OpenMP* programs, because the number of iterations/tasks is meant to be infinite or not statically decidable for the compiler. *GRAMPS* [83, 73] invents per-stage work stealing to balance irregular pipelined streaming applications. Comparing to naive work stealing, *GRAMPS* guides the scheduling with producer-consumer information (i.e., prioritize downstream tasks to lower the memory consumption, prefer stealing upstream tasks for more locality and less starvation), and also bounds the memory footprint with fixed-size queues and back-pressure mechanism.

Userspace Resource Allocation: With a customized userspace threading library, *Shenango* [66] reallocates CPU cores every 5 µs to achieve higher CPU efficiency for datacenter workloads. In *Shenango*, a daemon process (*IOKernel*) tightly monitors the task queue of each application in order to identify which one needs more CPU cores to keep tasks finished within the latency guarantee. *Shenango* is designed for I/O heavy workloads, so *IOKernel* is also the proxy of all applications for the network packet sending/receiving.

Application-Specifc Scheduling Policy: *ghOSt* [45] emphasizes the importance to take scheduling hints from the application. With *ghOSt*, application-specific scheduling policy could be deployed without rebooting the system. Google search, for example, customizes its *ghOSt* policy to take advantage of data locality by guiding the scheduler with system topology, such as CPU Core Complexes (CCX) and Non-Uniform Memory Architecture (NUMA).

Locality-Aware Scheduling: There has been proves-of-concept for the benefits of scheduling OpenMP [27] tasks for better locality with the respect to NUMA nodes [93], and multi-GPU nodes [22]. SWITCHES [29, 28] supports adjusting task granularity for better cache locality. *SLAW* [36, 38] points out the differences on locality and memory pressure between work-first and help-first policy in work stealing,

then further proposes an adaptive policy (dynamically picks work-first or help-first), and groups worker tasks to improve locality based on the hints from programmers.

Aforementioned scheduling techniques are invented to reduce overheads as well as improve resource utilization and data locality for many task parallel workloads. Not all of them have been applied to message queue task parallel workloads. The runtime system developed by this dissertation borrows many ideas and techniques from the prior works: the dependent task concatenation [29, 28], the light-weight userspace threading [66], the application-specific topology-aware scheduling [45], and the locality optimization [36]. Some of the adaptions are tailored for message queue task parallelism.

2.5 Streaming Parallel Processing

Before the era of multi-core processor, there have been studies [84, 33] on running domain-specific streaming applications on grid-based architectures. As parallel architectures become the mainstream of modern generous-purpose processors, a few more queue-based solutions [14, 5, 60, 7, 54, 68] have been developed to enhance programmability and cache locality.

Steaming Template Libraries: *RaftLib* [14] is a template library that provides a streaming-style programming interface: user-defined computation kernels are connected to be Directed Acyclic Graph (DAG) via C++ stream operators (e.g., >>). *RaftLib* runtime takes care of many execution details for users: parallelizable computation kernels will be duplicated at the time constructing the DAG; *RaftLib* allocates single-producer single-consumer ringbuffers and enumerates multi-producer multi-consumer queues with multiplexing; *RaftLib* has the execution options to switch on features (e.g., ringbuffer dynamical resizing, threadpool) with no change on the user code. *FastFlow* [5] practices similar ideas in a layered model: the bottom layer implements cache-friendly lock-free single-producer single-consumer queues and

locality-aware threading support; the middle layer adds arbitrator threads to enable multi-producer multi-consumer queue support; the top layer defines several parallel algorithm patterns (e.g., pipeline, divide & conquer, farm, all-to-all etc.) as the building blocks for programmers to use. *WindFlow* [60] build atop *FastFlow* to provide window-based programming interface like big-data analytic engines (e.g., *Apache Storm* [7]). *TaskFlow* [43, 44] captures computation with C++ closures, and extends the DAG representation with embedding conditional tasks into the graph.

Cache-Optimized Buffer Management: With the respect to a specific use case: processing streaming network traffic at line-rate, *MCRingBuffer* [54] proposes a multi-core synchronization mechanism that is based on a lock-free, cache-efficient ringbuffer implementation. The "packet-stealing" technique in *GRAMPS* [73] applies thread cache for packets and follows Last-In-First-Out order to gain more locality hence performance on cache-based systems.

The message queue runtime developed in this dissertation shares some design considerations with these parallel streaming data processing frameworks, like reducing the parallel programming effort, avoiding locking in concurrent queue access, improving cache locality and so on. Other than the things in common, the runtime system in Chapter 6 has its own focus on minimizing overheads in scheduling and memory management.

Chapter 3: Evaluation Methodology

3.1 Systems

As the design of *Virtual-Link* and *SPAMeR* include extensions to *AArch64* architecture, they are implemented and evaluated in a full system simulator, *gem5*. The *gem5* system configurations of cores as well as memory hierarchy (including the routing device settings of *Virtual-Link* and *SPAMeR*) are listed in Table 3.1. *ARMQ* runtime is developed on two high-end *AArch64* servers with different topologies. Table 3.2 compares the most important specifications of the two systems. During *ARMQ* experiments, Dynamic Voltage Frequency Scaling (DVFS) on both systems are turned off to ensure all CPU cores are running at their maximum speed.

Table 3.1: *gem5* Simulator Hardware Configuration.

Cores	16×*AArch64* OoO CPU @ 2 GHz
Caches	32 KiB private 2-way L1D, 48 KiB private 3-way L1I 1 MiB shared 16-way mostly-inclusive L2
DRAM	8 GiB 2400 MHz DDR4
VLRD	64 entries per *prodBuf*, *consBuf*, and *linkTab*
SRD	in addition to *VLRD*, 64 entries per *specBuf*

3.2 Benchmarks

Table 3.3 lists the benchmarks used in the experiments. *bitonic* is a sorting

Table 3.2: Specifications of Two *AArch64* Servers

Core-Coupled System A	
Cores	32× ARMv8 X-Gene CPU @ 3.3 GHz
Caches	32 KiB 8-way L1D, 32 KiB 8-way L1I 256 KiB 32-way L2 per 2 cores 32 MiB shared SLC
DRAM	128 GiB 2666 MT/s DDR4
kernel	`Linux 5.4.0-80-generic`
compile	`g++ 10.3.0, -O3, -ltcmalloc_minimal`
Two-Socket System B	
Cores	2 sockets × 80× ARMv8.2+ Neoverse-N1 @ 3.0 GHz
Caches	64 KiB 4-way L1D, 64 KiB 4-way L1I 1 MiB 8-way private L2 32 MiB 16-way mostly-exclusive SLC per socket
DRAM	256 GiB 3200 MT/s DDR4, 2 NUMA nodes
kernel	`Linux 5.14.0-69-generic`
compile	`g++ 11.3.0, -O3, -ltcmalloc_minimal`

algorithm with plenty of parallelism for hardware to exploit. *ping-pong*, *halo*, *sweep*, and *incast* are the common communication patterns that Ember benchmark suite [80] derives from high-performance computing workloads. *pipeline*, and *firewall* represent the styles of many network packet processing workloads [90]. *FIR* from digital signal processing is an important way processing streaming data. *outcast* is another common communication pattern in message queue parallel workloads. *chasing* does extensive chasing pointer style access (a well-known challenging memory access pattern) on the passing message buffer. *search* dispatches file chunks to perform word search in parallel then aggregates the results [14]. *tc*, *dc*, and *bc* are the graph analytic

Table 3.3: Benchmarks.

Benchmark	**Description, (#producer:#consumer) × #queue**
bitonic [11]	bitonic sort with varying number of threads (1:N)×1+(M:1)×1
ping-pong [80]	data back and forth between two threads (1:1)×2
halo [80]	exchange data with neighboring threads (1:1)×48
sweep [80]	data sweeps through a grid of threads corner to corner (1:1)×48
incast [80]	all threads sending data to the master thread (M:1)×1
pipeline [90]	4-stage pipeline with middle stages multi-threaded (1:4)×1+(4:4)×1+(4:1)×1+ (1:1)×1
firewall [90]	filter and dispatch packages (1:1)×3+(2:1)×1
FIR	data streams through N-stage FIR filter (1:1)×N
outcast	all threads receiving data from the master thread, (1:N)×1
chasing	pointer-chasing buffer access pattern on messages passed filter, (1:4)×1+(4:1)×1+(1:1)× Q
search [14]	search a given word in a file, (1:4)×1+(4:1)×1
tc [63]	triangle count on a graph, (1:1)×1+(1:4)×1+(4:1)×1
dc [63]	degree count on a graph, (1:4)×1
bc [63]	calculate betweeness centrality of a graph, (1:4)×1+(4:1)×1
STREAM [58]	issues many memory accesses to measure memory bandwidth

benchmarks from GraphBIG [63]. The graph computing tasks are mainly divided by vertices as well as steps in the algorithms (e.g., searching within an adjacency list, counting intersections of two lists, and accumulating the triangle counts as a vertex property). As marked at the end of each row in Table 3.3, (M:N)×k denotes the number of producers (M) and consumers (N) as well as the number of queues instance (k) in each benchmarks. The set of benchmarks cover all message queue types (i.e., four combinations of single/multiple producer/consumer), and have varying number of queues. *STREAM* benchmark [58] measures memory bandwidth with intensive

memory accesses, so the bus stress experiment in Chapter 4 runs it along with other benchmarks. Because the experiments in Chapter 4, 5 are simulation-based and very time-consuming, only the short benchmarks (from *bitonic* to *FIR*) are used. When used in the experiments of Chapter 6, those short benchmarks (except the first four) take more iterations and run for longer duration, in order to reduce the system noise impact. Because *ARMQ* requires the applications to be represented as Directed Acyclic Graph (DAG), *bitonic*, *ping-pong*, *halo*, and *sweep* are not used in the experiments of Chapter 6.

3.3 Metrics and Tools

The simulation experiments in Chapter 4, 5 use the statistics reported by *gem5* full system simulator [20]. The primary performance metric is the execution time of Region of Interest (ROI). The unit is tick, and is sometime converted to cycles, millisecond when presented. *Virtual-Link* and *SPAMeR* work inside the cache hierarchy, so their evaluations look at a few statistics in the memory system for in-depth analysis, such as snoop traffic, memory transactions, and coherence bus utilization and so on.

The area and energy estimations in Chapter 4, 5 use the Synopsys Design Compiler with the FreePDK 45 nm library [82] to synthesize the functional RTL code, then scale the design to 16 nm [81] for comparison.

The experiments in Chapter 6 are conducted on real machines (Table 3.2), so the execution time of ROI is measured by `high_resolution_clock::now` function from `std::chrono` library. As the data locality improved by *ARMQ* should affect the performance of cache, *Linux perf* [67] is used to measure the cache misses.

Chapter 4: *Virtual-Link*: A Scalable Multi-Producer Multi-Consumer Message Queue Architecture in Multi-Core Systems

4.1 *Virtual-Link* Architecture Design

Virtual-Link (*VL*) accomplishes the movement of cache lines from producers to consumers by attaching a routing device (*VLRD*) to the coherence network as illustrated in Figure 4.1. The *VLRD* is attached to the coherence network like a tightly coupled accelerator or system cache slice, from a port on the coherence network.

This *VLRD* enables *VL* to "link" unique "endpoints" together via a shared queue identifier (*SQI*). Endpoints subscribe to a *SQI* to form a *M:N* message channel Each *SQI* can support *M* producer endpoints and *N* consumer endpoints. Each unique endpoint for a *SQI* maintains its own local user-space buffer composed of multiple coherence granules or cache lines. Messages from each endpoint are received by the *VLRD* at a coherence granularity, in a lock-free manner. In abstract, ***VL* enables a virtual linking of cache lines from each unique endpoint subscribing to a *SQI* so that a producer can copy-over data from its own cache lines directly into a requesting consumer through a single level of indirection.**

Multiple endpoints on a single *SQI* come together to form a Virtual Queue (*VQ*). Figure 4.2 illustrates the ordering of operations between two producer endpoints and a consumer endpoint sharing a *SQI*. The *VQ* size is shown after each time step. In Figure 4.2, the cache lines are moved atomically, that is at time step 2, the blue producer cache line data appears to be copied-over atomically (through the interconnect, not main memory) to the consumer endpoint buffer. This copy-over operation leaves the producer cache line zeroed and in an exclusive state, which can be used for subsequent enqueue operations. After the copy-over operation, the data

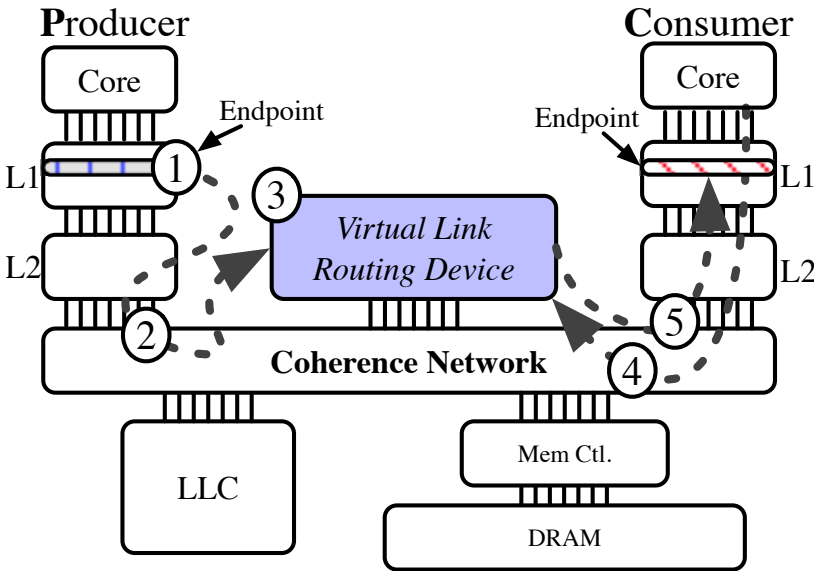


Figure 4.1: *Virtual-Link* architecture high-level view. A cache line moves from the producer at (**1**), at its own unique address location, to an indirection layer in hardware at (**2**). That indirection layer, the Routing Device, matches the *SQI* at (**3**) based on consumer endpoint demand which is registered by (**4**). The Routing Device forwards data to the target consumer buffer on a totally different address at (**5**).

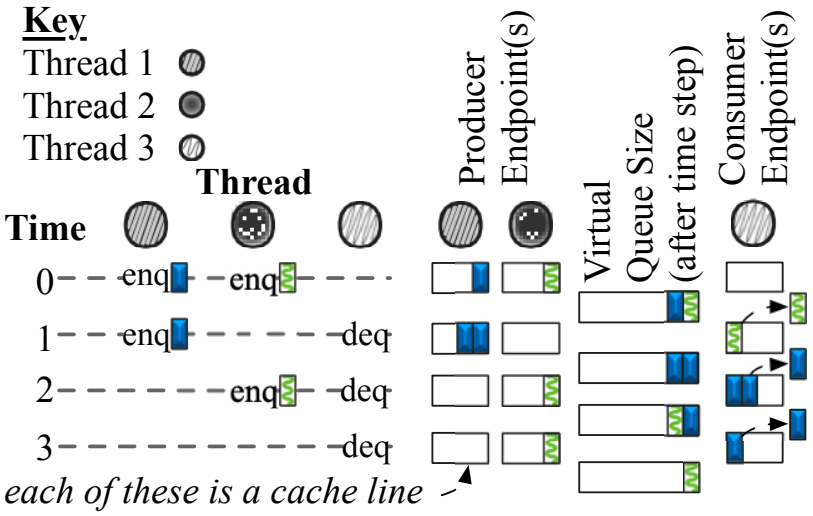


Figure 4.2: Virtual Queue (VQ) per time step. 2 producer endpoints (Threads 1, 2), 1 consumer endpoint (Thread 3), shares a SQI.

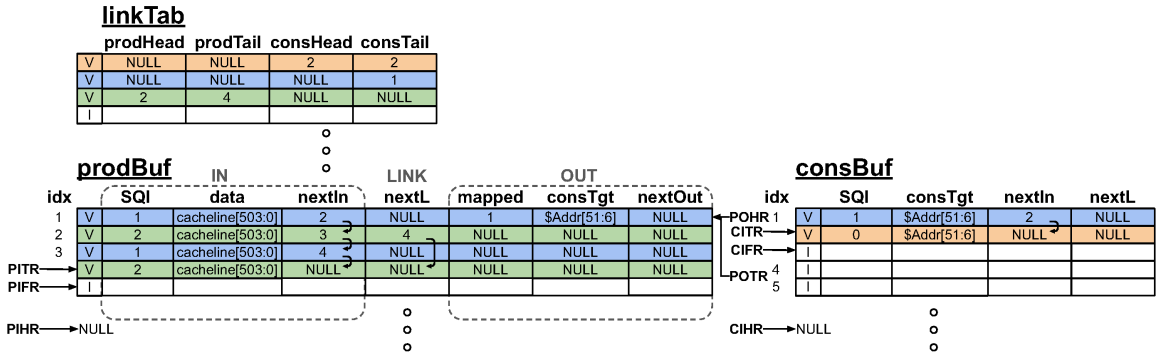


Figure 4.3: Table and buffer structures in the $VLRD$. Cells having the same background color belong to the same SQI.

are shipped to the consumer to dequeue, also in an exclusive state. At no point does the consumer or producer access a shared Physical Address (PA) or Virtual Address (VA) that could cause coherence traffic (*snoops*). Instead, threads check the endpoints owned by themselves and interact with the $VLRD$ for synchronization. The rest of this section presents the major components of VL, namely, the $VLRD$, ISA extensions and system software support.

Table 4.1: Address mapping pipeline actions per cycle. $xxx_n \triangleq$ the latch for xxx in Stage n

C.	**Stage 1** reads *linkTab*	**Stage 2** makes mapping decision	**Stage 3** updates tables and buffers
1	$prodHead_1$, $consTail_1 \leftarrow$ NULL, NULL /* linkTab[consBuf[1].linkId], CIHR $\leftarrow$ 2 */		
2	$prodHead_1$, $consTail_1 \leftarrow$ NULL, NULL /* linkTab[consBuf[2].linkId], CIHR $\leftarrow$ NULL */	miss: append to the linked list in consBuf /* because $prodHead_1$=NULL, no blue data */	
3	$consHead_1$, $prodTail_1 \leftarrow$ 1, 1 /* RAW */ /* linkTab[prodBuf[1].linkId], PIHR $\leftarrow$ 2 */	miss: append to the linked list in consBuf /* because $prodHead_1$=NULL, no orange data */	linkTab[1].cons{Head, Tail} $\leftarrow$ 1, 1 /* $linkId_2$=1, $CIHR_2$=1, new consHead read by Stage 1 */
4	$consHead_1$, $prodTail_1 \leftarrow$ NULL, NULL /* linkTab[prodBuf[2].linkId], PIHR $\leftarrow$ 3 */	hit: read consBuf[1] for consTgt, nextL /* $consHead_1$=1 */	linkTab[0].cons{Head, Tail} $\leftarrow$ 2, 2 /* $linkId_2$=0, $CIHR_2$=2 */
5	$consHead_1 \leftarrow$ NULL /*$nextL_2$ forwarded*/ $prodTail_1 \leftarrow$NULL /*linkTab[prodBuf[3].linkId]*/	miss: append to the linked list in prodBuf /* because $consHead_1$=NULL, no green request */	linkTab[1].consHead $\leftarrow$ NULL /* $nextL_2$ */ set prodBuf[1].OUT POHR, POTR $\leftarrow$ 1, 1

4.1.1 *Virtual-Link* Routing Device

The *VLRD* is tasked with matching incoming messages to a *SQI* and stashing those messages to the subscribed consumers. As Figure 4.3 shows, the *VLRD* is largely composed of three structures, the Link Table (*linkTab*), the Producer Buffer (*prodBuf*), and the Consumer Buffer (*consBuf*) (some control logic is omitted for brevity). The *linkTab* keeps metadata (i.e., head, tail) for each *SQI*, one per row. The *prodBuf* and *consBuf* are shared across multiple *SQI* entries, and buffer producer data and consumer requests, respectively. Buffer slots are taken in turn and shared by multiple *SQI*s, therefore these structures cannot be used as contiguous FIFOs but instead are managed as linked-lists (*LL*)s.

linkTab: The head/tail pointers in *linkTab* each point to the first and last entries in a hardware-managed inter-leaved *LL* data structure, which enables hardware to determine whether there is consumer demand on a specific *SQI* or data available from a producer to send. The producer head (`prodHead`) is updated if the current head is mapped and ready to be sent to a consumer. For example in Figure 4.3 the

green row, `prodHead` points to index 2 (Row 2 in *prodBuf*). Once index 2 is mapped with a green consumer request coming later, `prodHead` is set to the next green entry (4 in this example). The *linkTab* is addressed by the *SQI* field in *prodBuf* and *consBuf*.

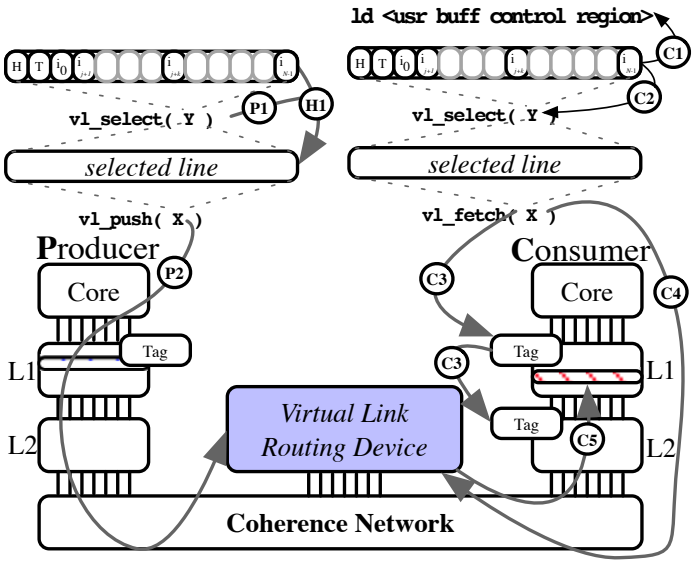
consBuf: Whenever a consumer request arrives in the *VLRD*, the port's control logic checks Consumer Input Free Register (*CIFR*) for a free buffer slot in order to buffer the consumer request. A buffer slot is free if the valid bit is unset, and *CIFR* always moves to the next free slot after a slot is taken, starting over from the first free *consBuf* slot again after touching the bottom. The consumer request is composed of two parts: 1) the address of the target consumer cache line (the local user-space buffer of a consumer endpoint) buffered in `consTgt` as shown in Figure 4.3; and 2) the *SQI* of the *VQ* from which data is requested. The former is the payload of the incoming packet, and latter is encoded in the device-memory physical address received through the coherence network (details in § 4.1.3.2). The `nextL` field together with the `consHead`, `consTail` in *linkTab* make *LL*s for *SQI*s. As mentioned before, the slots in *consBuf* is not always used in order when multiple *SQI*s are active. The `nextIn` field together with Consumer Input Head Register (*CIHR*) and Consumer Input Tail Register (*CITR*) forming a *LL*, so that *consBuf* can track the order to feed the address mapping pipeline. Address mapping pipeline stages are illustrated in Table 4.1 (explained later).

prodBuf: The Producer Buffer has three partitions, namely, **IN**, **LINK**, **OUT** as shown in Figure 4.3. On cache line arrival to the *VLRD*, the Producer Input Free Register (*PIFR*) is checked for a free buffer entry. The Producer Input Head Register (*PIHR*), and Producer Input Tail Register (*PITR*) point to the next, and the last buffered producer push waiting for address mapping, respectively. The **IN** partition plus the **LINK** partition are very similar to the *consBuf*, except that the `data` field stores the data enqueued by producers (§ 4.1.4). The **LINK** partition is a *LL* whose head is the oldest entry ready to be sent to a consumer; the order in which producer data was received is tracked by the *LL*; so data are sent to consumers in the same order. The **OUT** partition is for registering mapped entries, i.e., entries that have

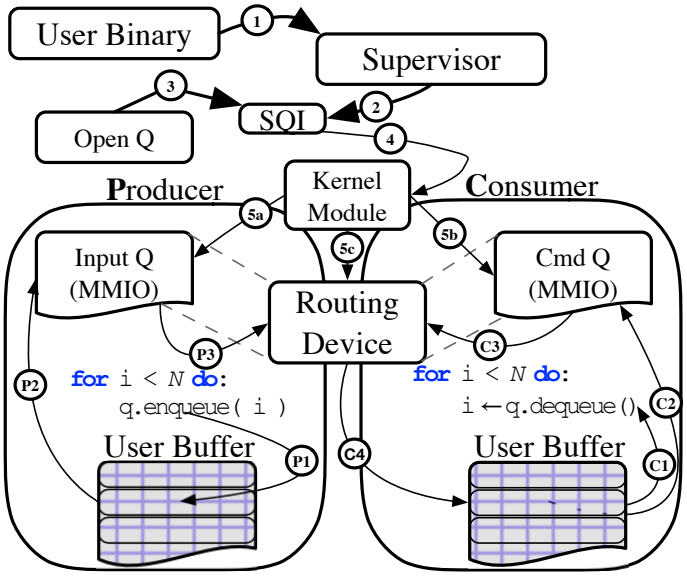
been assigned to a consumer target from the process in Table 4.1. For example, the first blue entry in the *prodBuf* is mapped to *consBuf* entry 1 as indicated by Figure 4.3. The `consTgt` field in the **OUT** partition stores the result of address mapping (i.e., a target consumer cache line address), and `mapped` field recording an index to the mapped *consBuf* slot. There are also two registers associated with this partition, the Producer Output Head Register (*POHR*), and the Producer Output Tail Register (*POTR*) to track the next, and the last entry ready to send out, respectively. Each of the three partitions is a separate SRAM block with its own read/write ports, making each partition accessed independently.

Address mapping: A *prodBuf* entry with valid data or a *consBuf* entry occupied by a consumer request will go through a 3-stage pipeline illustrated in Table 4.1, to map a producer push with a consumer pull. At the first stage the control logic takes *SQI* from the "head entry" (the entry pointed by either *PIHR* or *CIHR*) to access the *linkTab* and get the head, tail pointers of a corresponding queue. In Stage 2, a decision is made on whether to map the "head entry" to a consumer request or producer data buffered earlier. For example, in Cycle 1 the first blue consumer request reads blue `prodHead`, which is then checked in Cycle 2 Stage 2 in Table 4.1. The blue request has to append to blue consumer *LL* upon a *miss*. A *hit* occurs in Cycle 4 Stage 2, when a blue data enters the pipeline and hits the blue consumer request. The third stage performs writes, updating table and buffers according to the mapping decision.

There are a few trade-offs making the *VLRD* design simpler or more complex: 1) The multiple buffer partitions decouple the address mapping pipeline and bus I/O, so a burst of packets can be buffered first then fed into the pipeline, otherwise the *VLRD* just accepts one packet per clock cycle; 2) *LL* is chosen over a bitvector to deal with the sparse buffer entry usage, that is not only due to the consideration of FIFO property, but also because the authors feel *LL* is more scalable for large *VLRD*s. Additional trade-offs are discussed in § 4.1.3.2.



(a) Hardware view



(b) Software view

Figure 4.4: Flow of *VL* hardware, *ISA* and software interaction. Details in § 4.1.2 to § 4.1.4.

4.1.2 Instruction Set Extensions

To allow software to express the role of producer/consumer explicitly, *VL* adds three new instructions for `vl_select`, `vl_push` and `vl_fetch` operations. Technically they are "data cache" maintenance instructions with a `dc` nomenclature; we simply refer to them by their named function.

`vl_select Rt`: The `vl_select` identifies a specific cache line by a *VA* in the operand register `Rt`. As the name suggests, `vl_select` "selects" a cache line addressed by *VA*, so that a follow-on `vl_push` or `vl_fetch` instruction can perform its operation on the "selected" cache line. Through `vl_select`, the *VA* of the cache line is translated, and the *PA* gets latched into a system register (not part of context state) only accessible by `vl_push` or `vl_fetch`. Similar to load-linked store-conditional (*LLSC*), where a load-link always precedes a store-conditional, there is a dependency between a `vl_select` instruction and a `vl_push` or `vl_fetch` instruction, although `vl_fetch` itself can be executed speculatively and out-of-order with respect to instructions other than `vl_push` or `vl_fetch`. In the case the cache line to select has been evicted into memory, `vl_select` generates a cache miss and brings the cache line back to L1 data cache (*L1D*), just as any store would, in an "exclusive" cache state. On context swap or page migration, the latched *PA* is cleared.

`vl_push Rs, Rt`: The `vl_push` instruction takes the cache line from `vl_select` and conditionally writes it from cacheable memory to a *VLRD* memory target `Rt` (provided as a *VA*). This *VA* in `Rt` is assigned to the *VLRD* by the scheme described in § 4.1.3.2. The operand register `Rs` receives the result of zero for success or nonzero upon failure of a `vl_push` operation. On completion, the selection of the cache line ends (i.e., *PA* in the system register set by `vl_select` is zeroed). There are a few scenarios the `vl_push` operation could fail. First, a `vl_push` being called without a previous `vl_select` call results in a non-zero value written back to `Rs`. The second, is the most expected failure case where the *VLRD* has no buffering capacity or consumer demand which also returns a non-zero to `Rs`. A system register counting

`vl_push` instructions on-the-fly ensures no context swap or interrupt can occur before a `Rs` receives a result. The *VLRD* must make forward progress in a fixed interval, i.e. bounded by the time it takes to get to the *VLRD*, which is approximately 14 cycles in our implementation. `vl_push` is a device memory write on the coherence network, as such, the write is non-snooping and it cannot be merged with other writes.

`vl_fetch Rs, Rt`: The `vl_fetch` has the effect of pulling data from a *VLRD* memory location (the *VA* from `Rt`) into the calling core's private cache at the location specified by the paired `vl_select` call. Like `vl_push`, `vl_fetch` clears cache line selection on execution. If data is available on a given *SQI* (see § 4.1.3.2 for *VA* to *VLRD* and *SQI* mapping), then the *VLRD* sends a data injection to the user buffer location specified by `vl_select` immediately. If data is not available, the request is conditionally registered with the *VLRD*, conditional on buffering capacity for requests in the *VLRD*. A successful request results in a zero value being stored in `Rs`. Once data is available for the requested *SQI* then data is conditionally injected. `vl_fetch` sets a "pushable" bit within the calling core's private caches, this facilitates asynchronous (and speculative) conditional data injection by the *VLRD* while ensuring data still in-use is not overwritten by the *VLRD*. If there is a context swap, thread migration following a `vl_fetch`, or the line is evicted, the injection attempt is rejected, because the "pushable" cache flag is unset before any of those scenarios occur, and the data remain with the *VLRD*. The system register set by `vl_select` is cleared by `vl_fetch` as well. On being scheduled the programmer is expected to check the line to see if new data has arrived (e.g. examine control region from § 4.1.4), to re-issue the request which sets the cache tag as "pusheable" again.

The *ISA* described adds a single bit to the cache tag array of each private cache, and adds conditional write and push commands to support the signalling. *VL* uses an otherwise standard coherence network with non-snooping directed data transfer, the width of that network remains unchanged.

4.1.3 User-space and System Software

Using an existing queuing framework such as *BLFQ* or *ZMQ* with *VL* is simply a matter of mapping the *ISA* from § 4.1.2 corresponding to enqueue/dequeue semantics to the existing software queue application program interface. There are a few additional allocation constraints, such as specific alignment requirements and *VLRD* setup. Hence, we develop a library to ease the programmer burden.

In Figure 4.4b, a user binary starts by requesting a *SQI* (equivalent to a file handle) at (**1**) and (**2**). At (**3**) the programmer maps this *SQI* into a process accessible *VA* through a system call at (**4**), that sets up the *VLRD* with the *SQI* at (**5c**) and returns a mapped *VA* into user-space at (**5a**) and (**5b**).

4.1.3.1 *SQI* allocation & release

M:N endpoints assigned to a *SQI* are allowed to communicate. This is akin to "shared memory" Inter-Process-Communication (*IPC*) with the *SQI* being analogous to a file descriptor and following similar rules with similar supervisor/OS protections [79]. The *SQI* can be used to open endpoints from user-space, granting the calling thread access to map this *SQI* channel into its address space. Listing 1 is what is executed at (**1**) of Figure 4.4b, resulting in the *SQI* at (**2**). *SQI* closing and ordering semantics are identical to those of "shared" memory POSIX file handles, simplifying the programming interface.

Listing 1 Example of calling a POSIX compliant `shm_open` with the string handle "queue_name", with a read and write mode, and a `VL_QUEUE` flag that tells the supervisor that this is to be a *VL* shared memory operation.

```
const int SQI = shm_open( "queue_name",
                          O_RDWR,
                          VL_QUEUE /** flag for queue **/ )
```

4.1.3.2 Endpoint creation

As shown in Figure 4.4b, once a *SQI* is obtained, the programmer must "open" the queue (**3**) then map that descriptor to a *VA* to address the assigned *VLRD*. This *SQI* is mapped to a *VA* using `mmap` [79] (via a kernel module wrapper at (**5a**) and (**5b**)) as shown in Code snippet 2 using the addressing scheme described shortly.

Listing 2 Example of obtaining a *VA* mapping for the *SQI* from user-space. The *VA* returned is to a device memory location which maps the *VA* to the *PA* of the *VLRD*.

```
void *X = mmap( nullptr,
                QPAGE_SIZE,
                PROT,
                VL_QUEUE /** flag for queue **/,
                SQI,
                0x0 )
```

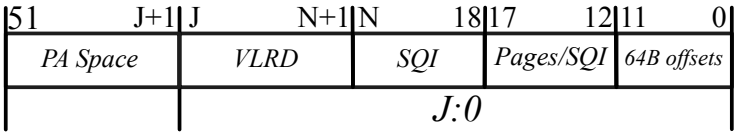


Figure 4.5: Device-memory physical address bit fields addressing the *VLRD*

A user-space library can subdivide the device-memory-mapped *VA* page further to make multiple non-overlapping (64 B-aligned) addresses for the same *SQI* within a single address space. Our implementation maps a 4 KiB page to each page-aligned *MMIO* address on the *VLRD*. A bit-vector within the user-space wrapper around `mmap` is maintained to quickly find an unused, 64 B-aligned offset to return. If `PROT_WRITE` is given the library call returns a producer page mapping, likewise if `PROT_READ` is given, a consumer page returned. Removing a user-space *VA* mapping for an endpoint is through the `munmap` command [79].

The allocated endpoint *VA* from `mmap` is the means by which `vl_push` is able to target the *VLRD*, and the *PA* (translated from the *VA*) is the means by which the *VLRD* can determine the *SQI*. Figure 4.5, describes the bit fields of the *PA* with *VL* information encoded. A *VLRD* simply takes $N:18$ as the *SQI*, while bits $J:N+1$

could distinguish different *VLRD*s if more than one *VLRD* are implemented to serve different VQs independently. Multiple pages may be used, e.g. to map into differing address spaces, or more than 64 endpoints are needed. This is what bits 17 : 12 used for, allowing up to thirty two 4 KiB pages. This memory mapping process is repeated for the consumer endpoints. A downside of this process is physical address space is used, e.g. with 1-*VLRD*, and 16-*SQI*s then $N \leftarrow 22$ and $J \leftarrow 26$ which would use up 67 MiB of address space (not physical memory). An alternative addressing scheme that we explored adds an address table to the *VLRD* (populated on `mmap`) to map to arbitrary addresses, however, at the cost of an extra cycle to the pipeline § 4.1.1 and content addressable memory for the routing table.

4.1.3.3 User-space buffer creation

VL enables both producers and consumers to use any page-aligned cacheable memory as the user-space buffer for local endpoints (e.g. the data source at (**1**) from Figure 4.1). The memory could be obtained from any generic memory allocation functions (e.g. `posix_memalign`). The capacity of these buffers can be adjusted in user-space without impacting *VL* to accommodate bursty behavior or non-stationary queue traffic distributions. It is these user-space memory buffers that are used in subsequent *enqueue* and *dequeue* operations (§ 4.1.4). The user-space buffer for each endpoint is used as a circular buffer for sending lines to the *VLRD*, as such it will typically be kept cache-local. Once a line from the user-space buffer is pushed to the *VLRD*, it is marked as cleaned, (e.g. reset control region as discribed in § 4.1.4), so that it is ready for follow-on *enqueue* operations.

4.1.4 Enqueue and dequeue

Figure 4.2 shows the queue order per single *SQI* atomically pushing a 64 B cache line size messages from *M:N* producer/consumer pairs. Messages larger than

a cache line can be incorporated via indirect buffers as pointers. While not demonstrated in this dissertation, it is trivial to incorporate an existing indirect buffer format such as VirtIO 1.1 [87], injection could be accelerated in this case by [8]. To facilitate small message transfer, we embed cache line local queue state into the line itself (see Figure 4.6). This consists of a 2 B control region at the Most Significant Byte (*MSB*) of each *VL* transported cache line. the remaining 62 B are user-data/payload. Valid data fills the data region from higher address towards Least Significant Byte (*LSB*). Within the control region, 2b encodes for size, e.g., byte, half word, word, double word. 6b encodes a cache line relative offset/head pointer. The remaining 1 B is reserved.

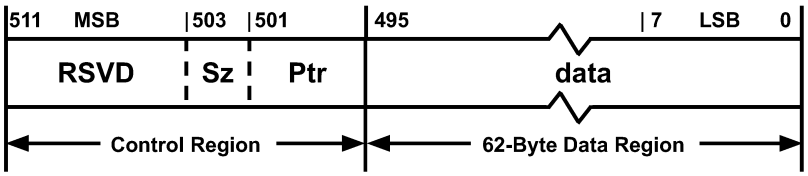


Figure 4.6: Control region and data region in a 64 B cache line.

enqueue: With respect to Figure 4.4a, the enqueue operation calls `vl_select` at (**P1**) on an allocated user-space buffer (Y). The user-space cacheable memory transitions to a "selected" state at (**H1**) that causes this cache line's *VA* to be translated and latched. The follow-on `vl_push` instruction at (**P2**) causes the cache line at the aforementioned latched *PA* from `vl_select` (Y) to be stored to the mapped *VLRD* device-memory address (X). Assuming the conditional store was successful, the original cache-able user-space memory from Y is owned by the *VLRD*. This order of events is necessary to prevent a single instruction from requiring two address generations simultaneously. If the enqueue succeeds, the cache line is zeroed, otherwise the return register (see § 4.1.2) is set appropriately so that the programmer can retry pushing the same data at some future point.

dequeue: Dequeue operations for *VL* are essentially operations that set a cache line as "pushable" while also notifying the *VLRD* that 64 B of data is requested at a

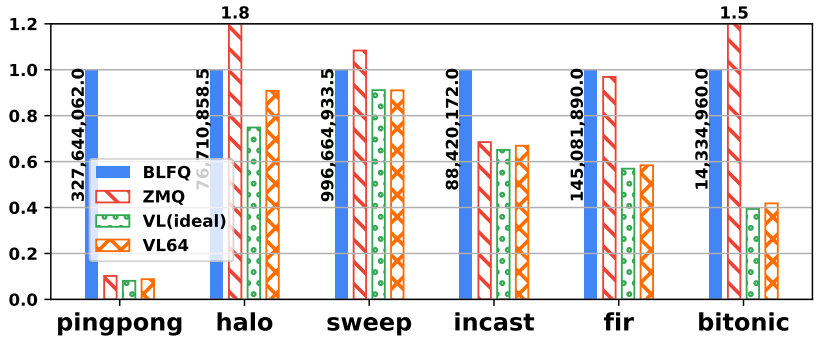
specific cacheable-memory *VA*. With respect to Figure 4.4a, the dequeue operation calls `vl_select` at (**C2**) on an allocated user-space consumer buffer, after determining at (**C1**) that no more data is available (e.g. by inspecting the control region). Calling `vl_select` at (**C2**) sets that *VA* and latches the *PA* of that line for a follow-on `vl_fetch` instruction (§ 4.1.2). As described in § 4.1.2, `vl_fetch` sets a "pushable" flag at (**C3**) for the cache line addressed by the previous `vl_select` statement. Following the setting of the "pushable" flag, `vl_fetch` causes the target *PA* and core-id to be registered with the *VLRD* at (**C4**). That registered *PA* is used when data becomes available for a given *SQI* for a follow-on injection of data to the requester at (**C5**).

4.2 Results and Analysis

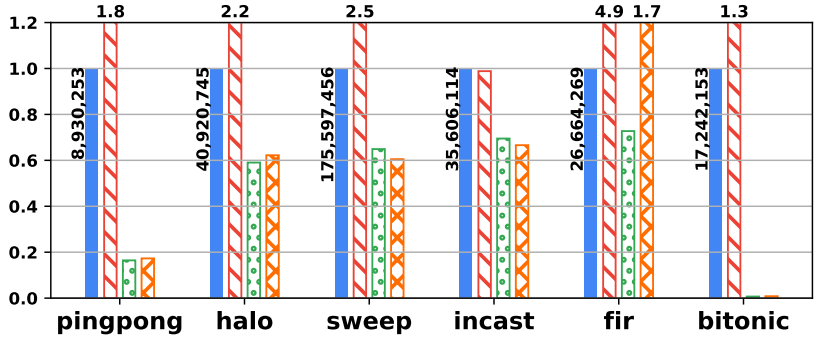
4.2.1 Performance, Snoop, and Memory Transactions

In Figure 4.7, we compare *VL* with two state-of-the-art software queues, *BLFQ* as baseline and *ZMQ*. In addition to this, we add *VL*(ideal) which has **infinity** queue capacity and **zero-latency** cache line transfers in order to show that those hardware limitations do not put much overhead on *VL*. Each *VL* run is given with 64 buffer entries, and denoted as *VL*64.

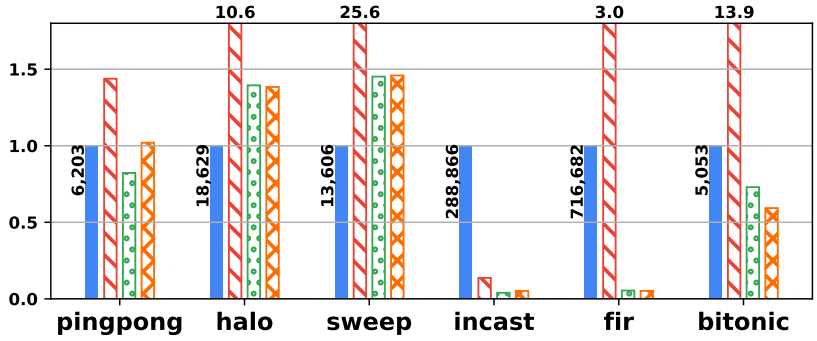
In Figure 4.7a we see that *VL* is on average 2.09× faster than software solutions, ranging from 11.36× faster for *ping-pong* to 1.10× faster for *sweep*. *ZMQ* falls somewhere in between on all benchmarks, though notably being slower on *halo* and *bitonic*, which both favor low-latency small message traffic. However, on *incast* and *FIR*, *BLFQ* builds up a long queue spilling to memory (many more memory transactions in Figure 4.7c), *ZMQ* and *VL* both have a back-pressure mechanism so get better performance. Figure 4.7b shows the relative magnitude of snoop transactions initiated per benchmark and with queue schemes. *VL* has fewer snoops than either of the two software queues (*BLFQ* and *ZMQ*). The only exception *FIR* has two threads per core creating many context switches, which lead to more frequent failures for VLRD's attempts to deliver cache lines. Software queues suffer from more snoop



(a) Execution time (ns) normalized to *BLFQ*



(b) Snoop traffic normalized to *BLFQ*



(c) Memory transactions normalized to *BLFQ*

Figure 4.7: Execution time, snoop traffic, and memory transaction comparison between software queues and *Virtual-Link*.

transactions due to cache coherence, while *Virtual-Link* reduces the snoop traffic to a minimum as it reduces the cache coherent state shared between communicating threads. Figure 4.7c compares the amount of memory transactions between queues. Overall, *VL* has the fewest memory transactions among the queuing schemes. *VL* and *ZMQ* are significantly lower on *incast* and *FIR* with the help of the back-pressure mechanism. On *ping-pong* and *bitonic*, *VL* also achieves about 20% reduction compared to *BLFQ*, while *ZMQ* has more memory transactions. *VL* has more memory transactions on *halo*, and *sweep*, because the benchmarks double buffer the communication channels and not all the buffers are managed by our provided queuing libraries, but by the application.

4.2.2 Scalability

Bitonic has a fixed workload divided among a varying number of worker threads. Figure 4.8 presents the scalability of *bitonic* with various queue implementations as the number of worker threads are changed (1, 3, 7, and 15 worker threads plus one master thread dispatching tasks to worker threads). Initially, *ZMQ* performs better than *BLFQ* with small numbers of threads (i.e., 2, 4), but *ZMQ*'s performance drops after 8 threads. The high overhead to maintain cache coherence (as shown in Figure 4.9) degrades the performance of *ZMQ*. Because *BLFQ* does *CAS* operations, it scales slightly better than *ZMQ*, however, neither scale as well as *VL*. *BLFQ* stops scaling by 4 threads. In contrast, *VL* is still able to gain speedup moving from 4 threads to 8 threads. At 8 threads, the computation part of the single master thread dominates the execution time and become the bottleneck; that is why none of the queuing mechanisms can help any more. In Figure 4.9, we present one big difference between *VL* and the other software queues at a microarchitecture level, to better understand why they scale differently. Both the *BLFQ* and *ZMQ* software implementations have more cache line upgrade events than *VL*, and the rate of snoop traffic synchronization goes more rapidly. *VL* has very few upgrades and snoops, therefore it is able to scale better than *BLFQ* and *ZMQ*.

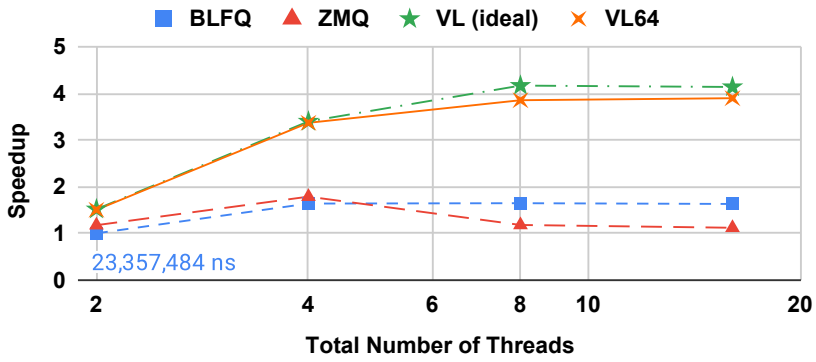


Figure 4.8: Scalability comparison between software queues and *Virtual-Link*. As the number of worker threads in *bitonic* is scaled from 2 to 16, *Virtual-Link* shows better scalability.

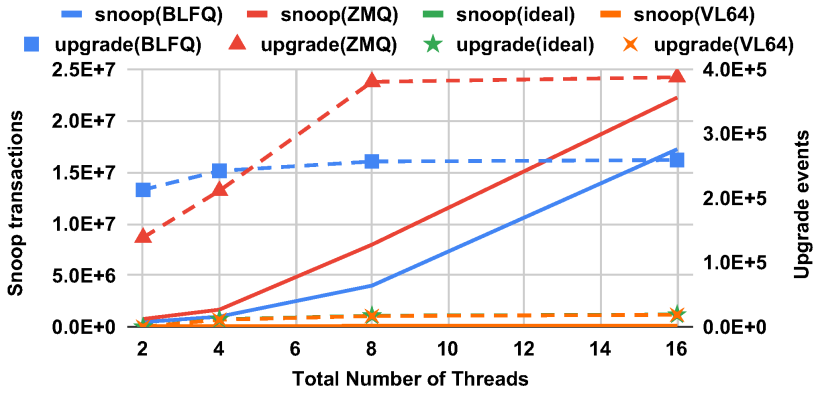


Figure 4.9: Snoop and cache line upgrade events comparison between software queues and *Virtual-Link* as the number of threads scales. *Virtual-Link* keeps snoop and update events at low level when the number number of *bitonic* worker threads is increased from 2 to 16, while software queues have high snoop and significantly increasing upgrade events.

4.2.3 Coherence Traffic Interference

VL channels use the coherence network to move data between cores. This could impact the coherence traffic patterns and hurt the performance of other applications that do not use *VL*. To study the impact, we ran the *STREAM* benchmark [58] concurrently with *ping-pong* using each queue implementation (*BLFQ*, *ZMQ*, *VL*). *STREAM* was chosen as it is known to stress the memory hierarchy. Figure 4.10 shows that the execution time for each queue implementation (*BLFQ*, *ZMQ*, *VL*)

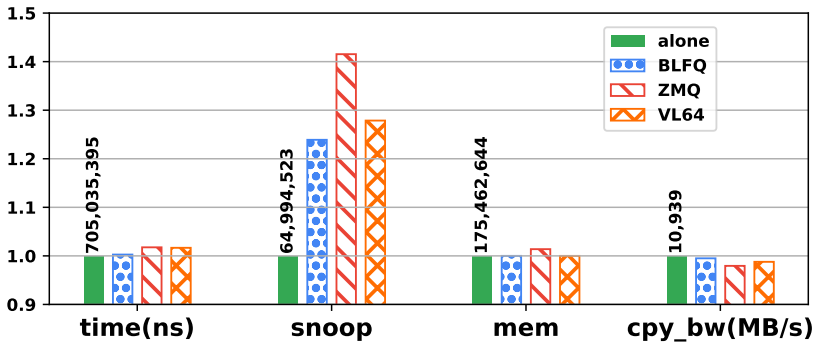


Figure 4.10: Performance impact on co-located memory-intensive workloads. A memory intensive benchmark (*STREAM* [58]) measures memory bandwidth standalone and with background queue workload.

varied by 2% or less when compared to *STREAM* executing alone. The other three bar groups report the system snoop and memory traffic. The snoop traffic introduced by *VL* is comparable to that of *BLFQ*, and significantly lower than that of *ZMQ*.

Area estimation: We developed RTL code for the *VLRD* (control logic + buffers), synthesized it using the Synopsys Design Compiler with the FreePDK 45 nm library [82], and scaled the design to 16 nm [81] for comparison. The resulting *VLRD* area is 0.142 mm^2 for buffers and 0.155 mm^2 in total including control logic. To put this into perspective, an Arm A-72 core at 16FF is ∼1.15 mm^2 [91]; our design is 13% of the single-core area, however, each *VLRD* is meant to serve N cores. A 16-core Arm A-72 configuration (like our simulation), excluding *L2* caches and wire overhead, would be approximately 18.4 mm^2. Based on this estimation, our *VLRD* shared by 16 cores, would occupy less than 1% of overall *SoC* area (adding *L2* and wire area would only improve this ratio).

4.2.4 Comparison with *CAF*

CAF [90] is a state-of-the-art hardware queue proposal similar to *VL* with a couple of differences: i. *CAF* divides buffers between queues and applies advanced credit management for QoS, while buffers in *VLRD* are shared by all queues; ii. *CAF* transferes 64-bit values between registers and Queue Management Device, whereas

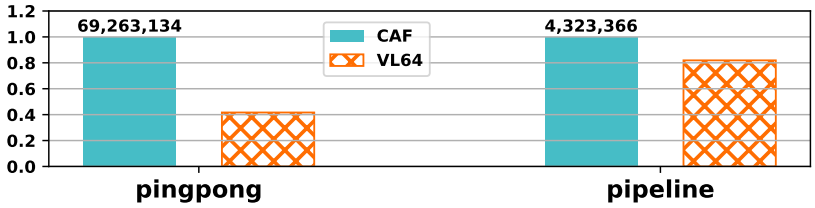


Figure 4.11: Performance comparison between *Virtual-Link* and a state-of-the-art hardware queue [90].

VL exploits cache lines as local buffer and as such lowers the frequency of performing relatively more costly data movement through the cache hierarchy. We compare *VL* with *CAF* on two benchmarks used in *CAF* paper, *ping-pong* and *pipeline*: *ping-pong* passes data through the queue, while *pipeline* uses the queue for pointers to 2 KiB network packet payloads. As shown in Figure 4.11, *VL* achieves 2.40× speedup over *CAF* on *ping-pong*, and 1.22× speedup on *pipeline*.

4.3 Summary

This chapter presents *Virtual-Link* (*VL*) a cross-core communication mechanism for fine-grained multi-threaded applications. *VL* is immune to cache contention for synchronization, provides back-pressure to reduce memory spills, and achieves low-latency cache injection by directly stashing the line into consumer *L1D* cache. This novel cross-core synchronization mechanism is similar to software queue mechanisms in flexibility but has the performance and efficiency of hardware solutions. Our full-system *gem5* simulation illustrated that we can obtain a 2.09× speedup and 61% average reduction in memory traffic over state-of-the-art software solutions across a variety of communications patterns and benchmarks.

Chapter 5: *SPAMeR*: Speculative Push for Anticipated Message Requests in Multi-Core Systems

5.1 *SPAMeR* Architecture Design

The *SPAMeR* design extends the *Virtual-Link* [94] architecture. This section first describes how *Virtual-Link* architecture is extended to give the functionality of *SPAMeR* (§ 5.1.1), then describes the specific micro-architecture (§ 5.1.2), ISA (§ 5.1.3), and library (§ 5.1.4) contributions. Section 5.1.5 explores the design space in terms of speculation algorithms; and Section 5.1.6 discusses potential security vulnerabilities and their mitigation.

5.1.1 How *SPAMeR* builds on the *Virtual-Link* Architecture

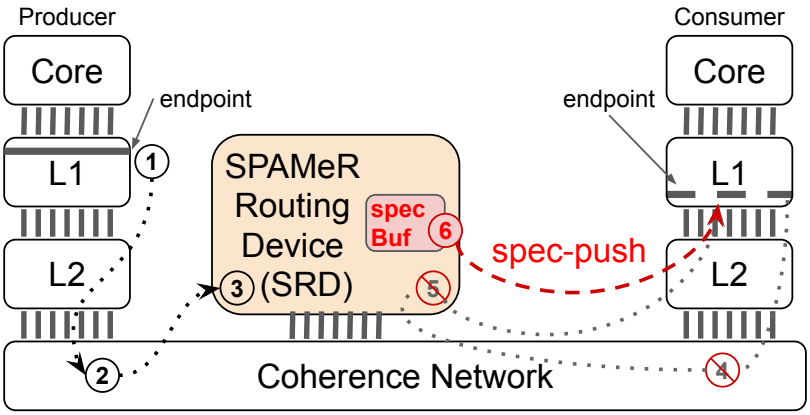


Figure 5.1: Overview of *SPAMeR* architecture. The *specBuf* added by *SPAMeR* in the routing device enables (**6**), speculative pushes. Speculative pushes can replace (**4**) and (**5**) in the baseline design and deliver the data with reduced latency.

Figure 5.1 provides an overview of the *SPAMeR* design, highlighting the

changes from *Virtual-Link* architecture (*VL*), in red. As mentioned before, there is a routing device (*VLRD*) attached to the coherence network in order to move data from core to core. The routing device is treated like a slice of the system cache or a tightly-coupled accelerator (as such a system could have more than one router) and could fit in any topology arrangement (the impact of topology and of multiple routers are not the focus of this dissertation). Each "endpoint", either producer or consumer, is a distinct address whose offsets serve as buffering points for data (e.g., a producer may have a 4 KiB page, the consumer a completely different page). With *VL* there is no shared coherent state despite giving the illusion of a shared memory connection, only a shared resource, the *VLRD*. Multiple endpoints from different cores can be associated with the same Shared Queue Identifier (*SQI*) and serve as one *M:N* queue. When a producer gets some data ready to push into the queue ((**1**) in Figure 5.1), the producer first selects the cacheline with the data using the `vl_select` instruction introduced in *VL*, then uses a `vl_push` instruction (2^{nd} new instruction from *VL*) to copy the content of the selected cacheline to the routing device ((**2**) in Figure 5.1). The `vl_push` instruction makes use of the existing cache data bus and is similar to cacheline flush or writeback. The differences lie in that `vl_push` does not change the coherence state of the cacheline, and the destination is a device memory address assigned to the routing device, rather than the memory controller. Once the routing device accepts the `vl_push` packet ((**3**) in Figure 5.1), the ownership of the data is transferred to the routing device, while the producer could start writing new data into the cacheline, which stays in the writable (e.g., exclusive) state before and after the `vl_push`. On the other side, the consumer issues a request for empty consumer endpoint via the `vl_fetch` instruction (at (**4**) in Figure 5.1, but could happen in any order with respect to (**1, 2, 3**)). The routing device matches the incoming producer data with a consumer request on the same *SQI* then copies the data over to the consumer cacheline at (**5**) via a stash operation.

The *SPAMeR* Routing Device (*SRD*) as shown in Figure 5.2 is analogous to the *VLRD* with several exceptions highlighted in red part (will be explained in

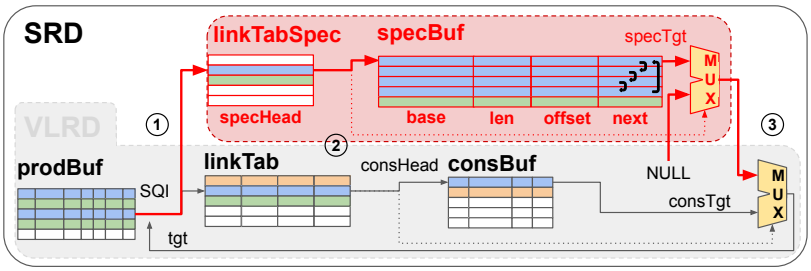


Figure 5.2: *SPAMeR* Routing Device (*SRD*). *SRD* includes the *VLRD* components from *Virtual-Link* Routing Device design (with the grey background), which we shrink for brevity, and some newly-added structures (highlighted in red) to enable speculative push. Cell filling colors indicate different *SQI* occupancy. Numbers in circle mark the three stages of the address mapping process.

Section 5.1.2). The *SRD* has a *prodBuf* to buffer the data copied from the producer endpoints, and *consBuf* to buffer the requests from the consumer endpoints. Each data/request takes up one entry. As the *SRD* serves multiple queues and *consBuf* entries are shared dynamically (e.g., the top *prodBuf* entry in Figure 5.2 is given to the blue *SQI*, but once the blue data gone, the entry could be filled with data for another *SQI*, say green). In addition to *prodBuf* and *consBuf*, the *SRD* has a *linkTab*, which stores the *SQI*-related metadata (i.e., head, tail to track consumer requests of each *SQI*) for all the queues, one per row. The aforementioned *SRD* action to pair producer data with corresponding consumer request is called address mapping. As labeled in Figure 5.2, the *SRD* builds a three-stage pipeline for address mapping: Stage 1 takes the *SQI* from *prodBuf* to lookup *linkTab*, getting `consHead`; `consHead` is used in Stage 2 to index *consBuf* and get the consumer cacheline address, `consTgt`; last stage is the only stage that writes back address mapping results and updates *prodBuf* and *linkTab*. It is worth mentioning that when a *prodBuf* entry enters the address mapping pipeline, there may or may not be a consumer request for the same *SQI* available, so a multiplexer in Stage 3 takes `consHead` (0 for no consumer request) as the select signal to pick between `consTgt` or *NULL*.

Figure 5.3 (except the red parts) shows what happens after address mapping in the original *Virtual-Link* architecture (the same process is used in the *SRD*).

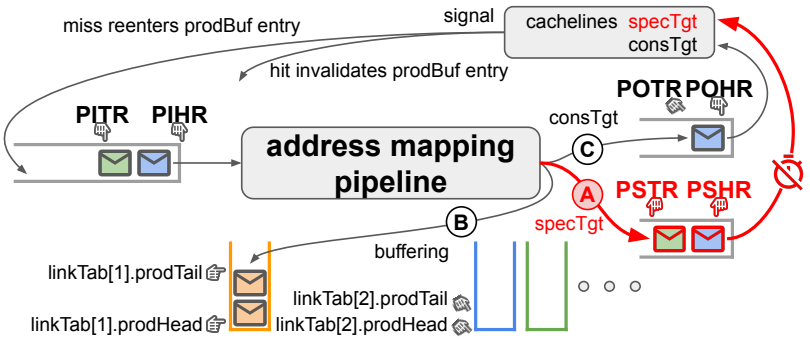


Figure 5.3: Address mapping in *SPAMeR*. There are three different possible outcomes from the address mapping pipeline (marked with letter A, B, C in circles), and the consumer cachelines give hit/miss response signals for every push (on-demand or speculative). Path and structures related to speculative push are highlighted in red, while the blue, green and orange colors indicate the affinity with different *SQI*. Please note that the multiple queues here are logical, physically each packet sticks to a *prodBuf* entry.

There are two possible outcomes: the data finds a target and gets into the sending queue (Path (**C**) of Figure 5.3); or it is temporarily buffered into a queue of the corresponding *SQI* (e.g., orange packets go into the orange queue as shown in (**B**) of Figure 5.3). Please note that the queues shown in Figure 5.3 are all logical queues, and the producer packet never left the *prodBuf* entry initially allocated to it. The logical queues are managed by the *SRD* via several pairs of head and tail pointers. For example, `PIHR` and `PITR` in Figure 5.3 stand for *Producer Input Head* and *Tail Register* respectively, holding the indices to the first and the last *prodBuf* entry that is going to enter the address mapping pipeline. After the address mapping, the *prodBuf* entry might be appended to a buffering queue (due to no consumer request on the same *SQI*). The corresponding head and tail pointers of the buffering queue are updated and stored in `prodHead` and `prodTail` fields of *linkTab* (e.g., the head, tail pointers for the orange buffering queue is in the first row of *linkTab*). When a producer packet reaches the front of the sending queue, the *SRD* sends the data through the coherence network to the consumer cacheline, then it receives the response signal from the targeted cache controller. If the data fills in the cacheline successfully, the *SRD*

frees the *prodBuf* entry; otherwise in the case when the target cacheline happens to be evicted or still holding valid data that cannot be overwritten, then the *SRD* would append the *prodBuf* entry after `PITR`, so that it would go through the address mapping pipeline again. These steps of the routing and mapping process are identical in both the *VLRD* and *SRD*. What makes *SRD* different is described next.

5.1.2 *SPAMeR* Routing Device

SPAMeR adds an additional speculative push path ((**6**) in Figure 5.1), which routes from the *SRD* to the consumer endpoint. The speculative push does not wait for the consumer request to arrive at the *SRD*, instead the *SRD* attempts to anticipate the request, speculatively sending the data to a consumer endpoint. In order to push speculatively, *SPAMeR* introduces a new data path in the routing device (the red part in Figure 5.2) that enables searching for a speculation target in parallel with the basic address mapping path. The buffer storage, *specBuf*, holds the target memory addresses and associated cachelines where the *SRD* could speculatively push data to. The *specBuf* is set by the application as Section 5.1.3 will explain. The *linkTabSpec* extends *linkTab* with the field `specHead` in order to store the index to a *specBuf* entry for the corresponding *SQI*. Therefore, from the *linkTab* lookup in Stage 1, we additionally get an index, `specHead`, to lookup *specBuf* in Stage 2 at the same time of looking up *consBuf*. Every valid entry in the *specBuf* represents a segment of memory (`specBuf.base` + `specBuf.len` $\times$ cacheline size) that *SRD* can speculatively push data to. The `specBuf.offset` field works like a counter for successful pushes: incrementing every time data is pushed to a consumer cacheline successfully, advancing by one till it reaches the limit (`specBuf.len`), at which point it is set to zero. We use `specBuf.offset` to derive target addresses for speculative pushes (i.e., `specTgt` = `specBuf.base` + `specBuf.offset` $\times$ cacheline size). This way, all consumer cachelines registered at that entry have a chance to receive data from speculative pushes. As mentioned before, more than one consumer endpoint

could be associated with the same *SQI* (e.g., 4 endpoints of blue *SQI* takes up 4 entries in *specBuf*). To link them up, there is a `specBuf.next` field per *specBuf* entry, which has the index to the next entry of the same *SQI*. When the address mapping result is written back in Stage 3, the `specBuf.next` field is used to update the `specHead` field in *linkTab*, so that for the next prediction, it would be a different *specBuf* entry that supplies `specTgt`. All the *specBuf* entry of a *SQI* form a loop and are used in turn. The speculative push address generated by *specBuf* is only taken as the target if there is no consumer request for this *SQI* in *consBuf*, otherwise the non-zero `consHead` value tells the multiplexer to pick `consTgt`. If the `specTgt` is selected, then the producer data logically enters the speculative push queue (Path (**A**) in Figure 5.3). After some delay, the *SRD* sends the data to the target cacheline. The delay is key for efficient speculation (further discussion in § 5.1.5 and § 5.2.2).

5.1.3 Instruction Set Extension

We need to update the *specBuf* in order to let the *SRD* know the addresses of cachelines that could potentially accept a speculative push. This task is fundamentally the same as entering a consumer request (*SQI* plus cacheline address) into *consBuf*. Two *VL* instructions already exist for this purpose: `vl_select` translates the virtual address of a consumer cacheline to the physical address and writes back to a system register (only readable by `vl_fetch`, `vl_push` instructions, the physical address is not user-space accessible) `vl_fetch` reads the physical address from the system register then writes it to a device memory address which belongs to the routing device. In *SPAMeR*, we allocate another range of device memory address for *specBuf*. A `vl_fetch` instruction writing to *specBuf* is under the alias `spamer_register`. When the *SRD* receives a `spamer_register`, the routing device updates *specBuf* rather than *consBuf*.

5.1.4 Library Optimizations

To enable software to make use of *SPAMeR*'s speculative push functionality, we first need to configure the *SRD* with the `spamer_register` instruction introduced in Section 5.1.3. This is configured in the same library function where *VL* creates consumer endpoints (the consumer cachelines associated with each endpoint are allocated in that function too). The original *VL* library is revised to register consumer cachelines with the `spamer_register` instruction before returning the endpoint to the user application. These consumer endpoints are *spec-push-enabled* and their cacheline addresses are recorded in *specBuf* after the `spamer_register` instructions, at this point the *SRD* can speculatively push data into these endpoints when appropriate. As a legacy option, user applications could request the library to provide non-speculative endpoints (i.e., when the `spamer_register` instructions are skipped). As we show in Figure 5.1, consumer requests (step 4) could be replaced by speculative pushes (step 6) for *spec-push-enabled* endpoints, thus *SPAMeR* further optimizes the dequeue library function by eliminating the part of the code issuing `vl_select` and `vl_fetch` at compile time. We also make the most frequently invoked queue functions as macros, so they are inlined at the compiler preprocessing phase, potentially avoiding some function calling overheads during execution.

5.1.5 Speculation Algorithms

SPAMeR speculation consists of two predictions: which cacheline and associated endpoint to speculative push to (e.g., 1 of M endpoints subscribed to a *SQI* that are speculation-enabled), and what is the perfect timing to push.

For the speculative push target selection, we let all valid *specBuf* entries participate in the address mapping in turn (§ 5.1.2), and rotate the target cacheline addresses in each entry (via `specBuf.offset`). This design collaborates with the library, which would use the cachelines of an endpoint in a round-robin fashion. Across

specBuf entries, the strategy sounds like round-robin, while it is actually weighted in two ways. First is that we can intentionally control the number of targets in the entries, and effectively adjust the speculative push rate for each target. For example, if we have one entry with 2 targets α and β, while another entry of the same *SQI* has only one target γ; Assuming the two entries receive the equal chance to be looked up during address mapping, the ratio between the three targets for receiving speculative pushes is $1 : 1 : 2$. In other words, the number of speculative pushes on target γ is doubled compared to target α, or β. Secondly, there is a throttling mechanism that sets an "on_fly" bit per *specBuf* entry when there exists a target from this entry in the speculative push queue. Until the previous speculative push finishes, this *specBuf* entry stops giving speculation target. Then the probability of selecting a target is effectively influenced by the delay prediction algorithms.

We first introduce two simplest delay prediction algorithms of the many we have evaluated. The first one is called `0delay`, which does not add any additional delay, but lets the speculative push go as soon as possible. The `0delay` algorithm can maximize the performance, because as long as there are available producer data in *SRD*, it keeps trying speculative pushes. This lets the `0delay` algorithm never miss the earliest chance to push the data into a consumer cacheline. The down side is that it could eat up bus/port bandwidth and affect other workloads. The second delay prediction algorithm adjusts the delay based on the speculative push results, so we refer it as the `adapt` delay algorithm. The `adapt` delay algorithm saves the delay values in registers (one per *linkTab* entry or per *specBuf* entry), and reduces the delay by half (right shift by 1-bit) upon a successful speculative push, otherwise double the delay for a failed speculative push. The `adapt` delay algorithm helps the *SRD* to build a profile of the consumer data ingest rate and pushes data according to their perceived ability to consume it. However, the `adapt` algorithm approach is too simple to fully model the consumer behavior (as the evaluation in Section 5.2.2 will show).

SPAMeR comes up with a `tuned` delay prediction algorithm that is tuned for

Listing 3 `tuned` algorithm `updateResponse()` function updates history based on prediction results.

```
updateResponse(link_id, is_hit) {
  spec_entry = specTab[link_id];
  if (is_hit) {
    // use the interval of the most recent hit responses as the
    // reference, [ref-τ, ref+ζ] is the scanning range
    spec_entry.delay = tsc - tau - spec_entry.last;
    spec_entry.ddl = tsc + zeta - spec_entry.last;
    spec_entry.nfills++;
    spec_entry.last = tsc;
  } else {
    elapse = tsc - spec_entry.last;
    stepped = spec_entry.delay + delta;
    doubled = spec_entry.delay << alpha;
    if (spec_entry.delay < spec_entry.ddl) {
      // before deadline, retry after δ
      spec_entry.delay = stepped;
    } else {
      // passed deadline, left shift α bits
      spec_entry.delay = doubled;
    }
  }
  spec_entry.failed = !is_hit;
}
```

Listing 4 `tuned` algorithm `lookupSpecTab()` function predicts delay.

```
lookupSpecTab(link_id) {
  spec_entry = specTab[link_id];
  halved = spec_entry.delay >> bithash(spec_entry.delay, tsc);
  elapse = tsc - spec_entry.last;
  if (is_init(spec_entry.nfills)) {
    // initializing phase
    return tsc + spec_entry.failed ? delta : 0;
  } else if (elapse < halved) {
    // early enough to try halved delay
    return spec_entry.last + halved;
  } else if (elapse < spec_entry.delay) {
    // early enough for planned delay
    return spec_entry.last + spec_entry.delay;
  } else if (!spec_entry.failed) {
    // data available later than planned and have not tried yet
    return tsc;
  } else if (elapse < spec_entry.ddl) {
    // planned delay falls behind, but not cross deadline yet
    return tsc + delta;
  } else {
    return tsc + spec_entry.delay;
  }
}
```

the benchmark which analysis suggests has the greatest potential (§ 5.2). The `tuned` algorithm takes interval between the most recent two successful pushes at the same endpoint as the reference to predict the delay for the next push to this endpoint. Because the intervals could fluctuate more or less, the `tuned` algorithm adjusts the delay from the reference in both multiplicative (i.e., shifting bits left or right), and additive (i.e., adding a constant delta) ways, creating a set of delays. This set of delays is then tried in chronological order. The yellow blocks in Figure 5.4 are the additional information latched in *specBuf* for the `tuned` algorithm to make its predictions. From the top to the bottom: `specBuf.nfills` counts the number of successful pushes; `specBuf.last` records the timestamp when the last push succeeds; `specBuf.ddl` sets the threshold (deadline) for the delay to multiplicatively increase once the deadline is exceeded; `specBuf.failed` is a one-bit flag indicating if the last push was successful; `specBuf.delay` holds the delay to be used in the current prediction. In Listing 3, the function `updateResponse()` shows how the values of each field get updated upon receiving a push response, and the corresponding logic circuit is shown on the right side of Figure 5.4. Function `lookupSpecTab()` in Listing 4, along with the left part of Figure 5.4, elaborates how the algorithm generates the delay time to send data. The orange Greek letters in Figure 5.4 are the parameters of the algorithm. Parameters ζ and τ outline a range around the interval reference (i.e., the duration between the 2 most recent successful pushes), and in the range, delay is increased by δ. Therefore, larger ζ and τ mean a wider range and more tolerance to the interval variation, and a smaller δ means denser steps, higher probability to deliver the data at the first moment. However, larger ζ, τ and smaller δ would contribute to more failures (we can see the trade-off in § 5.2.5). Parameter α decides how fast the delay would be increased after the deadline. Parameter β controls the phase of initialization phase (when delay is always increased by step δ). After tuning the parameters on a hard-to-predict benchmark, we pick a set of values ($\zeta = 128$, $\tau = 48$, $\delta = 32$, $\alpha = 1$, $\beta = 2$), then as a cross-validation, apply the parameterized algorithm to all the benchmarks in the evaluation. As future work, *SPAMeR* could be dynamically reconfigured with

an optimal set of parameters for each benchmark.

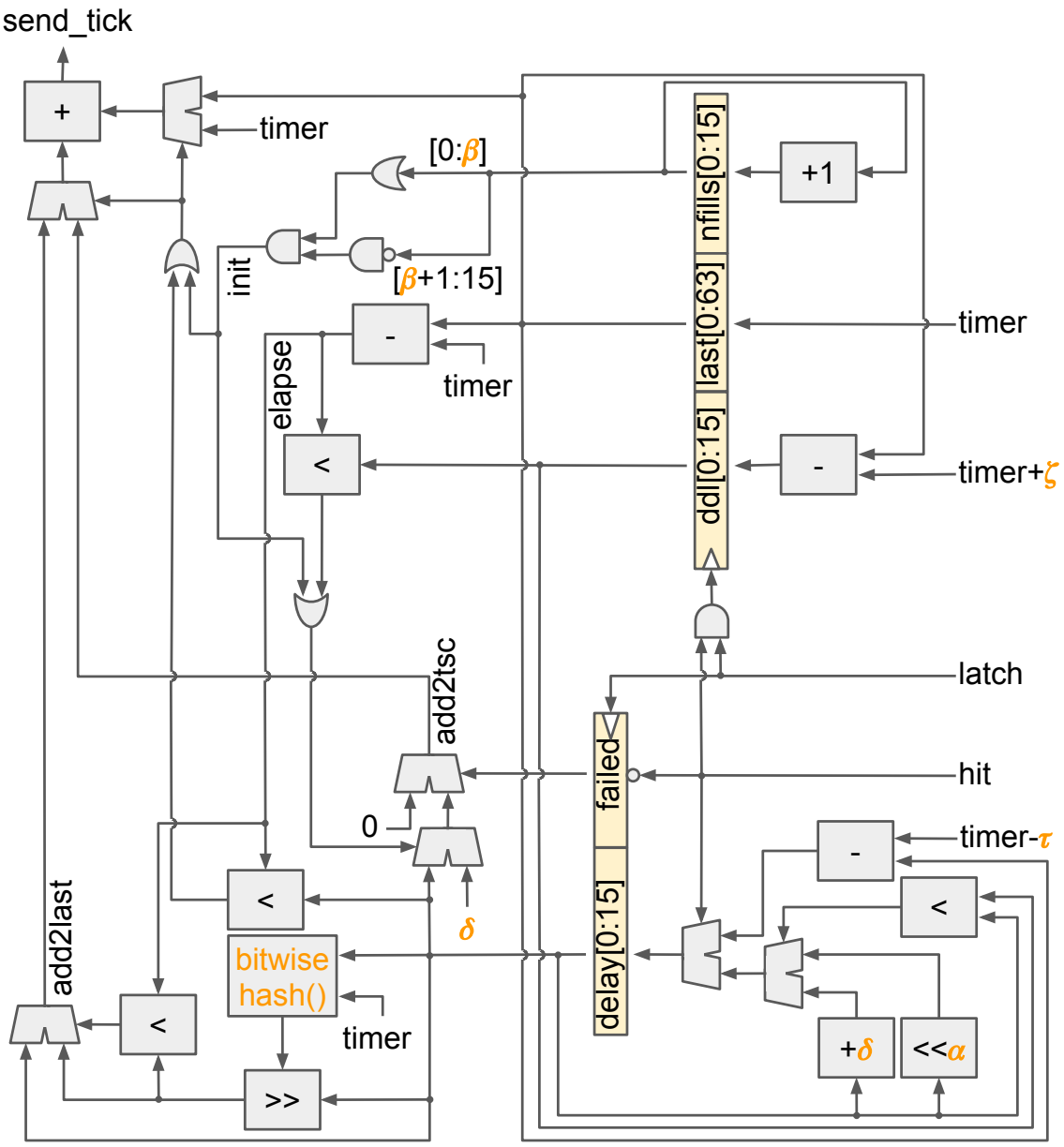


Figure 5.4: The example hardware logic implementation of the proposed `tuned` delay prediction algorithm. All "timer" label in the diagram refer to the time stamp counter (tsc), while "timer+ζ" and "timer-τ" input ports on the right could be from two other time stamp counters configured to have constant offsets (i.e., ζ, $-\tau$) from tsc.

5.1.6 Potential Vulnerabilities and Mitigation

It may be thought that speculative pushes could be like prefetching that is vulnerable to side-channel attacks. However, a few differences between *SPAMeR* and cache prefetchers make *SPAMeR* more secure. The 3 most popular ways to attack prefetching via side-channel would not work on *SPAMeR*: 1) HW-prefetcher metadata, such as stride, leaks secrets [78, 19]. The latency counters in *SPAMeR* might also have the secrets but there are isolation (counters are per-endpoint, and each endpoint is assigned uniquely to a thread) and obfuscation (augmented by random chance) to prevent secrets from leaking. 2) Content-based prefetcher might take the secret (brought by transient instructions sometimes) as a hint of prefetching address [4], while *SPAMeR* does not use content for prediction. 3) Attacker could derive memory layout from prefetching latency [35]. In contract, the destination of the speculative pushes must be "push-enabled" (registered, marked) by the target core, effectively white-listing specific cache lines of an endpoint as being amenable to a speculative push. Therefore, attacker cannot gather any useful information from *SPAMeR* for the memory layout. Regardless the style of side-channel attacks, it is also more difficult to probe *SPAMeR* than prefetching, because the prefetching changes cacheline coherence state [37], while speculative push does not.

Another security concern is that a malicious producer could aggressively occupy many *SRD* and network resources for DoS, or inject malicious data messages into the channels of other processes (e.g., if this mechanism was used to push lambda threads, then an attacker could potentially execute arbitrary code with privilege). However, attackers would have to first bypass all existing mitigation provided by the virtual memory system architecture. As in *VL*, *SPAMeR* allocates or frees resources via system calls similar to memory management (no new system calls are added by either *SPAMeR* or *VL*), so DoS can be mitigated by setting limits (e.g., ulimit for soft limits, and AArch64 MPAM extension allows the microarchitecture to enforce resource utilization like bandwidth per partition-id).

Lastly, the speculative push feature of *SPAMeR* is enabled per endpoint. If a program (or a thread) has a specific security concern or higher confidentiality requirements, it could disable speculation per-endpoint or totally per *SQI*. Based on the discussion above, we believe *SPAMeR* design is vulnerability-free as for now.

5.2 Evaluation

This section first analysis the trace of message queue transactions to reveal the potential latency saving by speculation (§ 5.2.1). Then *SPAMeR* is evaluated from the perspectives of performance (§ 5.2.2), micro-architecture impacts (§ 5.2.3, 5.2.4), sensitivity (§ 5.2.5), and the area and power cost (§ 5.2.6).

5.2.1 Message Transaction Tracing Analysis

In order to get an intuitive sense on how speculation would hide the cross-core data movement latency, this section develops a trace-based analysis. Figure 5.5 illustrates this analysis with the message transaction trace from *incast*. Please note *incast* is chosen because *incast* can have the simplest queue setting (single producer, single consumer, single queue, single cache line), so that it would be relatively easy for reasoning. As shown on the y-axis of the bottom chart in Figure 5.5, there are a few events specifically to look at in the trace. From the bottom to the top, the diamond marker at the lowest row indicates data arrival from the producer to the *SRD*, and the dot marker on the second lowest row is for the request arrival from the consumer to the *SRD*. The square marker above that indicates when the consumer cacheline is ready to receive new data, and the marker on the second top row is for when the producer data fill into the consumer cache target, followed by the topmost marker ($\times$) for the consumer's first use of the data. For each marker, its x -axis value is the timestamp.

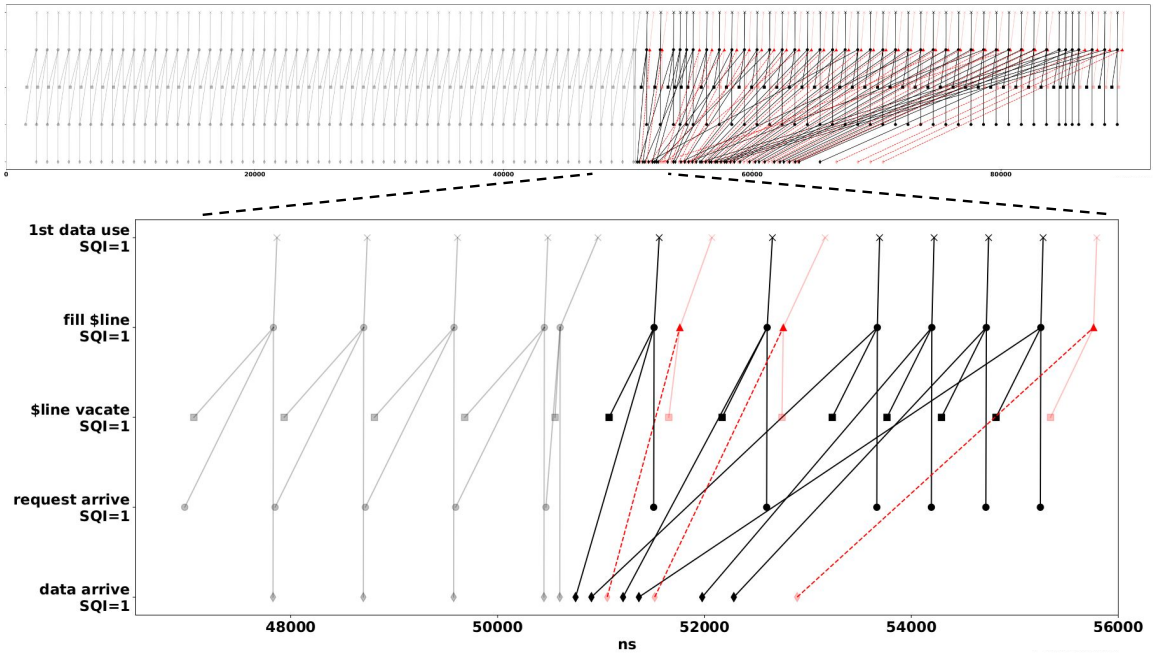


Figure 5.5: Message transaction trace. One of the simplest traces showing different message queue transactions in *incast* benchmark (single *SQI*) with single consumer cacheline, single producer thread. For each marker, its x axis value is the timestamp, and it is vertically located according to the event type. Red dashed lines indicate speculative pushes (no request arrival), while solid lines are on-demand pushes to fulfill consumer requests. Darker lines are those transactions that could have shorter latency with speculation.

From the overview chart at the top of Figure 5.5, we can see two phases as designed by the benchmark to explore different scenarios: when the consumer runs faster at the beginning, transactions happen in a stable fashion, and the throughput is bounded by the slower producer; after about 50 000 ns, the producer starts to generate a burst of data and the consumer becomes the bottleneck.

The bottom chart of Figure 5.5 zooms in to reveal more details at the transition of these two phases. The markers joined by lines are the different events of the same transaction. For on-demand pushes (solid lines in the chart), data arrival, request arrival, and cacheline vacating must precede the cacheline fill, while speculative pushes (red dashed lines) have no request arrival event associated to the transaction. Request arrival is a false dependence, while data arrival and cacheline vacating are necessary either way of doing pushes. Figure 5.5 highlights some of the on-demand push transactions in dark black, because in these transactions, filling the consumer target with data is hindered by the request arrival, the last one among the three events that an on-demand transaction requires. If a speculative push was triggered, the delivery of the data could have happened earlier as soon as both data arrival and cacheline vacate events happen. Therefore, the potential speculative push saving in a transaction is calculated as the difference between the cacheline fill timestamp and the timestamp of data arrival or cacheline vacating (whichever comes later).

$$s = f - max(v, a) \tag{5.1}$$

Equation 5.1 shows the calculation of potential speculative push saving (denoted as s) in a transaction. In Equation 5.1 f stands for the timestamp when filling cacheline happens, and v, a are for cacheline vacate, data arrive, respectively. The same calculation could apply to all transactions. Let S be the total potential speculation saving in the execution of a program, then it could be calculated as shown in Equation 5.2, where the subscript i means the i-th transaction in the program:

$$S = \sum_i s_i = \sum_i (f_i - max(v_i, a_i)) \tag{5.2}$$

Not all transactions in Figure 5.5 are highlighted. Those dim on-demand transactions in Figure 5.5 have zero potential latency saving via speculation. That is because in these dim transactions, either the data is not available or the cacheline is not ready until the cacheline filling actually happens.

It is noticed that a "prerequest" behavior exists in the trace. For example the leftmost transaction in the zoom-in section, has the request arrival earlier than the cacheline actually becomes empty. It is because when the consumer is looping to pop a queue, it is highly likely a `vl_fetch` instruction is going through the cache hierarchy when data is filled into cache. That leads to the "prerequest" phenomena. The "prerequest" is not guided, and its random impact on the performance of *VL* is observed later (§ 5.2.2). *SPAMeR* replaces such "prerequest" with educated speculation.

5.2.2 *SPAMeR* Speedup

As mentioned in Section 5.1.4, we optimize the library by applying function inlining and fetch skipping. Experiments reveals the inline function has limited improvement (1.02× speedup on average). Nevertheless, in the following evaluation, we apply the function inlining optimization to the baseline *Virtual-Link* setting as well, in order to show the benefits brought purely by speculation.

Figure 5.6 compares the performance of *SPAMeR* against the baseline, *Virtual-Link*, as well as another state-of-the-art hardware queue, *CAF* [90]. As we can see, with the aggressive `0delay` algorithm, *SPAMeR* is able to achieve more than 1.24× speedup over the baseline on 5 of the benchmarks. The highest speedup, 2.59× occurs on *FIR*, where the filtering stage workers stay on the fast path all the time with the shorter latency. There is almost no performance gain on *ping-pong* and *sweep*, because the consumers in those benchmarks are always ready ahead while the data production is on the critical path. Without available producer data, *SPAMeR* is not able to try speculative push for the first place. The two queues in *bitonic* are biased,

and the starvation of producer data also happens to the (1:N) queue. For all the benchmarks except *FIR*, the `adapt` delay algorithm obtains performance improvement fairly close to the `0delay` algorithm. This is because the *FIR* worker threads could switch between fast path and slow path, and the `adapt` algorithm adjusts the delay too dramatically, then easily it learns the period of slow path instead of the fast path. In contrast, because the `tuned` algorithm would carefully increase the delay additively to approach the fast path period, it is able to lock the worker threads on the fast path for the most of time. On average across all the benchmarks (geometric mean), *SPAMeR* with `0delay` algorithm, the `adapt` and the `tuned` delay algorithm get 1.45×, 1.25× and 1.33× speedup, respectively. Another state-of-the-art hardware queue, *CAF*, has the option to prepush (by producer instructions) the payload to where is closer to the consumers. We enable the basic prepush scheme described in *CAF* design in order to make the comparison fairer. As shown in Figure 5.6, *SPAMeR* beats *CAF* on all the benchmarks. On *ping-pong*, *pipeline*, and *firewall*, both *Virtual-Link* and *SPAMeR* enjoy the advantages of batching messages in the cachelines, which *CAF* is unable to do as it uses registers instead of cachelines. On the other benchmarks, *SPAMeR* speculation is more effective than the software-guided prepush in *CAF*, because *SRD* has accurate hardware timing information and precise target addresses. Overall, *SPAMeR* with `tuned` algorithm achieves about 1.82× speedup over *CAF*.

5.2.3 Speculation Effects on Cacheline Occupancy and Transaction Latency

Figure 5.7 breaks down the execution time into two: when the consumer cacheline is empty and the rest. This provides the insight of where does *SPAMeR* saves time. For *VL*, the cycles when a consumer cacheline is empty could include the time spent on requesting data and waiting for the data to arrive. As Figure 5.7 indicates that on most benchmarks, *SPAMeR* cuts off some empty cycles to reduce the total execution time; while *SPAMeR* might also transfer some empty cycles into non-empty

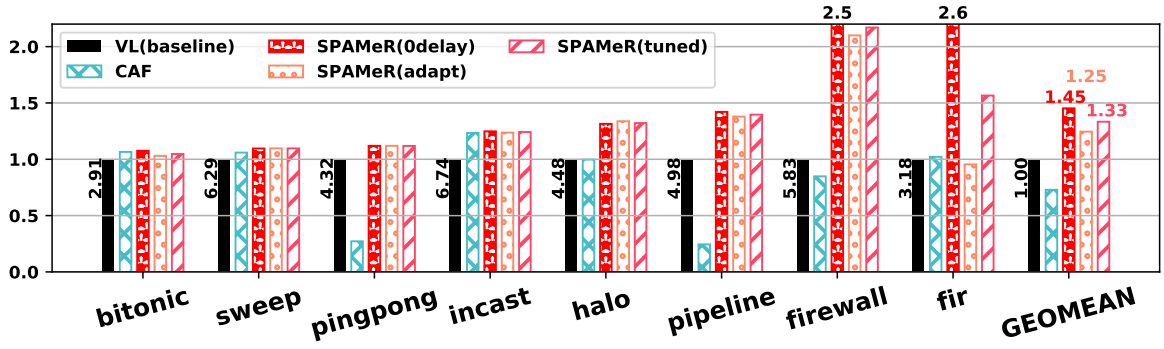


Figure 5.6: Performance comparison between *Virtual-Link* and *SPAMeR*. The bars show the speedup *SPAMeR* gains over *Virtual-Link*, the higher the more performant. Execution time is normalized to the baseline (labeled on the left of the black solid bar in millisecond).

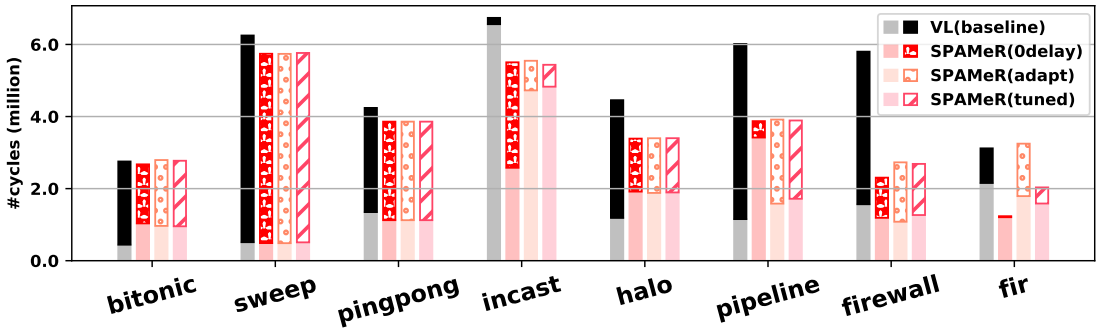
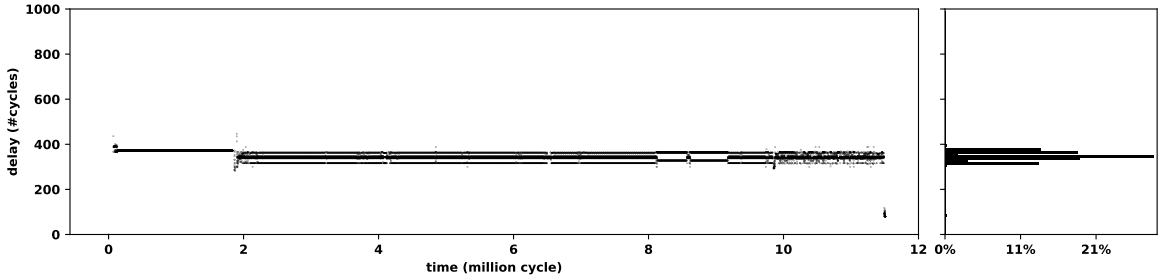


Figure 5.7: Time breakdown with the respect to cache line emptiness. The top section of the bars stands for average consumer cacheline empty cycles, and the bottom section of the bars is for non-empty.

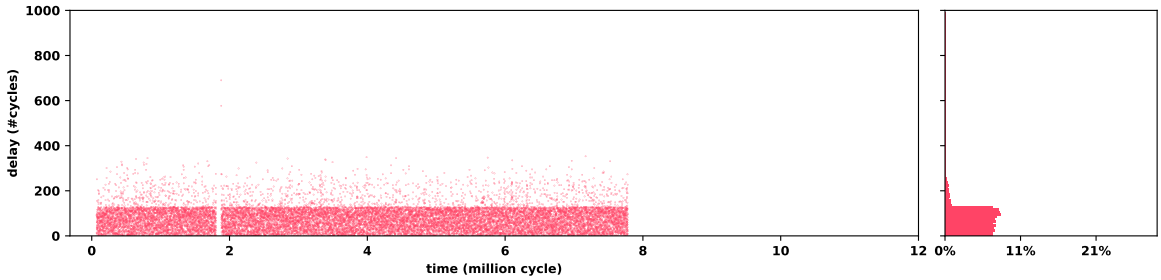
once hit peak consumer throughput, for example, *bitonic* and `0delay` on *pipeline*. This observation validates the philosophy of *SPAMeR* design that speculation could get the data into consumer cachelines earlier and take chances to reduce load-to-use latency. There are 32 consumer cachelines in *incast*, and `0delay` might quickly fill up all 32 cachelines then blocked on one (round-robin as designed in § 5.1.5), until the consumer thread uses up all data in other cachelines. This pattern causes half of the cycles in *incast* with `0delay` algorithm are empty cycles. For *FIR*, with data ready earlier, *SPAMeR* reduces the number of times for the *FIR* threads going through the slow path, where the consumer cachelines are likely filled when half way through. Therefore, *SPAMeR* is able to reduce the non-empty cycles in *FIR* considerably as well by avoiding the slow path.

Other than the CPU cycle breakdown analysis, the trace analysis (§ 5.2.1) provides more insights on how *SPAMeR* affects the transaction latency. As illustrated in Section 5.2.1, there could be potential latency (more or less) to overlap in each transaction of moving data from the routing device to the consumer cacheline. Figure 5.8 presents the potential latency saving for the bottlenecked queues (consumer is slower) in *pipeline* and *FIR*. In Figure 5.8a, a large cluster of transactions having relatively long latency (around 380 cycles) is spotted. This is because *Virtual-Link* does no speculation, and if the consumer requests come lately, considerable load-to-use latency is involved in the transaction. In contrast, Figure 5.8b shows many transactions skew towards 0-cycle delay, and the majority are now limited within about 160 cycles (which is determined by the selected parameters, $\zeta + \delta$). Similarly, the comparison between Figure 5.8c and Figure 5.8d suggests *SPAMeR* `tuned` algorithm brings lower latency to many transactions. However, the occasional switching to the slow path in *FIR* disturbs the algorithm, leading to about 17% transactions predicted too late by mistake.

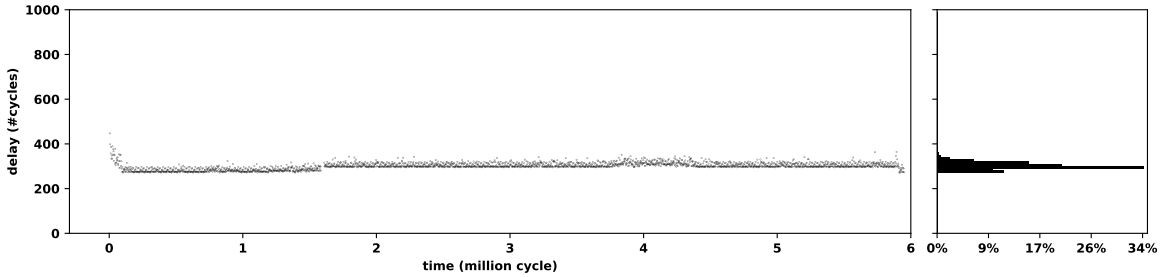
For each benchmark, Figure 5.9 aggregates the latency over all transactions, and shows how much potential latency saving that *SPAMeR* `tuned` algorithm has achieved with the respect to *VL*. As we can see, the `tuned` algorithm lowers the total



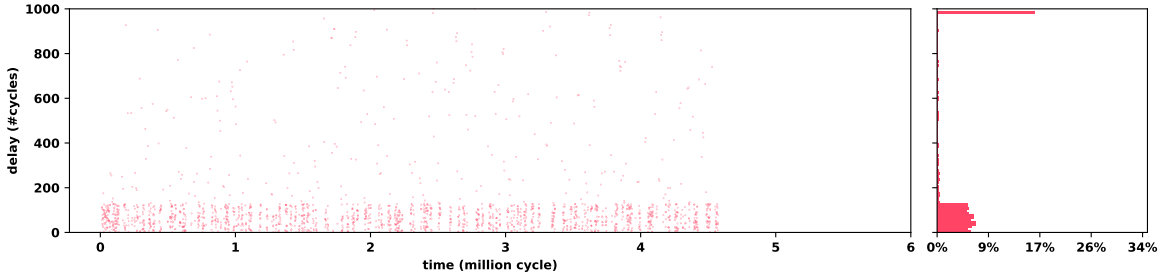
(a) *pipeline* Queue 3, *VL* (baseline)



(b) *pipeline* Queue 3, *SPAMeR* (`tuned`)



(c) *FIR* Queue 1, *VL* (baseline)



(d) *FIR* Queue 1, *SPAMeR* (`tuned`)

Figure 5.8: Potential latency saving per transaction over time and the distribution. Each dots marks a transaction. Higher potential latency saving/delay means worse.

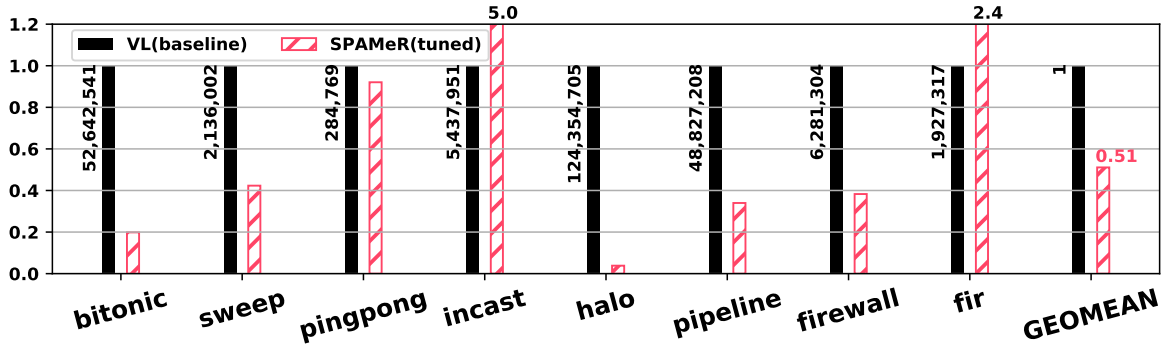
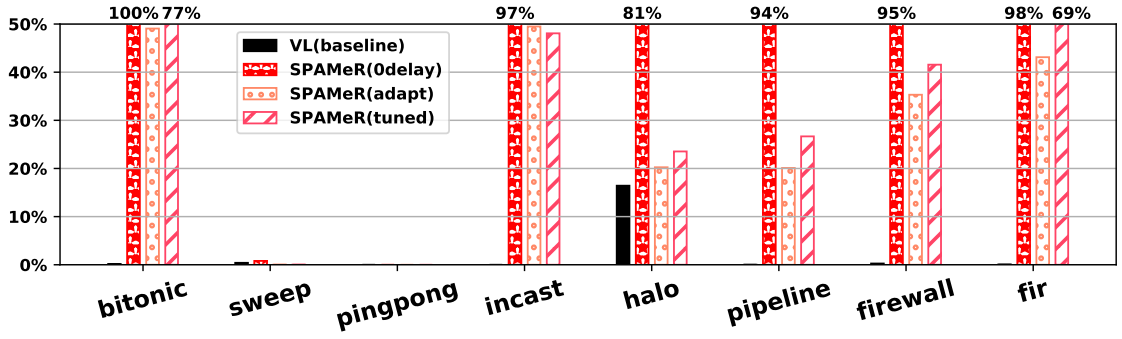


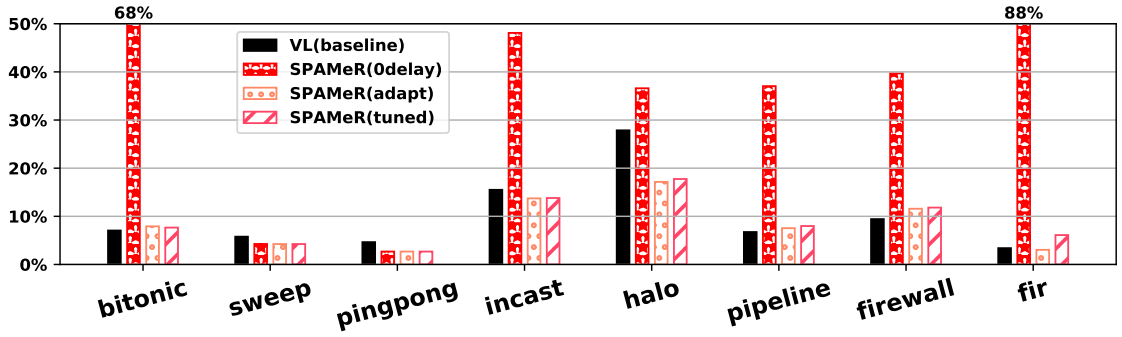
Figure 5.9: Total potential latency saving. *SPAMeR* (`tuned`) have less potential latency saving compared with *Virtual-Link*, because *SPAMeR* has already overlapped some data movement latency. The absolute numbers on the left of the black solid bars are the total cycles of potential speculation saving (Equation 5.2).

potential speculation saving in 6 out of the 8 benchmarks, and the geometric average (GEOMEAN) across benchmarks suggests an about half latency reduction. There are two exceptions, that *SPAMeR* has higher latency on *incast* and *FIR*. In *incast*, *SPAMeR* algorithm is able to figure out among the multiple consumer cachelines, which is the one that the consumer needs the data the soonest and what is the pace for other ones, so it essentially saves the most critical latency but allows some laziness to populate every consumer cachelines. If the number of consumer cachelines in *incast* is reduced to 1, our experiment tells *SPAMeR* (`tuned`) has about only 0.82× latency of *VL*. In *FIR*, the baseline *VL* successfully matches the speed of each stage by letting the worker threads always go through the slow path every iteration. Whereas *SPAMeR* tries to accelerate the data passing in order to get the worker threads on the fast path. When using the `tuned` algorithm, most of the iterations are tuned into fast path, but there exists transitions back and forth fast path and slow path sometimes. It is those occasional mismatches increases the overall latency for *SPAMeR* (`tuned`). When `0delay` algorithm is applied, every iteration in *FIR* is fast path, and the potential speculation saving is reduced to about 9% of the baseline.

5.2.4 Failure Rate and Bus Utilization



(a) push failure rate



(b) bus utilization

Figure 5.10: Push failure rate and bus utilization comparison between *Virtual-Link* and *SPAMeR*. The higher the less efficient.

Despite of achieving better performance, *SPAMeR* raises the worry that the retries after failures could costs more bus utilization and energy than the baseline, especially for the `0delay` algorithm. It is such a concern that motivates the development of the `adapt` and the `tuned` delay prediction algorithms. We evaluate their effectiveness in this section. In Figure 5.10a, we compare how many pushes (counting both on-demand pushes and speculative ones) fail out of total across baseline and *SPAMeR* with 3 different algorithms. There is no push failure for *VL* on almost all the benchmarks, except *halo*. There is grid of multiple threads in *halo*, and the threads

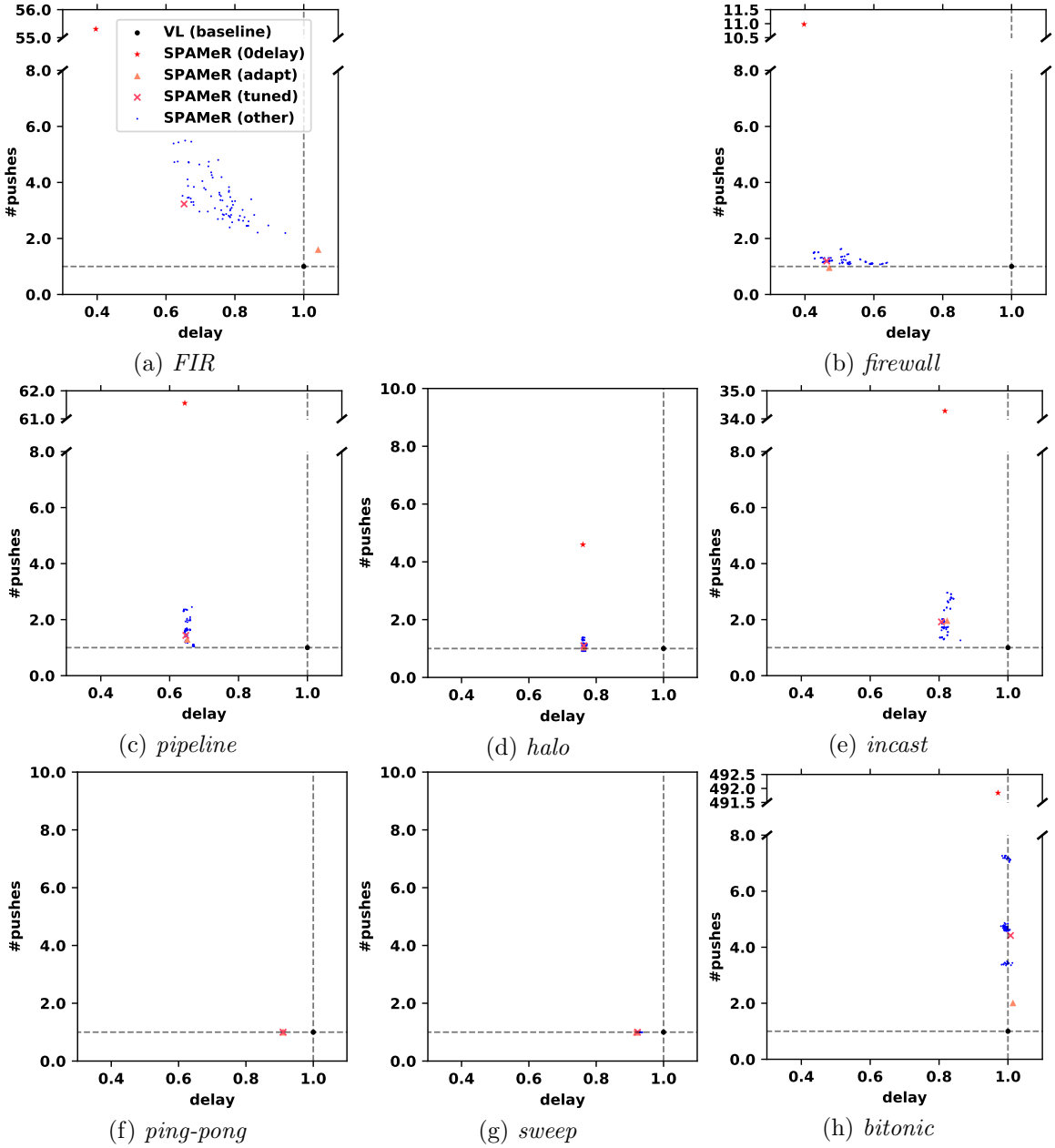
might frequently request data again and over again from their neighbors, leading to a higher chance for the unintended "prefetches". Also a single thread in *halo* might need to handle 2 to 4 queues, so a data in the consumer cacheline is not guaranteed to be taken timely, then some of the "prefetches" would fail. However, due to the plenty speculation opportunities exist in *halo*, *SPAMeR* gets 1.33× speedup on *halo*, and such "prefetches" is actually beneficial to the the overall performance of the *VL* baseline as well (without "prefetches" *VL* would be 0.94× slower on *halo*). *SPAMeR* with `0delay` algorithm shows super high failure rates on most of the benchmarks as expected. *ping-pong* and *sweep* share a pattern in common that the data packets go back and forth between two ends periodically and by the the time the data packet is back visiting a node again, there is sufficient time to have the consumer cacheline ready to accept new data. Therefore, *ping-pong* and *sweep* are the only two where `0delay` algorithm would not make many failures. The `adapt` delay algorithm manages to lower the failure rate under 50% on all the benchmarks. Because *SPAMeR* changes the two-way traffic (request and data push) in *VL* to one-way, 50% failure rate means *SPAMeR* would have equal or fewer packets going through the bus (verified in Figure 5.10b). The failure rate for the `tuned` algorithm is slightly higher than the `adapt` algorithm. Achieving near-zero speculative push failure rate for arbitrary workload is not easier than improving the prefetching accuracy to almost 100%, however, the most common prefetchers (stride prefetcher and Markov prefetcher) can only get accuracy on around 50% [88].

The higher the push failure rate, the more wasted traffic on the bus. Figure 5.10b reports the bus utilization, which is the percentage of cycles that have at least one packet (request or data) reaches the bus. As we can see, *SPAMeR* with `0delay` algorithm consumes much more bandwidth than others on most of the benchmarks. *SPAMeR* with the `adapt` or the `tuned` algorithm has comparable or even lower bus utilization than the baseline. The reason is that for each successful on-demand push in *VL*, there must be a consumer request going through the bus before, so the total number of transactions is twice as the number of successful pushes.

Since the `adapt` delay algorithm is able to bring the failure rate under 50% for most benchmarks, there are chances for it to spare more bus cycles than the baseline.

5.2.5 Sensitivity Study

There are several parameters (i.e., ζ, τ, δ, α, β) in the `tuned` algorithm design, so this section explores the sensitivity to the parameter combinations as shown in Figure 5.11. Every marker represents a speculation algorithm (or `tuned` algorithm with different parameters). Their x-axis and y-axis values are benchmark end-to-end execution time (denoted as delay) and the number of pushes, respectively. As the more pushes the routing device does (no matter on-demand or speculative push), the more energy it would consume, so the number of pushes is taken to reflect the dynamic part of energy in this analysis. Different benchmarks have different communication patterns therefore react differently to the varying parameters. In order to gain a consistent feeling about the parameter sensitivity across benchmarks, the scale is kept the same for all benchmarks after normalizing both delay and energy to the baseline (the black dot). Apparently, the closer to the origin point, the better an algorithm is (meaning running faster with less energy cost). As Figure 5.11a reveals, *FIR* is hard-to-predict as the adaptive algorithm (the triangle marker) only wastes energy on improperly-timed pushes and cannot reduce the execution time; the 0-delay algorithm (the star marker) gets good speedup on *FIR* at the cost of too much higher energy to be realistic. The parameters of the `tuned` algorithm allow us to balance the trade-off in between as those small blue dots in Figure 5.11a illustrate. Because the set of parameters ($\zeta = 128$, $\tau = 48$, $\delta = 32$, $\alpha = 1$, $\beta = 2$, the cross marker) we choose is based on the tuning on *FIR*, it is one of the settings that are on the side closer to the origin point. As Figure 5.11b to Figure 5.11h show, the chosen parameter setting might be sub-optimal on other benchmarks, for example, there are parameter combinations run slightly faster on *firewall*, or cost marginally less energy on *incast*. Nevertheless, the `tuned` algorithm parameters have very limited impact if not none



(a) *FIR* (b) *firewall*

(c) *pipeline* (d) *halo* (e) *incast*

(f) *ping-pong* (g) *sweep* (h) *bitonic*

Figure 5.11: Execution time v.s. the number of pushes (including on-demand pushes and speculative pushes, serves as an estimation for the dynamic part of energy consumption). Both axis are normalized to *VL* baseline. Other combinations of the `tuned` algorithm parameters are included to show how is the algorithm sensitive to the parameters.

on the performance of other benchmarks. With this validation, we believe if only one fixed set of parameters must be hardened for the `tuned` algorithm, we can tune it for the hard-to-predict workloads and it should work well with other insensitive applications.

5.2.6 Area and Power Estimation

The *Virtual-Link* [94] work estimated the area cost of the *VLRD* by developing RTL code and scaled the synthesis result on the FreePDK 45 nm library [82] to 16 nm technology node [81]. Given the fact that *SRD* shares its major structures and data paths with *VLRD*, we follow the same methodology as *Virtual-Link* to estimate the area cost. RTL synthesis and scaling shows that with the additional *specBuf*, *SRD* uses 0.156 mm^2 for all the buffers, and the overall area is 0.170 mm^2 (within 15% increase from the area of *VLRD*). As a single Arm A-72 core at 16FF is reported to be ˜1.15 mm^2 [91], the 16-core Arm A-72 configuration we simulate should be at least 18.4 mm^2 (excluding *L2* caches and wire overhead), making *SRD* cost less than 1% of the overall *SoC* area. This estimation is based on the basic setting with 64 *specBuf* entries using the `0delay` algorithm. Different delay prediction algorithms (e.g., `adapt` delay algorithm) might require additional storage and control logic. 64 *specBuf* entries are more than the benchmarks need (at most 48), while if there is a situation where the workloads register more *specBuf* entries, the operating system needs to manage the *specBuf* as other limited resources (e.g., physical memory).

With 16FF and 0.86 V supply voltage, the power of the baseline, *VL* is estimated to be 9.33 mW (dynamic) and 0.82 mW (leakage). As considering *SRD* pushes more frequently than *VLRD* does, we multiply the dynamic power by the factor of push frequency. It turns out the `0delay` algorithm would yield too much higher power to be realistic, while the `adapt` and the `tuned` algorithm are bounded to be at most 2.45×, 5.03× more than *VL*, respectively. That is 47.75 mW for *SRD* power in total at most. The power of a 20 Cortex-A72 processor with 28 MB cache is reported to be around 30 W [91], so assuming a 16-core SoC system consumes about 21 W power,

SRD would only contribute to about 0.23% of the total power. Since the power ratio is at the same magnitude of its area share, so *SRD* is unlikely to be the peak thermal component.

5.3 Summary

This chapter presents a novel mechanism, *SPAMeR*, to reduce the cross-core communication latency in multi-core systems. In *SPAMeR*, there is a routing device that anticipates the incoming requests, then speculatively pushes the data into a target consumer cacheline. Our full system simulation using the *gem5* infrastructure illustrates that *SPAMeR* is able to obtain 1.33× speedup over a state-of-the-art hardware message queue architecture on 8 task-parallel benchmarks, as well as outperforms another state-of-the-art hardware queue design that does data movement speculation by 1.82×. This chapter also uses *gem5* to study the benchmarks, and perform a detailed analysis on the message queue communication overhead. The proposed architecture would assist the effectiveness of multi-core systems handling message queue task parallel workloads.

Chapter 6: *ARMQ*: A Light-Weight Runtime for Message Queue Task Parallelism

6.1 *ARMQ* Runtime Design

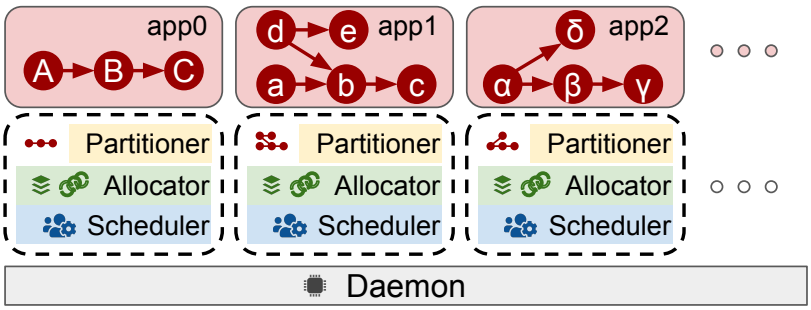


Figure 6.1: *ARMQ* architecture.

Figure 6.1 visualizes *ARMQ* design at a high-level abstraction. At the top there are the applications written as a Directed Acyclic Graph (DAG, a graph cycle otherwise inherently introduces complexity and risks [83]) of computation kernels (§ 6.1.1). Programmers are expected to merely focus on the computation logic and the dependencies between the kernels. The execution of the applications are delegated to the runtime modules. It is allowed to exercise a variety of runtime schemes suiting to the characteristics of different applications alongside. The runtime scheme is the combination of the following three modules: 1) partitioner analyzes the application DAG and groups kernels (§ 6.1.2); 2) allocator is responsible to manage the queue buffers where the data resides (§ 6.1.3); and 3) scheduler creates tasks (and maps to threads) to fulfill the computation of each kernel (§ 6.1.4). As a part of *ARMQ*, there is a system-wide daemon that balances the CPU core allocation across all applications.

6.1.1 Application Programming Interfaces

ARMQ aims to keep the interface exposed to programmer simple and intuitive, with the goal of minimizing programmer effort to use. *ARMQ* borrows the streaming programming style from the C++ template streaming library *RaftLib* [14], and extends it (§ 6.1.1.1). The *ARMQ* application programming interface enables passage of two essential pieces of information from programmers to the underlying runtime: the first (and most obvious) is the computation to perform in each compute kernel (Listing 5), the second, which is critical for *ARMQ*, is connectivity information with critical meta-data to describe how messages are passed between kernels (Listing 6).

Listing 5 An example *ARMQ* computation kernel that does filtering.

```
1 class Filter : public armq::Kernel {
2  public:
3   Filter() : armq::Kernel() {
4     add_input<int>("0"_port); /* add an input port */
5     add_output<int>("0"_port); /* add an output port */
6   }
7   virtual armq::kstatus::value_t
8   compute(armq::StreamingData &dataIn,
9           armq::StreamingData &bufOut) {
10    auto val(dataIn.pop<int>()); /* pop input message */
11    if (0 != val) { /* filter away zero values */
12      bufOut.push(val); /* push out non-zero values */
13    }
14    return armq::kstatus::proceed;
15    /* proceed to next invocation */
16  }
17 };
```

Listing 5 gives an example of computation kernel (*referred to simply as kernel from this point forward*) described using *ARMQ*. The user-defined kernel, `Filter`, inherits from the `armq::Kernel` base class which has a set of structures and methods that are designed to be used by the runtime. The computation of a kernel is described through the implementation of a `compute()` function. The `compute()` function receives input data, applies the computation written by the programmer, then sends output data (e.g., the kernel receives data streams, acts on it, then if there is an output, streams output data). This `compute()` function is only invoked when conditions

set by the runtime are met (e.g., it could be called constantly, or only when data is available on input streams).

Listing 6 An example 3-stage pipeline application in *ARMQ*.

```
int main() {
  Generater gen; /* a random number generate kernel */
  Filter f; /* the filter kernel */
  Print p; /* a kernel printing received values */
  armq::DAG dag; /* the Direct Acyclic Graph */
  /* add a 3-stage pipeline to DAG */
  dag += gen >> f >> p; /* streaming style */
  /* then execute the DAG with specific runtime scheme */
  dag.exe< armq::RuntimeBasic >();
  return 0;
}
```

Listing 6 is an example three stage application pipeline composed using *ARMQ*. The three kernels that make up this pipeline are: 1) `Generator` generates and sends out random values, 2) `Filter` passes only the non-zero values received to the next stage, 3) `Print` prints out every received value. The *RaftLib* template library specified several C++ operator overloads to define its Domain Specific Language (DSL). Within that DSL, connections between each kernel are specified by a stream operator (`>>`), the underlying transport layer of each connection is semantically undefined; at this point only the topology is specified. *ARMQ* extends the usage of operator reloading in *RaftLib* to capture extra heuristic information (§ 6.1.1.1) from the right-hand-side of the add-increment operator (`+=`) into the internal *ARMQ* representation of the compute Directed Acyclic Graph (DAG). Upon calling the `exe()` function of *ARMQ*, a runtime execution scheme is selected (i.e., in Listing 6, by default it is a preset runtime scheme defined by `armq::RuntimeBasic`).

6.1.1.1 Hints for the Runtime

Listing 7 An example of *ARMQ* hints. This line of code replaces Line 7 in the earlier 3-stage pipeline example.

```
1  dag += ( gen >> f * 4 ) >> p * 0;
```

ARMQ extends *RaftLib* with a number of runtime hints. These hints provide metadata that assists *ARMQ* with optimizing allocation and scheduling of the application *DAG*. The actual usage of the hints is determined by the underlying execution scheme; hints are safely ignored if they are not of usage to a given scheme. Listing 7 shows the aforementioned simple application example (see Listing 6) augmented with two types of hints used by *ARMQ*. These hints are used to indicate to the runtime a kernel rate multiplier (using the `*` operator overload) and for grouping heavily communicating/related kernels using parenthesis. The rate multiplier is an estimation of how many parallel workers may be needed to match the throughput of the upstream paths. As an example, `f * 4` means the `Filter` kernel runs likely 4× slower than the `Generator` kernel. On the other end of the spectrum a multiplier of zero would indicate an upstream filtering effect where we would expect this kernel to run fewer times for a given rate and therefore likely need a lower priority. The grouping hint informs the runtime that traffic between the indicated *DAG* partition is considered heavier (i.e., larger and/or higher frequency messages).

6.1.2 Partitioning

Before the application *DAG* is executed by *ARMQ*, it is analyzed by a partitioner. Two aspects considered in *ARMQ* partitioning are data locality and load balance. If the traffic (a product of message size and frequency) between two kernels is heavy, then it would likely be better to group the two so that they could be assigned to the same or clustered cores (i.e., sharing cache at some layers of the hierarchy). The application thereby increases the potential sharing of data within the cache memory

hierarchy, taking advantage of data locality. Managing shared resource utilization is synonymous with load balancing. If two kernels are heavy users of a shared resource (e.g., CPU core, cache and memory bandwidth), then it would likely be better to distance these kernels in order to avoid contention and the potential for hardware resource starvation.

Because *ARMQ* partitions *DAG* statically at the time of execution, there is only the topology and the message size information available. When confronted with pointers within the message queue, the true size of this indirect buffer is often hard to determine without further information (indirect buffers could even have further nested indirect buffers), which is one place where the hints provided by *ARMQ* (§ 6.1.1.1) can have an impact. When statically partitioning for load-balance, *ARMQ* makes the assumption that parallel tasks of the same kernel type will likely require the same resource types, and therefore *ARMQ* implicitly tries to distribute tasks of the same kernel to different cores.

6.1.3 Allocation

The *ARMQ* runtime allocator manages the memory where buffers are stored.

An obvious approach to implementing an allocator is to allocate buffer from heap whenever a message needs to be stored. This approach is flexible and portable. An added advantage is that message to cache line alignment can be customized on a per-message basis, reducing the potential of false sharing. This approach simply wraps the system-provided memory allocation library to serve the message queue interface. The downside is that the system allocator does not know the message queue usage pattern, and therefore some optimizations opportunities could be lost.

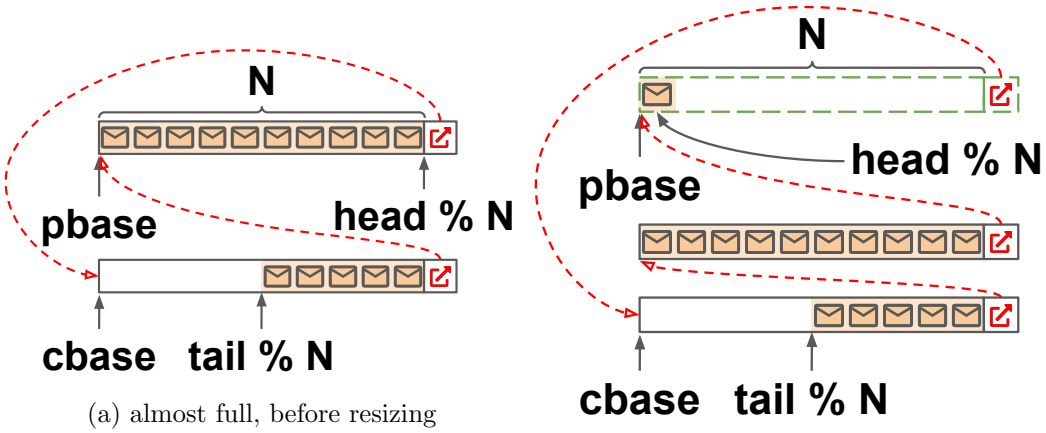
One common optimization is to pre-allocate a chunk of large enough memory and use it as a ringbuffer. New messages would enqueue into a slot in the ringbuffer where pointed by the head pointer and move the head pointer forward. On the other

side, consumers dequeue messages from the tail of the ringbuffer. With this approach, the same buffer space would be repeated used in the sequential order, so the better spatial locality would improve cache performance. However, there are two challenges: 1) Multiple producers or multiple consumers might have contention on updating the queue status concurrently; 2) when running out of buffer space, resizing the ringbuffer size could be a costly operation.

RaftLib [14] implements a single-producer-single-consumer (*SPSC*) ringbuffer to lower the contention. Multi-producer or multi-consumer message queues are emulated by letting the threads select data from the set of *SPSC* queues connected to it in round-robin order. *RaftLib* also has an option to launch a buffer monitoring thread. This thread monitors each ringbuffer to check for blocking over a window of time, doubling the capacity of the buffer (with limits) when needed (e.g., if kernels are blocked on the buffer for 75% of the time over the sampling window). Resizing in *RaftLib* could suffer from several overheads: 1) the monitoring thread needs to acquire exclusive access before resizing; 2) the monitoring thread have to copy the content in the old buffer (likely full) over to the new buffer. One could argue that the runtime should simply pick an arbitrarily large buffer, however, doing so (or also growing the ringbuffer indefinitely) can exhaust valuable system resources, lead to more paging and cache misses, and overall performance regression [14]. It can also be shown that choosing the exact ringbuffer size for a streaming system is a NP-Hard problem, this is known as the Buffer Allocation Problem [6]. Therefore, *ARMQ* extends the iterative approach to dynamic allocation adopted by *RaftLib*, and make it far more efficient.

6.1.3.1 Chunk-Based Zero-Copy Ringbuffer Resizing

ARMQ redesigns the ringbuffer to support zero-copy resizing. The ringbuffer that *ARMQ* uses is chunk-based, with each chunk holding up to N messages (where N is configurable) (see Figure 6.2 for an example). To facilitate resizing of the buffer, additional chunks can be added via memory pointers, forming a linked-list of chunks.



(a) almost full, before resizing

(b) new chunk, after resizing

Figure 6.2: Resizing chunk-based ringbuffer without copying. pbase/cbase is the pointer pointing to the chunk currently used by the producer/consumer, and head/tail is the monotonically increasing counter to index the slot where the producer/consumer enqueues/dequeues a message. Chunks are linked to form a ringbuffer.

The linked-list of chunks forms a loop where the last chunks points back to the start of the new chunk, making a resizble ringbuffer. For both producer and consumer, when they reach the end of a chunk, their base pointers (i.e., pbase, cbase in Figure 6.2) would be advanced to the next chunk using this link pointer. Because all chunks together forms a loop, the producer and consumer would repeatedly access the chunks as it is a ringbuffer.

Before a producer advances to the next chunk after filling up the current one, it also checks whether the ringbuffer is almost full (i.e., the tail is pointing to somewhere in the immediate next chunk, as shown in Figure 6.2a). To resize a almost full ringbuffer, the producer simply allocates a new chunk and updates the linking pointer (analogous to a linked-list, mid-link, insertion) then advances to the new chunk as shown in Figure 6.2b. This ringbuffer is meant to serve single producer and single consumer, and it synchronizes the local head/tail counters in batch/chunk to reduce cache bouncing [54]. Unlike the dynamically-resizeable ringbuffer in *RaftLib*, the *ARMQ* design avoids copying when resizing and this resizing strategy is applicable to non-trivially-copyable data types [2] (e.g., `std::string`).

Other than those memory-heap-based queuing options, *ARMQ* is also extensible to use shared memory or hardware queues.

6.1.4 Scheduling

As observed in Figure 1.6, enqueue and dequeue operations block on a full or empty queue. Blocking can arise from two conditions. The first condition (that of buffer sizing), was covered in § 6.1.3. The second condition is that of a rate mismatch between producer and consumer kernels. Queuing theory [49] states that the consumer must have a throughput greater than the producer (or equal to if all rates are perfectly deterministic) to ensure a bounded queue depth (e.g., that the queue is not always full). Many real software-based dataflow systems have unbalanced flows; indeed, even when programmers attempt to design perfectly deterministic systems they find that execution varies in unexpected ways [13]. *ARMQ* uses scheduling optimizations to modulate the throughput of a computation kernel, thereby mitigating cycle-wasting blocking behavior induced by rate-mismatch.

When enqueuing to a full queue, a condition that would result in blocking (and wasted cycles), the kernel/task producing data can take one (or more) of the actions shown in Figure 6.3. Actions ***poll*** and ***yield*** are the most general actions, and the most seen. When the hardware running the tasks is over-subscribed (often the case for data-center systems) *poll* alone could cause deadlock [40]. To prevent this, it is often necessary to *yield* after *poll* to allow other tasks to make forward progress. Nevertheless, *yield* does not convey a very strong signal to the scheduler that this task is blocked. A downside of this approach is that the task could cycle through *poll* and *yield* conditions indefinitely, without performing useful computation. A technique to prevent this from happening is to use a condition variable so that the scheduler has enough information to know when conditions are correct for this task to perform useful work, enabling the scheduler to safely exclude this task from running (e.g., it

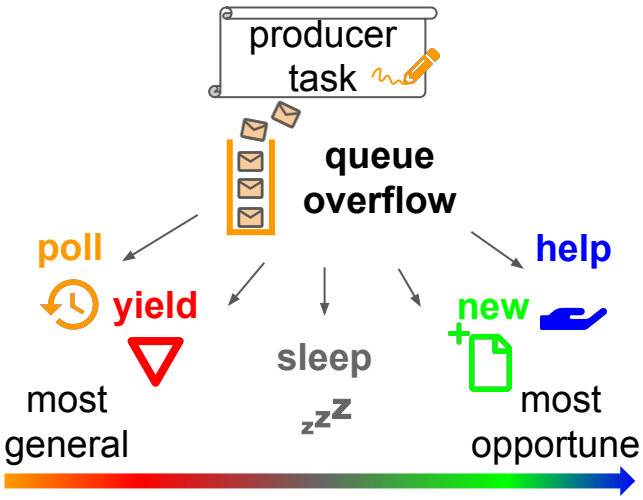


Figure 6.3: Actions a producer task might take when enqueuing to a full queue: 1) *poll*: continuously check whether the queue has free space or not; 2) *yield*: lower the priority been scheduled; 3) *sleep*: not been considered in scheduling until waking up upon condition change; 4) *new*: enlarge the buffer to fit in the overflowed message; 5) *help*: schedule a helper task to perform the computation defined by the consumer kernel.

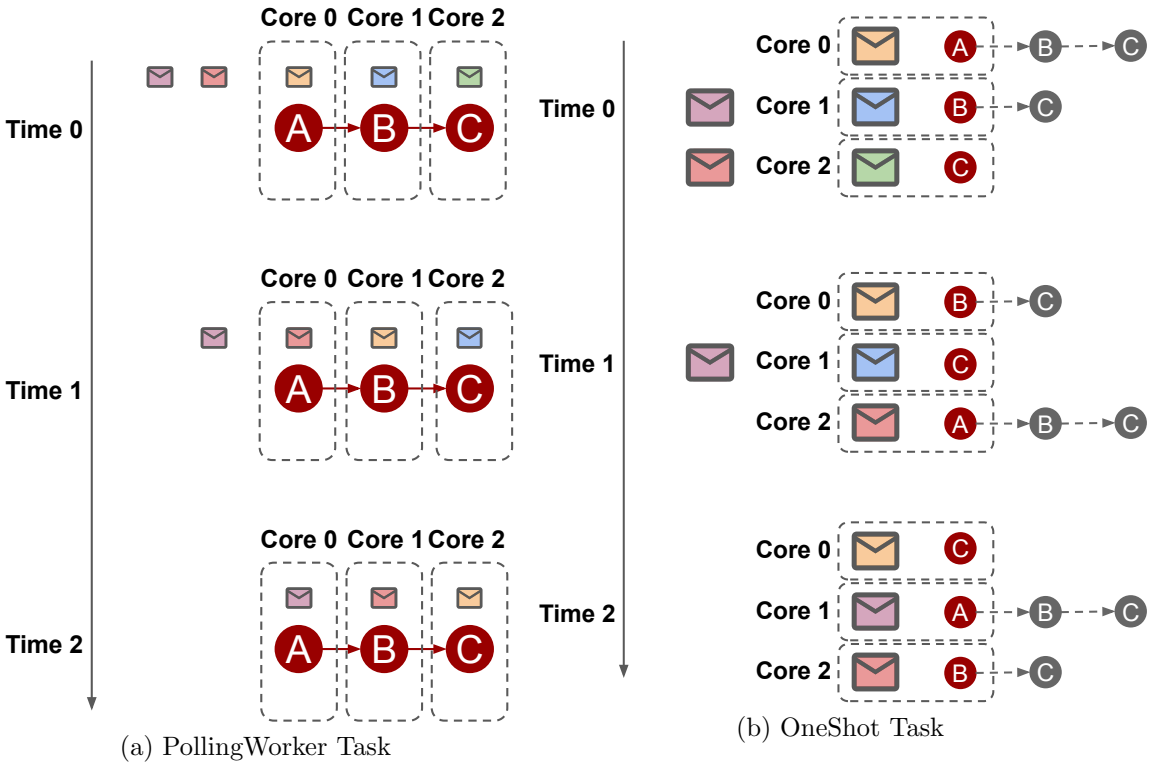
is ***sleep***, until conditions are correct to make forward progress). Such a exclusion method relies on passing information to the scheduler from the application, making scheduling more precise. Condition variables are the first example we provide of proactive steps the kernel can take to improve scheduling.

There are two additional proactive steps to reduce cycle-wastage when blocking on enqueue: ***new*** and ***help***. Instead of rescheduling the compute kernel when an equeue is blocked due to buffer exhaustion, the *new* action allocates a larger buffer to hold the overflowing message, thereby preventing blockage and reducing the probability of blocking in the future. *RaftLib* implemented buffer resizing, however, *ARMQ* provides much more efficient zero-copy resizing (see § 6.1.3). On the other hand, the action *help* unblocks the producer via changing the running task itself to be the consumer, on the same core that was previously executing the producer. This has the advantage of consuming queue elements (thereby emptying it) plus it takes advantage of data-locality (e.g., recently produced elements are assumed to be closer to

the producing core within a cache hierarchy).

While this section has focused on the producer side, the consumer task could block when the queue is empty on dequeue. The consumer task can take similar actions to the producer such as *poll*, *yield*, or *sleep* to make scheduling more precise.

6.1.4.1 Tasks



(a) PollingWorker Task

(b) OneShot Task

Figure 6.4: PollingWorker tasks vs. OneShot tasks.

Kernels are defined by users (§ 6.1.1) and focus purely on the computation, while *ARMQ* will internally creates tasks (computation plus data/messages, Listing 8 Line 1–4) to carry out the computations. There are different tasks types in *ARMQ*, each are distinguished by their computation and scheduling patterns. All tasks, do have one thing in common, they each have an associated computation kernel (§ 6.1.1)

and streaming data (either the actual data or a queue). The distinction between task types are their execution logic. The PollingWorker task is the long-last task that performs the same series of computations on a sliding window of streaming data, and yields after certain number of iterations (to avoid deadlock as discussed earlier). A OneShot task is initially designed to compute just once on a set of data and vanish, hence the name OneShot. An improved version of OneShot task reloads the task structure with a downstream consumer kernel, so that it can operate on the data that was just produced. The reloading continues until the OneShot task reaches a sink node in the computation graph (i.e., no output), then the OneShot task would be destroyed. This "run-to-completion" optimization reduces the number of task creation. As illustrated by Figure 6.4, a PollingWorker task repeats the same computation on different messages, while a OneShot tasks performs different computation on the "same" data buffer. Listing 8 presents the simplified version of task definitions in *ARMQ*.

6.1.4.2 Mix Scheduling

ARMQ starts with a basic scheduler design where each kernel in the programmer specified application topology is set as a basic PollingWorker task (the baseline runtime *RaftLib* [14] follows this same pattern). *ARMQ* augments this basic pattern with hints to assist the basic scheduler in assigning computation (see § 6.1.1.1). Assuming programmer estimates the throughput ratio accurately, all PollingWorker tasks get data to compute almost every iteration. Otherwise if blocking occurs, the PollingWorker schedule suffers from blocking overheads.

Alternatively, a OneShot scheduler creates OneShot tasks for the source kernels, and lets them run to completion. There would be no blocked producer during the execution, however, it would be too frequent to create an OneShot task for every message and the scheduling overhead on task creation becomes a concern.

Listing 8 Different *ARMQ* tasks.

```
class Task { /* is defined by computation/kernel & data  */
  Kernel *kernel;
  StreamingData dataIn, bufOut;
}; /* end of Task definition  */
class PollingWorker : public Task {
  void exe() {
    while (! shouldExit()) { /* when tearing down DAG */
      if (dataReady()) { /* input queue has data */
        kernel->compute(dataIn, bufOut);
      }
      if (loopedNTimes()) { yield(); } /* no dead lock */
    }
  }
};
class OneShot : public Task {
  void exe() {
    while (! isSink(kernel)) { /* run-to-completion */
      kernel->compute(dataIn, bufOut);
      reload(); /* update kernel, move output as input */
    }
  }
};
```

In order to avoid blocking or paying too much scheduling cost, *ARMQ* proposes a mix scheduling strategy. The mix scheduling begins with PollingWorker tasks only. The multiplier hints (§ 6.1.1.1) guide the mix scheduler as to how many PollingWorker tasks to create per kernel. Please note a difference between the basic scheduler and the mix one is that mix scheduling does not spawn a PollingWorker task for kernels having zero multiplier hint, while the basic scheduler will still create one PollingWorker task. Zero multiplier indicates likely there is no data in the incoming queue, so mix scheduling skips the kernel to reduce the blocking on empty queues. During the execution, if a PollingWorker task generates an outgoing message but is blocked on enqueue, the *mix* scheduler creates a OneShot task instead, enabling the scheduler to switch to the consumer OneShot immediately on blocking. Given the producer and consumer task are now run consecutively, and on the same core, there should be fewer data cache misses when accessing messages from the producer.

6.1.4.3 Userspace Threading Library: *libut*

Unlike the basic scheduling that spends one-time cost on creating PollingWorker tasks and switches tasks only after certain rounds of polls, the *mix* scheduling (§ 6.1.4.2) invokes task creation and switching more frequently, so it is especially important to keep the task management as low-latency as possible. In addition to the "run-to-completion" optimization (§ 6.1.4.1) that reduces the number of OneShot tasks created, *ARMQ* also applies lightweight user-space threading for fast task creation and context switching. To that end, a user-space threading library called *libut* is developed on top of the customized threading library from *Shenango* [66]. The *Shenango* threading implementation employs many techniques to push the threading overhead down to nanosecond-level on a 2.2 GHz machine:

1. *Shenango* makes use of Thread-Local-Storage (TLS) to manage pre-allocated thread cache for userspace task stacks that are allocated and free frequently;

2. Huge pages are used to reduce Translation-Lookaside-Buffer (TLB) misses;

3. It does not waste time on saving those registers that are safely clobbered;

4. It avoids atomic memory accesses for wake-ups;

5. Local task queue per kernel-level-thread (e.g., *pthread*) to avoid contention;

6. It employs work stealing to balance the tasks among *pthreads*.

We port *Shenango* threading library from *x86* to *AArch64* after writing the counterparts of some low-level code, making *libut* able to run on both *x86* and *AArch64* servers. Additionally, *libut* adds several features specific to the needs of *ARMQ*:

1. *locality awareness*: the capability to understand cache and memory hierarchy for more efficient work stealing and locality aware scheduling;

2. *affinity*: setting task affinity to support the task grouping (§ 6.1.2);

3. *consumer-first*: Hoare-style condition variables and low-overhead task spawn followed by immediate execution to preserve more data locality (§ 6.1.4.2).

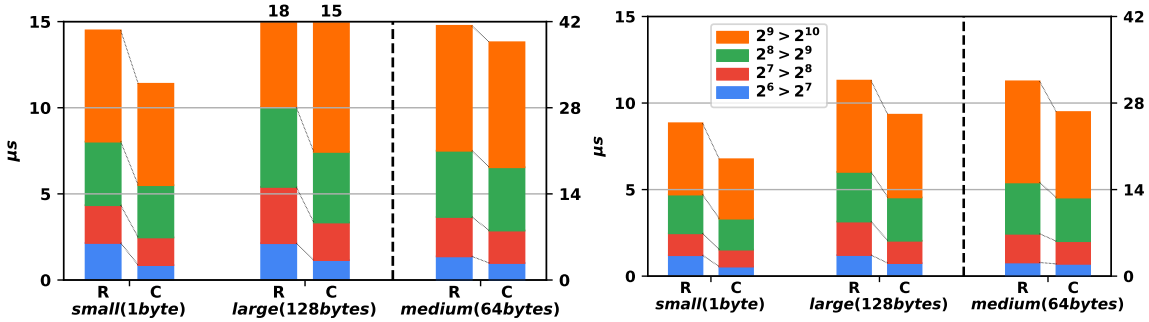
6.2 Evaluation

6.2.1 Zero-Copy Resizeable Ringbuffer

This section evaluates the ringbuffer resizing overheads with a microbenchmark. Please note that the microbenchmark is single-threaded and does no push or pop operation, while in real applications, the monitoring thread (which resizes full queues) in the baseline would suffer from extra overhead caused by the concurrent

push/pop operations. It is also worth mentioning that the chunk-based ringbuffer resizing latency reported in this section likely will overlap with the dequeue operations on the consumer side, thanks to the relaxed requirement on exclusiveness, whereas the baseline (*RaftLib* ringbuffer) pauses both the producer and consumer, and has to pay the the full overhead.

The microbenchmark tests ringbuffers of three different messages: the **small message** is a plain type with a data width of 1 B; the **medium message** is a class type with a data width of 64 B (i.e., cacheline size of the servers); and the **large message** is a C++ class type with a data width of of 128 B. When initializing, the microbenchmark creates 10^6 ringbuffers of a message type, with the initial capacity set to 64 entries. Then the microbenchmark resizes the ringbuffers, doubling the capacity iteratively until 1024. For every step doubling the capacity, the 10^6 ringbuffers of the same message type are resized all together for timing. The per-step per-ringbuffer resizing time is reported in Figure 6.5.



(a) Core-Coupled System A

(b) Two-Socket System B

Figure 6.5: Resizing latency comparison between baseline (i.e., the original *RaftLib* ringbuffer, marked by "R") and the chunk-based ringbuffer design in *ARMQ* (marked by "C"). Large message ringbuffers store message pointers (8 B).

Because the large message (128 B) does not fit into a cacheline, both the baseline runtime and *ARMQ* adaptively put the message pointers (8 B) in the ringbuffer instead of the messages. Therefore, the effective message size for the large message is in between the size of small and medium messages. The latency for large message

ringbuffer resizing turns to be closer to the small message ringbuffer, so they share the same left y-axis in Figure 6.5, while the medium bars use the secondary y-axis. As Figure 6.5a shows, the baseline (stacked bars in black and grey) always spend more time on doubling the capacity due to the copying overhead. Notably, *ARMQ* speeds up the resizing more when the size is smaller: increasing from $64 \rightarrow 128$ has the most speedup; small message ringbuffers shows the most speedup. This is because *ARMQ* has to make more invocations of the memory allocation functions to reach the designated capacity with fixed-length chunks, while the baseline only needs to allocate memory once per resizing.

6.2.2 Nano-Second-Level Userspace Threading

This section evaluates the performance of *libut* with some microbenchmarks from *Shenango* [66]. In each microbenchmark, a common threading operation is performed repeatedly for 10^7 iterations on a single CPU core. Please note the experiment does not scale up to multiple cores because in *ARMQ* task queues are local and the work stealing rarely, if not never, happens. The experiments measure how many nano-seconds elapse to finish those threading operations and calculate the per-operation average. Table 6.1 reports the results of *libut* and compares with three other threading options: 1) *pthread* is the de facto kernel-level threading library Linux provides; 2) *qthread* [92] is a light-wight locality-aware userspace threading library used by *RaftLib*; 3) the Go programming language [1] is designed to have built-in concurrent threading support. The original *Shenango* implementation is not included in the comparison because it does not support the *AArch64* instruction set architecture, extending the library to do so in the form of *libut* is one of *ARMQ*'s contributions.

As we can see from Table 6.1, *libut* has the lowest latency for three out of four threading operations: yielding a thread for a voluntary context switch; waking up a thread waiting for a condition variable; spawning threads to join. Go has lower

Table 6.1: Threading operation latency comparison between pthraed, qthread, go and *libut*. Numbers are nanoseconds per operations. *libut* is the only one finishing all different common threading operations within 1 µs.

	pthread	qthread	Go	*libut*
Uncontended Mutex	56	334	35	63
Yield Ping Pong	948	2,239	240	127
Condvar Ping Pong	4,184	N/A	512	243
Spawn-Join	38,984	5,075	1,098	415

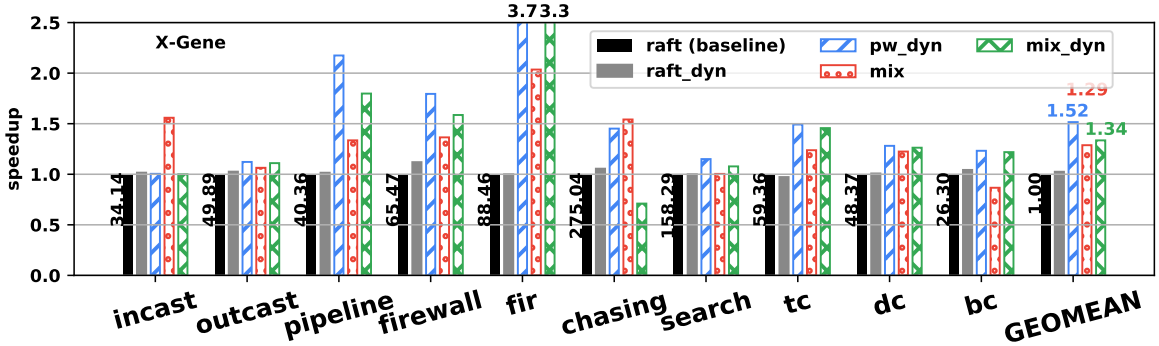
latency mutex operations thanks to the inline optimization done by the compiler [66]. Overall, only *libut* is able to keep all threading operations below 1 µs.

The low-overhead threading support from *libut* allow *ARMQ* to explore scheduling options at blocking, such as spawning OneShot tasks to help draining the pipeline and obtain performance gains. Without the userspace threading design, the scheduling optimizations of *ARMQ* would be offsetted.

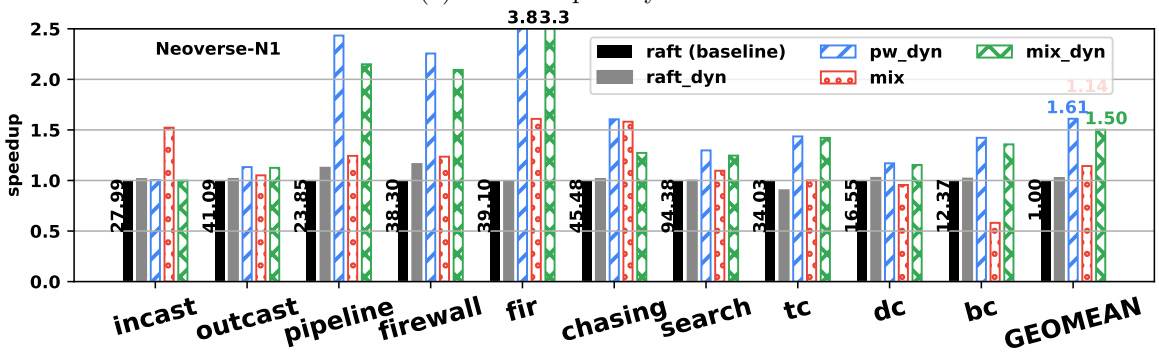
6.2.3 Speedup

Figure 6.6 presents the speedup achieved by different runtime schemes over the baseline (i.e., original *RaftLib* with fixed-size queue, black bars labelled as "`raft`"). Another variant of *RaftLib* that provides dynamically resizing ringbuffer support (grey bars labelled as "`raft_dyn`") is plotted in the same figure in order to demonstrate relative performance of *ARMQ*'s dynamically resizable ringbuffer implementation (§ 6.2.1). Three other schemes from *ARMQ* are PollingWorker scheduling with dynamically ringbuffer resizing (blue bars labelled as "`pw_dyn`"), PollingWorker mixed with OneShot scheduling (orange bards labelled as "`mix`"), and `mix` scheduling with dynamic ringbuffer resizing (green bars labelled as "`mix_dyn`").

As we can see in Figure 6.6a, the dynamic ringbuffer allocator in the original *RaftLib* has very limited performance improvement on few benchmarks (i.e., *firewall*, *chasing*, and *bc*). As described in § 6.2.1, the *RaftLib* ringbuffer must acquire exclusive



(a) Core-Coupled System A



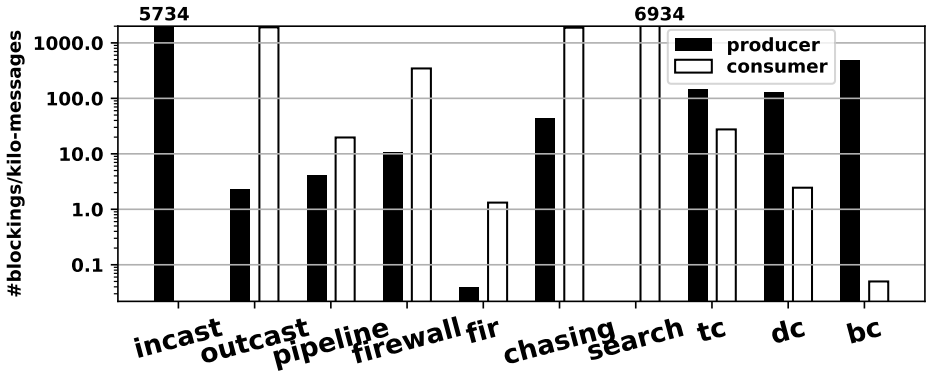
(b) Two-Socket System B

Figure 6.6: Speedup of different runtime schemes over the baseline runtime (i.e., the original *RaftLib*). Each baseline bar is labeled with execution time in seconds for reference.

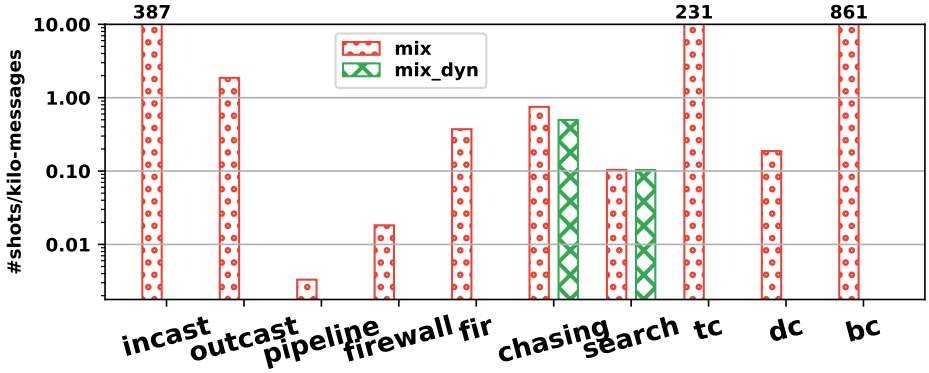
access over the target ringbuffer; a condition that rarely occurs in practice (although for very long-running applications, this may not be a huge deficit). In contrast, *ARMQ*'s implementation removes the need for this, instead having the producer thread to add an additional buffer chunk to the linked-list ringbuffer when the queue would otherwise be full (instead of blocking). For our benchmarks, *ARMQ* achieves an appreciable performance gain when resizing is enabled. On average, the speedup for `pw_dyn`, and `mix_dyn` is about 1.52× and 1.34×, respectively.

Mix scheduling, even with no dynamical resizing, is able to help addressing blocking as well. Figure 6.6a shows that benchmarks benefiting from from mix scheduling usually have structures like incast (i.e., *incast*) or long pipelines (i.e., *FIR*) or both (i.e., *chasing*). The rationales behind this are that: 1) the incast/fan-in pattern has more producers than consumers and it is more likely to have producer blocking and OneShot tasks would help; 2) long pipeline creates more starving Polling-Worker tasks that waste time on polling, while OneShot tasks always occupy cores with useful computation and have better data locality (as all the stages of the long pipeline are executed in one place). On other hand, creating too many OneShot tasks may lead to load imbalance and hurt the performance, which is observed in *bc*. The geometric mean of speedup achieved by "`mix`" over the baseline across all benchmarks is about 1.29×. Figure 6.6b shows on System B, *ARMQ* dynamic-resizing enabled schemes (i.e., "`pw_dyn`", and "`mix_dyn`") achieve even higher speedups on some of the benchmarks, while the benefit of mix scheduling is weaken a bit. One more observation from Figure 6.6 is that the combining of mix scheduling with dynamic ringbuffer (i.e., "`mix_dyn`") resizing does not yield a speedup higher than dynamic ringbuffer resizing alone (i.e., "`pw_dyn`"). This is likely because with dynamic ringbuffer resizing, blocking on full queue would never happen (Figure 6.7b) while "`mix_dyn`" still pays the cost to check the condition whether spawn oneshot tasks or not.

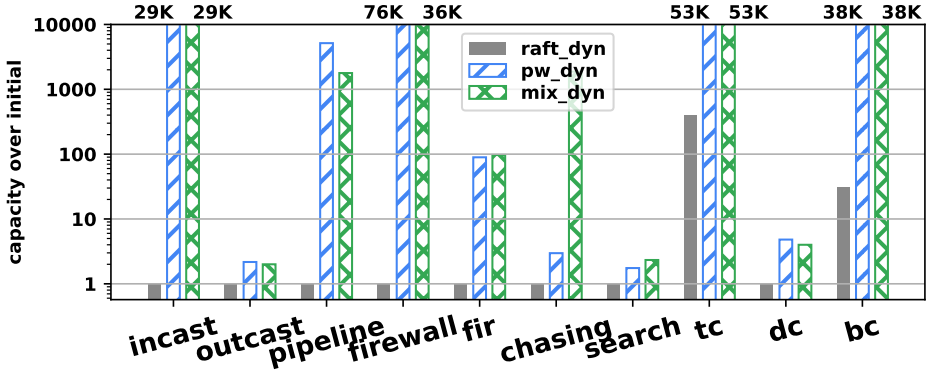
6.2.4 Statistics



(a) Blockings per 1000 Message in `raft (baseline)`



(b) OneShot Tasks per 1000 Messages



(c) Ringbuffer Capacity over Initial Allocation

Figure 6.7: Statistics of blockings, OneShot tasks and ringbuffer capacity.

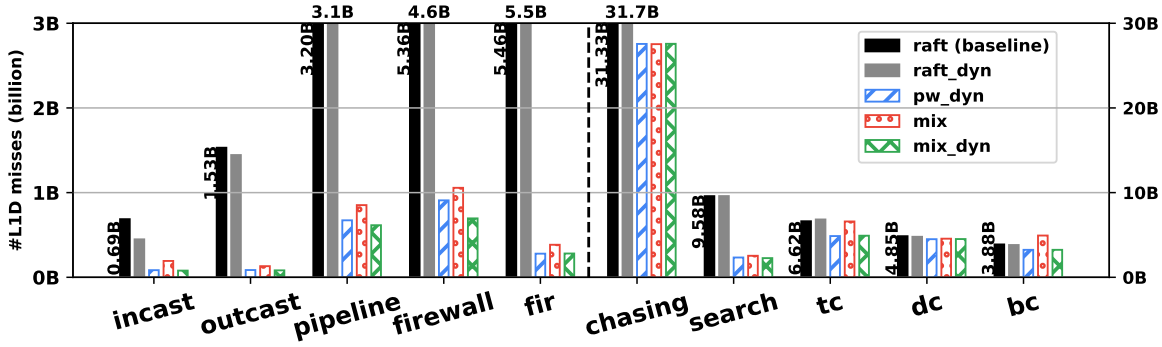
Figure 6.7a reports the statistics of how often producer tasks and consumer tasks are blocked in the baseline runtime. As we can see, blocking on producer enqueue and consumer dequeue exist in most benchmarks. On average, every message passed would experience blocked at least once. This frequency of blockage implies that considerable execution cycles are wasted (recall from § 6.1.4 that existing strategies to deal with stalls often do not contribute to forward progress of the application).

Mix scheduling avoids blocking via spawning OneShot tasks. Figure 6.7b shows how many OneShot tasks mix scheduling (i.e., `mix`, `mix_dyn`) issues out of every 1000 messages. The difference between `mix` and `mix_dyn` is that queues in `mix_dyn` never get filled up, so `mix_dyn` will only spawn OneShot tasks for kernels having no PollingWorker tasks (i.e., marked by zero-multiplier hint). For instance, the print stage after the search stage in *search* have relatively low chance to execute, but contributes many consumer blockings (Figure 6.7a), so zero-multiplier is added to the print kernel, then `mix_dyn` spawns OneShot tasks in *search*. It is similar for *chasing* (the pipeline stages after filter are marked by zero-multiplier hints) except the fan-in structure in *chasing* occasionally triggers producer blockings, so `mix` would spawn more OneShot tasks than `mix_dyn` on *chasing*. Although *libut* makes the cost of task creation very low (§ 6.2.2), it is also observed that spawning OneShot tasks too frequently leads to performance degradation: about 86% of the message in *bc* is processed by OneShot tasks (Figure 6.7b), and *bc* is the only one that `mix` runtime scheme is slower than the baseline (Figure 6.6).

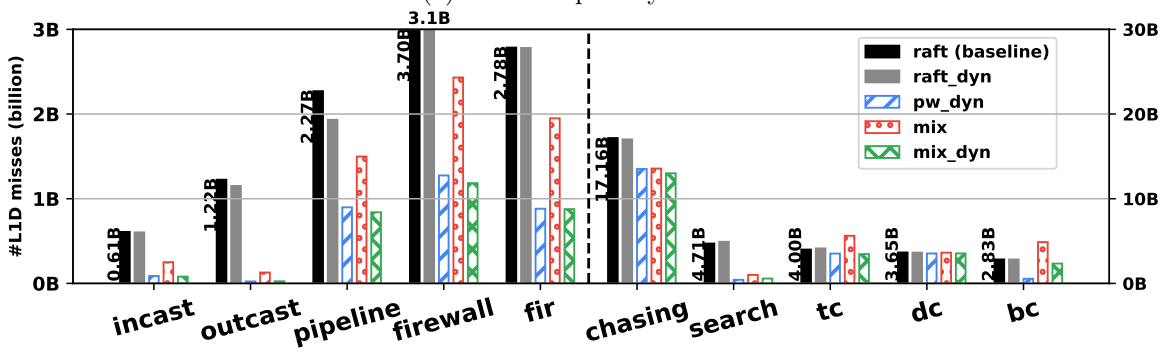
Dynamically resizing ringbuffer is another approach that *ARMQ* takes to avoid blocking. Figure 6.7c presents the ratio of resized ringbuffer capacity when benchmarks finish over the initial buffer allocation. Unlike the baseline (i.e., `raft_dyn`) that is only able to perform resizing on *tc* and *bc* (where tasks are relatively more coarse-grain), *ARMQ* gets ringbuffers resized in every benchmark. Benchmarks such as *pipeline* and *firewall* have inline (vs. in-buffer pointers), non-trivially-copyable message types, so `raft_dyn` is not able to resize the ringbuffers; this is not an issue for the *ARMQ* link-list-based ringbuffer. Resizing allows producer PollingWorker tasks

to get rid of blocking and finish earlier. If the cores freed up by earlier-terminated producer tasks are utilized to process remaining messages, the overall performance will be improved (e.g., *pipeline*, *firewall*, *FIR*), otherwise the performance remains the same (like *incast*) but the CPU utilization would go down; thereby allowing external global schedulers (e.g., those like GhOSt [45]) to schedule other applications.

6.2.5 Cache Performance



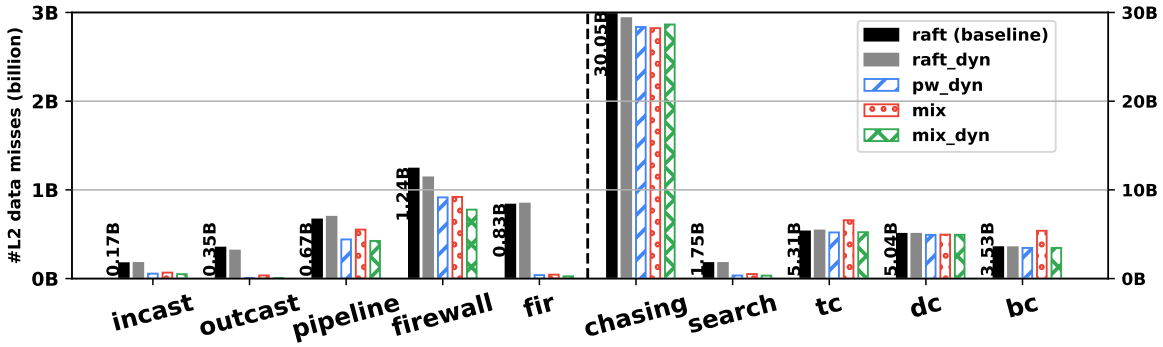
(a) Core-Coupled System A



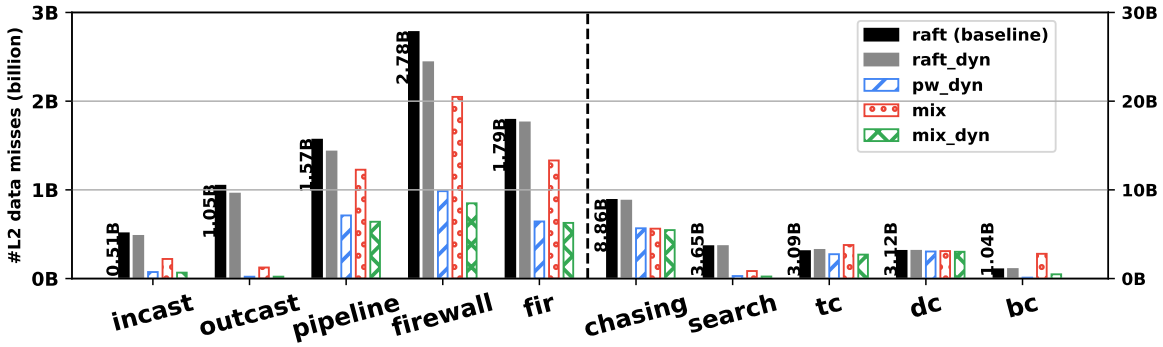
(b) Two-Socket System B

Figure 6.8: *L1D* cache misses of different runtime schemes. For the purpose of visibility, the left 5 benchmarks and the right 5 benchmarks use different scales.

Modern cache-heavy memory hierarchies are optimized for data reuse [89, 74, 77]. To take advantage of these hardware structures often means not only within thread reuse but data sharing between cooperative threads, e.g., data locality. *ARMQ* aims to improve data locality between kernels in the execution *DAG*, this section

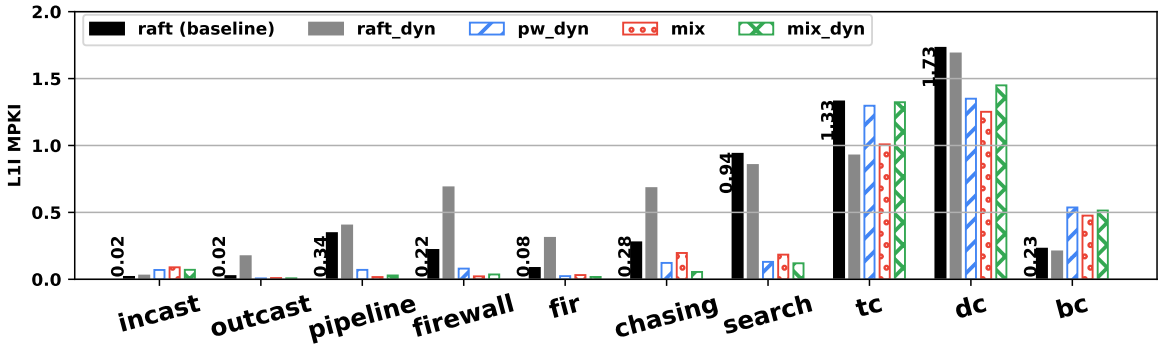


(a) Core-Coupled System A

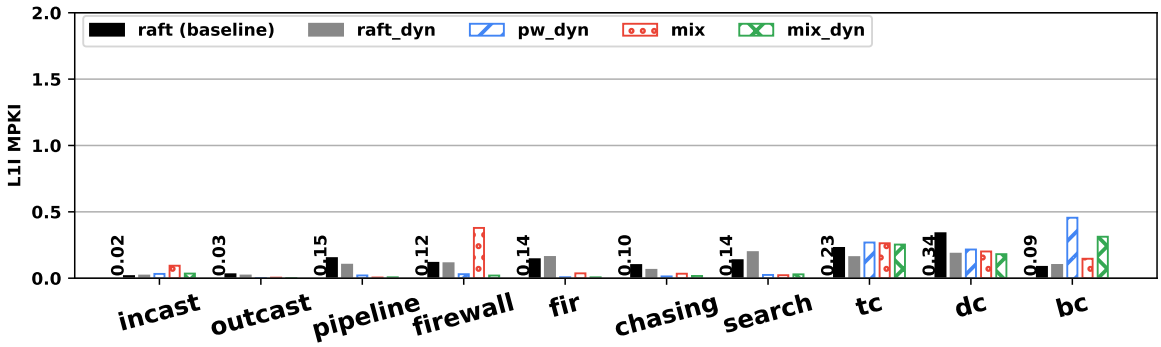


(b) Two-Socket System B

Figure 6.9: *L2* cache data misses of different runtime schemes. For the purpose of visibility, the left 5 benchmarks and the right 5 benchmarks use different scales.



(a) Core-Coupled System A



(b) Two-Socket System B

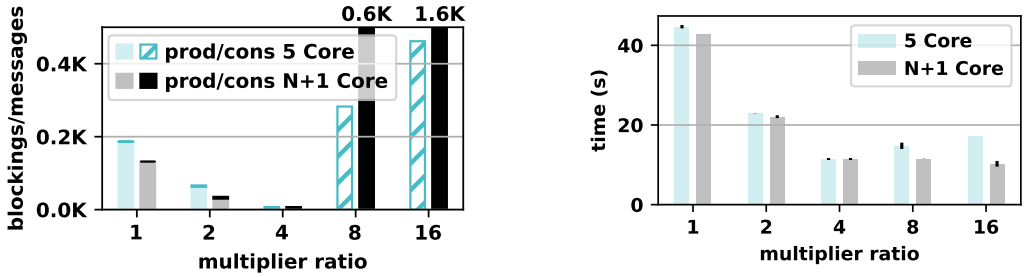
Figure 6.10: *L1I* cache Misses Per Kilo-Instructions (MPKI) of different runtime schemes.

evaluates if *ARMQ* hits the mark.

The data shown in Figure 6.8 and 6.9 suggest that *ARMQ* significantly reduces the count of overall *L1D* and *L2D* misses across many benchmarks. On average (geometric mean), the *L1D*, *L2D* cache miss reduction obtained by `pw_dyn` on System A are about 40%, and 17%, respectively. It is likely that *ARMQ*'s locality enhancement techniques are responsible for this decrease, a correlation that we will investigate further in § 6.2.6 in order to attribute causation. Regardless of the cause, fewer cache misses correlate with the performance speedup of *ARMQ* (§ 6.2.3). For example, *FIR*, *pipeline*, and *firewall* demonstrate the greatest execution time reduction with *ARMQ* (`pw_dyn`) while also exhibiting significant overall cache misses reduction.

When spawning and executing OneShot tasks, the same kernel-level thread (kthread) in *ARMQ* switches from executing the producer `compute()` function to the consumer ones. This "moving compute to data" approach trades instruction locality for data locality. One question that naturally arises is that whether the more frequent task switching in the mix scheduling causes negative impact, and if so, how severe it is? To address this concern, Figure 6.10 reports *L1I* cache Misses Per Kilo-Instructions (MPKI). On system A (Figure 6.10a), most benchmarks have instruction cache MPKI lower than 0.5 in the baseline (`raft`), except *search*, *tc*, and *dc*. Surprisingly, only on *incast* and *bc*, *ARMQ* gets higher *L1I* cache MPKI than the baseline, and *ARMQ* lets none of the benchmarks' *L1I* cache MPKI exceed 1.5. This is likely because the branch predictors and instruction preftechers [30] in modern processors are sophisticated enough to deal with those tasks having static dependencies. On System B (Figure 6.10b), all benchmarks have very low instruction cache MPKI (under 0.5) regardless runtime scheme. Therefore, although mix scheduling involves more frequent task switching in scheduling, the performance impact on instruction cache is likely negligible.

6.2.6 Case Study



(a) Blocking

(b) Time

Figure 6.11: The impact of multiplier ratio on blocking and execution time. Blocking bars are broken down between producer (bottom) and consumer (top). The little black sticks on the bars in the time chart indicates standard deviation. From both blocking and time perspectives, 4 is the best multiplier ratio because the throughput ratio of the two stages in the microbenchmark is close to 4.

Multiplier Hint: To demonstrate how multiplier hints affect the performance, we conduct a case study with a 2-stage microbenchmark similar to *outcast*. The firing rate of the first stage (the producer) is about 4 times as the firing rate of the second stage (the consumers). On the second stage, we apply a multiplier hint, which varies from 1 to 16. Only PollingWorker tasks are used, otherwise OneShot tasks could augment the multiplier ratio. There are two settings of CPU cores in the case study: either with limited core count (i.e., 5 cores, bluish bars in Figure 6.11) or unlimited (i.e., N+1 cores) cores, making sure every task running on its own core. As shown in Figure 6.11a, the producer blocks less frequently when the multiplier ratio increases, because there are more consumers matching up the throughput of the producer. However, if the number of the consumers is increased over 4, starving consumers starts competing for limited CPU cores with others. The contention causes performance regression in Figure 6.11b. When allowing core count scales along with the number of consumers, we observe the execution time remains stably low after multiplier ratio is increased over 4, but there are many more consumer blockings, indicating the CPU cores are actually utilized in a wasteful way.

6.3 Summary

In conclusion, this chapter presents *ARMQ*, a message queue runtime system, where applications could test different strategies to handle message queue blocking and find the most suitable one. *ARMQ* reduces the overhead of resizing ringbuffers with a chunk-based ringbuffer design, and lowers the scheduling overhead via a customized userspace threading library. By taking advantages of application hints and system topology info, *ARMQ* groups tasks to keep the locality of heavy message traffic. *ARMQ* proposes a scheduling policy that mixes polling with OneShot helper threads to avoid blocking on full queues and to improve the data locality. The evaluation shows *ARMQ* outperforms the baseline up to 3.8×.

Chapter 7: Conclusion

Over the decades, the massive deployment has proven the success of multi-core systems. In the foreseeable future, computing systems are likely going to continue the trend of adding more Processing Elements (PE). Meanwhile, many emerging workloads (e.g., data analytics, machine learning) are both data-intensive and computation-intensive. How to accelerate these workloads with growing multi-core systems is a challenging problem. Message queue task parallelism (or task data flow) computing paradigm serves as a great harness that allows us to drive parallelizable computation tasks in those emerging workloads on different PEs simultaneously. Message queue task parallelism is also good at dealing with huge volume of data, which could be sliced, packaged in message to go through a pipeline-style processing.

However, message queue task parallelism is not always able to achieve an ideal speedup as stated by Amdahl's Law [39] due to some inefficiencies at architecture and system level. The first issue is the concurrent accesses to the shared queue states incur coherence overhead, and the coherence traffic will increase along with the number of cores, limiting the scalability. Second, there would be delay from requesting a message to getting access when messages are buffered in multiple local storage. For example, the private L1 cache in modern multi-level cache hierarchy is local to each core. In addition, message-driven tasks could be blocked when queues run out of buffer or get drained entirely. Leaving the tasks blocked wastes CPU cycles, while schedule other task might have to pay extra overheads and disturb data locality. This dissertation focuses on these issues in message queue task parallelism, and makes contributions as summarized in the following section.

7.1 Summary

The dissertation proposes architectural supports to enhance the scalability of message queue task parallelism, and to reduce latency as well as blocking on queue operations.

The first contribution of the dissertation is *Virtual-Link* [94] scalable message queue architecture. *Virtual-Link* augments multi-core architecture with an in-network routing device. The routing device manages the shared queue states (like head, tail) for threads using message queues, so that there is no concurrent access to a cacheline for queue operations any more. In this way, *Virtual-Link* reduces the cache coherence traffic, and enhances the scalability of message queue task parallelism. Additionally, *Virtual-Link* utilizes the on-chip resources (i.e., interconnect and storage) to accelerate the cross-core communication.

Another contribution of the dissertation is *SPAMeR* [95] producer-driven data movement speculation mechanism. *SPAMeR* comes up with a prediction algorithm that is easy to be integrated with message queue architectures. *SPAMeR* can adaptively learn how soon a consumer will request messages based on the history. This enables the buffer having the messages to speculatively pushes a message to the consumer ahead of time. The producer-driven speculation also helps to reduce the consumer request traffic, letting the consumers spend more time on processing the messages. The technique provided by *SPAMeR* overlaps data movement latency with message processing time, considerably improves the performance of message queue task parallel workloads, especially those workloads limited by consumer tasks.

Finally, the dissertation contributes *ARMQ* locality-aware runtime for message queue task parallelism. *ARMQ* integrates several scheduling strategies handling message queue blocking. To lower the overhead of creating and switching tasks, *ARMQ* employs a state-of-the-art userspace threading implementation, and optimizes it to preserve data locality between dependent tasks. To lower the overhead of resizing, *ARMQ* designs a chunk-based ringbuffer, which avoids copying and con-

tention. *ARMQ* is designed as a streaming-style template library, making it easier to be adapted by message queue task parallel workloads. There are also interfaces for programmers to provide *ARMQ* runtime with hints, which is combined with system topology information to take advantage of cache locality.

7.2 Future Works

This section explores some potential research directions to further support message queue task parallelism in the future.

Computing systems are not only increasing the number of cores, but also becoming more heterogeneous. Asymmetric multiprocessor (powerful big cores combined with power-efficient small cores) is commonly seen in modern *SoC* design [34, 56, 70]. Moreover, there are domain-specific accelerators/processing units, reconfigurable cores, gotten integrated into the processors [41, 75, 76, 26]. The initiatives [85, 24] to establish a standard of coherent interconnect between accelerators and CPUs make it possible to extend *Virtual-Link* for heterogeneous systems. While the question arises with this is how should cross-core synchronization accommodate the differences between cores, so that slower cores would not hinder faster cores.

There have been proposals that apply machine learning in hardware speculation, such as cache prefetcher [18, 88, 57], branch predictor [96, 46]. Given the accuracy achieved by these machine learning approaches, it worth trying to improve the producer-driven speculation mechanism in *SPAMeR* with similar machine learning models.

The scaling of message queue task parallelism is not limited within chip multicore processors. There are many task data flow applications deployed on large scale distributed systems [10]. Some of those systems are managed with hardware visualizations (e.g., virtual machine and container), which provides more elasticity in terms of giving more servers to applications when their load surges. This exposes more opportunities and challenges to schedule message queue task parallel workloads while

preserving locality [25, 32].

Works Cited

[1] The go programming language. URL https://golang.org.

[2] CPP copy_constructor, 2023. URL https://en.cppreference.com/w/cpp/language/copy_constructor.

[3] Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses: A microarchitectural perspective. *ACM Trans. Comput. Syst.*, 36(3), jun 2019. ISSN 0734-2071. doi: 10.1145/3319393. URL https://doi.org/10.1145/3319393.

[4] Sam Ainsworth and Timothy M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 132–144. IEEE Press, 2020. ISBN 9781728146614. doi: 10.1109/ISCA45697.2020.00022. URL https://doi.org/10.1109/ISCA45697.2020.00022.

[5] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: high-level and efficient streaming on multi-core (A FastFlow short tutorial) . In *Programming Multi-core and Many-core Computing Systems*, chapter 13. Wiley, 2011.

[6] Venkat Anantharam. The optimal buffer allocation problem. *IEEE Transactions on Information Theory*, 35(4):721–725, 1989.

[7] Apache. Apache storm, 2023. URL https://storm.apache.org/index.html. [Online; accessed 07-May-2023].

[8] *Revere-AMU System Architecture*. Arm Limited, September 2019. URL https://bit.ly/3kajJuQ.

[9] Timothy G. Armstrong, Justin M. Wozniak, Michael Wilde, and Ian T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 299–310, 2014. doi: 10.1109/SC.2014.30.

[10] Mutaz Barika, Saurabh Garg, Albert Y. Zomaya, Lizhe Wang, Aad Van Moorsel, and Rajiv Ranjan. Orchestrating big data analysis workflows in the cloud: Research challenges, survey, and future directions. *ACM Comput. Surv.*, 52(5), sep 2019. ISSN 0360-0300. doi: 10.1145/3332301. URL `https://doi.org/10.1145/3332301`.

[11] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629579. URL `https://doi.org/10.1145/1629575.1629579`.

[13] Jonathan C. Beard and Roger D. Chamberlain. Use of a Levy distribution for modeling best case execution time variation. In A. Horváth and K. Wolter, editors, *Computer Performance Engineering*, volume 8721 of *Lecture Notes in Computer Science*, pages 74–88. Springer International Publishing, September 2014. ISBN 978-3-319-10884-1. doi: http://dx.doi.org/10.1007/978-3-319-10885-8_6.

[14] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: A c++ template library for high performance stream parallel processing. *Interna-*

tional Journal of High Performance Computing Applications, 2016. doi: http://dx.doi.org/10.1177/1094342016672542.

[15] S. Bell, B. Edwards, J. Amann, R. Conlin, Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, Feb 2008. doi: 10.1109/ISSCC.2008.4523070.

[16] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *SIGOPS Oper. Syst. Rev.*, 23(5):102–113, nov 1989. ISSN 0163-5980. doi: 10.1145/74851.74861. URL `https://doi.org/10.1145/74851.74861`.

[17] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, may 1991. ISSN 0734-2071. doi: 10.1145/103720.114701. URL `https://doi.org/10.1145/103720.114701`.

[18] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. Perceptron-based prefetch filtering. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2019.

[19] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. A formal security analysis of even-odd sequential prefetching in profiled cache-timing attacks. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347693. doi: 10.1145/2948618.2948624. URL `https://doi.org/10.1145/2948618.2948624`.

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL `https://doi.org/10.1145/2024716.2024718`.

[21] boost. Class template queue. `https://bit.ly/37hAMHJ`, 2020. Accessed: 2020-08-19.

[22] Bérenger Bramas. Impact study of data locality on task-based applications through the heteroprio scheduler. *PeerJ. Computer science*, 5:e190, 05 2019. doi: 10.7717/peerj-cs.190.

[23] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, page 51–61, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897915348. doi: 10.1145/143365.143486. URL `https://doi.org/10.1145/143365.143486`.

[24] CCIX Consortium. *CCIX™ Consortium Enables Next Generation Compute Architectures with the Availability of Base Specification 1.0 [Press release]. URL https://tinyurl.com/2tdd9z96*, 2018. URL `https://tinyurl.com/2tdd9z96`.

[25] Andrei-Alin Corodescu, Nikolay Nikolov, Akif Quddus Khan, Ahmet Soylu, Mihhail Matskin, Amir H. Payberah, and Dumitru Roman. Locality-aware workflow orchestration for big data. In *Proceedings of the 13th International Conference on Management of Digital EcoSystems*, MEDES '21, page 62–70, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383141. doi: 10.1145/3444757.3485106. URL `https://doi.org/10.1145/3444757.3485106`.

[26] Vidushi Dadu and Tony Nowatzki. Taskstream: Accelerating task-parallel workloads by recovering program structure. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1–13, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507706. URL https://doi.org/10.1145/3503222.3507706.

[27] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. doi: 10.1109/99.660313.

[28] Andreas Diavastos and Pedro Trancoso. Auto-tuning static schedules for task data-flow applications. In *Proceedings of the 1st Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ANDARE '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450353632. doi: 10.1145/3152821.3152879. URL https://doi.org/10.1145/3152821.3152879.

[29] Andreas Diavastos and Pedro Trancoso. Switches: A lightweight runtime for dataflow execution of tasks on many-cores. *ACM Trans. Archit. Code Optim.*, 14(3), sep 2017. ISSN 1544-3566. doi: 10.1145/3127068. URL https://doi.org/10.1145/3127068.

[30] Babak Falsafi and Thomas F. Wenisch. *A Primer on Hardware Prefetching*. Morgan and Claypool Publishers, 2014. ISBN 1608459527.

[31] Reza Fotohi, Mehdi Effatparvar, Fateme Sarkohaki, Shahram Behzad, et al. An improvement over threads communications on multi-core processors. *arXiv preprint arXiv:1909.11644*, 2019.

[32] Alexander Fuerst and Prateek Sharma. Locality-aware load-balancing for serverless clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, page 227–239,

New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391993. doi: 10.1145/3502181.3531459. URL https://doi.org/10.1145/3502181.3531459.

[33] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 291–303, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135742. doi: 10.1145/605397.605428. URL https://doi.org/10.1145/605397.605428.

[34] P. Greenhalgh. big.little processing with arm cortex-a15 & cortext-a7. ARM White Paper, 2011.

[35] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 368–379, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978356. URL https://doi.org/10.1145/2976749.2978356.

[36] Y. Guo, V. Cave, V. Sarkar, and J. Zhao. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, Los Alamitos, CA, USA, apr 2010. IEEE Computer Society. doi: 10.1109/IPDPS.2010.5470425. URL https://doi.ieeecomputersociety.org/10.1109/IPDPS.2010.5470425.

[37] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1550–1550, Los Alamitos, CA, USA, may 2022.

IEEE Computer Society. doi: 10.1109/SP46214.2022.00121. URL https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00121.

[38] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 341–342, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588773. doi: 10.1145/1693453.1693504. URL https://doi.org/10.1145/1693453.1693504.

[39] John L. Gustafson. *Amdahl's Law*, pages 53–60. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_77. URL https://doi.org/10.1007/978-0-387-09766-4_77.

[40] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear. *The Art of Multiprocessor Programming*. Elsevier Science, 2020. ISBN 9780123914064. URL https://books.google.com/books?id=7MqcBAAAQBAJ.

[41] Mark D. Hill and Vijay Janapa Reddi. Accelerator-level parallelism, 2019.

[42] Pieter Hintjens. Zeromq: the guide. *URL http://zeromq.org*, 2010.

[43] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. Cpp-taskflow: A general-purpose parallel task programming system at scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(8):1687–1700, 2021. doi: 10.1109/TCAD.2020.3025075.

[44] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Trans. Parallel Distrib. Syst.*, 33(6):1303–1320, jun 2022. ISSN 1045-9219. doi: 10.1109/TPDS.2021.3104255. URL https://doi.org/10.1109/TPDS.2021.3104255.

[45] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Jack Turner, and Christos Kozyrakis. ghost: Fast and flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, page 588–604, New York, NY, USA, 2021. URL https://doi.org/10.1145/3477132.3483542.

[46] Siavash Z. Kamali. *Using Convolutional Neural Networks to Improve Branch Prediction*. PhD thesis, The University of Texas at Austin, Austin TX, 2022.

[47] Dongkeun Kim and Donald Yeung. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 159–170, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135742. doi: 10.1145/605397.605415. URL https://doi.org/10.1145/605397.605415.

[48] Dongkeun Kim and Donald Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Trans. Comput. Syst.*, 22(3):326–379, aug 2004. ISSN 0734-2071. doi: 10.1145/1012268.1012270. URL https://doi.org/10.1145/1012268.1012270.

[49] L. Kleinrock. *Queueing Systems. Volume 1: Theory*. Wiley-Interscience, 1975.

[50] Rakesh Krishnaiyer, Emre Kultursay, Pankaj Chawla, Serguei Preis, Anatoly Zvezdin, and Hideki Saito. Compiler-based data prefetching and streaming non-temporal store generation for the intel(r) xeon phi(tm) coprocessor. In *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, pages 1575–1586, 2013. doi: 10.1109/IPDPSW.2013.231.

[51] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*,

ISCA '07, page 162–173, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937063. doi: 10.1145/1250662.1250683. URL https://doi.org/10.1145/1250662.1250683.

[52] Eric Lau, Jason E Miller, Inseok Choi, Donald Yeung, Saman Amarasinghe, and Anant Agarwal. Multicore performance optimization using partner cores. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*, Berkeley, CA, May 2011. USENIX Association. URL https://www.usenix.org/conference/hotpar11/multicore-performance-optimization-using-partner-cores.

[53] Patrick P. C. Lee, Tian Bu, and Girish Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2010. doi: 10.1109/IPDPS.2010.5470368.

[54] Patrick P. C. Lee, Tian Bu, and Girish Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, 2010. doi: 10.1109/IPDPS.2010.5470368.

[55] Sanghoon Lee, Devesh Tiwari, Yan Solihin, and James Tuck. Haqu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 99–110. IEEE, 2011.

[56] Taehee Lee, Dongkeun Kim, and Joonseok Kim. Exynos 1080 high-performance, low-power cpu and gpu with amigo. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–14, 2021. doi: 10.1109/HCS52781.2021.9567394.

[57] Manel Lurbe, Josué Feliu, Salvador Petit, Maria E. Gómez, and Julio Sahuquillo. Deepp: Deep learning multi-program prefetch configuration for the ibm power 8. *IEEE Transactions on Computers*, 71(10):2646–2658, 2022. doi: 10.1109/TC.2021.3139997.

[58] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture newsletter*, 2(19–25), 1995.

[59] Niall McDonnell and Gage Eads. *Queue Management and Load Balancing on Intel® Architecture*. `https://intel.ly/3hY0Zy8`, 2020. Accessed: 2021-01-09.

[60] Gabriele Mencagli, Massimo Torquati, Dalvan Griebler, Marco Danelutto, and Luiz Gustavo L. Fernandes. Raising the parallel abstraction level for streaming analytics applications. *IEEE Access*, 7:131944–131961, 2019. doi: 10.1109/ACCESS.2019.2941183.

[61] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 267–275, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897918002. doi: 10.1145/248052.248106. URL `https://doi.org/10.1145/248052.248106`.

[62] Todd C. Mowry. *Tolerating latency through software-controlled data prefetching*. PhD thesis, Stanford University, Stanford CA, 1994.

[63] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. Graphbig: understanding graph computing in the context of industrial solutions. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015. doi: 10.1145/2807591.2807626.

[64] Ajeya Naithani, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout. Vector runahead. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 195–208, 2021. doi: 10.1109/ISCA52012.2021.00024.

[65] G. Ottoni, R. Rangan, A. Stoler, and D.I. August. Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 12 pp.–118, 2005. doi: 10.1109/MICRO.2005.13.

[66] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL `https://www.usenix.org/conference/nsdi19/presentation/ousterhout`.

[67] perf. perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`, 2023. Accessed: 2023-05-09.

[68] Steven J. Plimpton and Tim Shead. Streaming data analytics via message passing with application to graph algorithms. *Journal of Parallel and Distributed Computing*, 74(8):2687–2698, 2014. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2014.04.001. URL `https://www.sciencedirect.com/science/article/pii/S0743731514000884`.

[69] DPAA QorIQ. Primer for software architecture. Technical report, Technical report, Freescale Semiconductor Inc, 2012.

[70] Qualcomm. Snapdragon 888+5g mobile platform, 2021.

[71] T. Ramírez, A. Pajuelo, O. J. Santana, O. Mutlu, and M. Valero. Efficient runahead threads. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[72] Karl Rupp. Microprocessor trend data, 2022. URL `https://github.com/karlrupp/microprocessor-trend-data`.

[73] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, page 22–32, USA, 2011. IEEE Computer Society. ISBN 9780769545660. doi: 10.1109/PACT.2011.9. URL https://doi.org/10.1109/PACT.2011.9.

[74] Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. Data placement across the cache hierarchy: Minimizing data movement with reuse-aware placement. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 117–124, 2016. doi: 10.1109/ICCD.2016.7753269.

[75] Yakun Sophia Shao and David Brooks. Morgan & Claypool, 2015. ISBN 978-1-627-05832-2. URL https://ieeexplore.ieee.org/servlet/opac?bknumber=7347037.

[76] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358302. URL https://doi.org/10.1145/3352460.3358302.

[77] Fanfan Shen, Yanxiang He, Jun Zhang, Qingan Li, Jianhua Li, and Chao Xu. Reuse locality aware cache partitioning for last-level cache. *Computers & Electrical Engineering*, 74:319–330, 2019. ISSN 0045-7906. doi: https://doi.org/10.1016/j.compeleceng.2019.01.020. URL https://www.sciencedirect.com/science/article/pii/S0045790618307572.

[78] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 131–145, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243736. URL https://doi.org/10.1145/3243734.3243736.

[79] SHM. The open group base specifications issue 7, 2018 edition ieee std 1003.1-2017 (revision of ieee std 1003.1-2008). https://bit.ly/2Hfww0w. Accessed October 2020.

[80] sstsimulator. *Ember* communication pattern library, 2020. URL https://github.com/sstsimulator/ember.

[81] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of cmos device performance from 180nm to 7nm. *Integration*, 58:74 – 81, 2017. ISSN 0167-9260. doi: https://doi.org/10.1016/j.vlsi.2017.02.002. URL http://www.sciencedirect.com/science/article/pii/S0167926017300755.

[82] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal. Freepdk: An open-source variation-aware design kit. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, pages 173–174, 2007. doi: 10.1109/MSE.2007.44.

[83] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. Gramps: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1), feb 2009. ISSN 0730-0301. doi: 10.1145/1477926.1477930. URL https://doi.org/10.1145/1477926.1477930.

[84] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R. Nigel Horspool, editor, *Compiler*

Construction, pages 179–196, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45937-8.

[85] S. Van Doren. Abstract - hoti 2019: Compute express link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 18–18, 2019. doi: 10.1109/HOTI.2019.00017.

[86] Sevin Varoglu and Stephen Jenks. Architectural support for thread communications in multi-core processors. *Parallel Computing*, 37(1):26–41, 2011.

[87] VIRTIO. Virtual I/O Device (VIRTIO) Version 1.1. `https://bit.ly/3jaEqWf`. Accessed October 2019.

[88] Haoyuan Wang and Zhiwei Luo. Data cache prefetching with perceptron learning, 2017.

[89] Jiajun Wang. *Reuse Aware Data Placement Schemes for Multilevel Cache Hierarchies*. PhD thesis, The University of Texas at Austin, Austin TX, 2019.

[90] Yipeng Wang, Ren Wang, Andrew Herdrich, James Tsai, and Yan Solihin. Caf: Core to core communication acceleration framework. In *2016 International Conference on Parallel Architecture and Compilation Techniques*, pages 351–362. IEEE, 2016.

[91] Scott Wasson. *Inside ARM's Cortex-A72 microarchitecture*. `https://bit.ly/3sf0a9h`, 2015. Accessed: 2021-01-09.

[92] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008. doi: 10.1109/IPDPS.2008.4536359.

[93] Markus Wittmann and Georg Hager. A proof of concept for optimizing task parallelism by locality queues, 2009.

[94] Qinzhe Wu, Jonathan C. Beard, Ashen Ekanayake, Andreas Gerstlauer, and Lizy K. John. Virtual-link: A scalable multi-producer multi-consumer message queue architecture for cross-core communication. *2021 IEEE International Parallel and Distributed Processing Symposium*, pages 182–191, 2021.

[95] Qinzhe Wu, Ashen Ekanayake, Ruihao Li, Jonathan Beard, and Lizy John. Spamer: Speculative push for anticipated message requests in multi-core systems. In *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450397339. doi: 10.1145/3545008.3545044. URL `https://doi.org/10.1145/3545008.3545044`.

[96] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N. Patt. Branchnet: A convolutional neural network to predict hard-to-predict branches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 118–130, 2020. doi: 10.1109/MICRO50266.2020.00022.