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

Qinzhe Wu
qw2699@utexas.edu
University of Texas at Austin
Austin, Texas, United States

Ashen Ekanayake
ashen.ekanayake@utexas.edu
University of Texas at Austin
Austin, Texas, United States

Ruihao Li
liruihao@utexas.edu
University of Texas at Austin
Austin, Texas, United States

Jonathan Beard
Jonathan.Beard@arm.com
Arm Inc.
Austin, Texas, United States

Lizy K. John
ljohn@ece.utexas.edu
University of Texas at Austin
Austin, Texas, United States

ABSTRACT

With increasing core counts and multiple levels of cache memories, scaling multi-threaded and task-level parallel workloads is continuously becoming a challenge. A key challenge to scaling the number of communicating tasks (or threads) is the rate at which existing communication mechanisms scale (in terms of latency and bandwidth). Architectures with hardware accelerated queuing operations have the potential to reduce the latency and improve scalability of moving data between processing elements, reducing synchronization penalties, and thereby improving the performance of task-level parallel workloads. While hardware queues reduce synchronization penalties, they cannot fully hide load-to-use latency, i.e., perfect pipelines often are not realized. There is the potential, however, for better overlap. If the inter-processor communication latency is equal to or less than the time spent processing a message at the consumer, any and all latency may be overlapped while the consumer is processing. We exploit this property to speedup parallel applications above and beyond existing hardware queues.

In this paper, we present *SPAMeR*, a speculation mechanism built on top of a state-of-the-art hardware-driven message queue architecture. *SPAMeR* has the capability to speculatively push messages in anticipation of consumer message requests. 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* adds algorithms that attempt to predict this. We evaluate the effectiveness of *SPAMeR* with a set of diverse task-parallel benchmarks utilizing the *gem5* full system simulator, and observe a 1.33× average speedup.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '22, August 29-September 1, 2022, Bordeaux, France
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545044

KEYWORDS

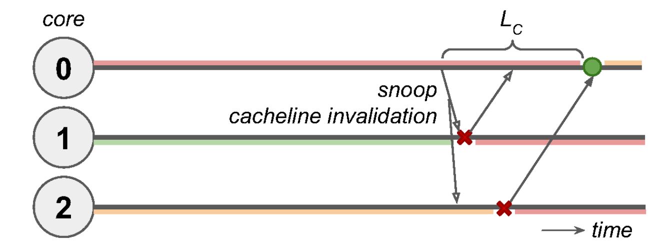
multi-core system, parallel computing, message queue, speculation

ACM Reference Format:
Qinzhe Wu, Ashen Ekanayake, Ruihao Li, Jonathan Beard, and Lizy K. John. 2022. SPAMeR: Speculative Push for Anticipated Message Requests in Multi-Core Systems. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3545008.3545044

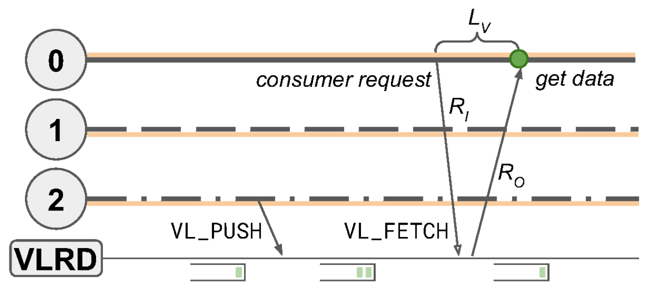
1 INTRODUCTION

The scaling of cooperative threads is an attractive way to increase performance of parallel workloads. This is especially true as modern Chip Multiprocessors (*CMP*) have more processing elements than ever before. Unfortunately, increasing cooperative threads often leads to diminishing performance returns [14, 48]. Coherent shared-memory multi-threaded applications particularly suffer due to load-to-use latency and synchronization overhead [4, 25]. The introduction of multi-core compute systems brought with it non-uniform latency between processing elements. *CMP*s have cores at the top of the hierarchy, where each core connects to a private cache, then after a certain level (point of coherence) the cache is often shared among a cluster of cores, and eventually all the cores have the access to a consistent memory. For parallel programs to execute correctly, these hierarchies implement a cache coherence protocol which is mediated by snoop messages traveling back and forth in the hierarchy (invalidating, updating, and checking-out cache lines for data requests) to enable cross-core communication between software agents at a coherence line granularity (e.g., snoop and invalidation in a MOESI coherence protocol, Figure 1a). With increasing core counts and communicating software, more messages must be sent across this hierarchy (both data and coherence traffic), increasing contention for data network resources (and shared data) and thereby increasing the overall latency of many data access operations [34].

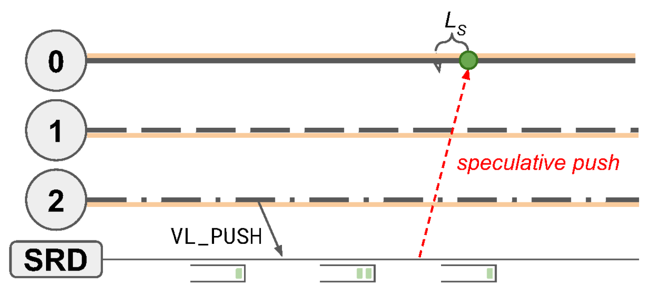
To address cross-core communication overheads (i.e., latency and synchronization), researchers have proposed adding hardware queues that could directly transmit data from one Processing Element (*PE*) to another using a dedicated network [20]; such machines often adopt a dataflow, streaming, communicating sequential process, or systolic array-like computation patterns [21, 27]. Such dataflow patterns are prevalent in machine learning [1, 29, 35], and



(a) MOESI: acquire cacheline to update queue states



(b) Virtual-Link: no interference between endpoints w/ unique addresses



(c) *SPAMeR*: routing device speculatively pushes data

Figure 1: Cross-core message queue communication latency (*Lc*) gets reduced from cache coherence-based queue to Virtual-Link [48] hardware queue (*Lv*) and *SPAMeR* hardware queue with speculative pushes (*Ls*).

many other high-performance computing tasks [13]. More recently, researchers have proposed architectures that achieve a similar effect to hardware queues with dedicated networks, while reusing components of the traditional hierarchical data coherence network. For instance, both Wang et. al. [46] and Wu et. al. [48] propose architectural message queue frameworks (with associated microarchitecture). Both approaches reduce the overall synchronization overhead, a key bottleneck for some applications as demonstrated by Virtual-Link (*VL*) [48]. In most, if not all, hardware queue implementations (e.g., *VL*) the data is served in an on-demand manner, that is, data is fetched only when requested by a receiver (Figure 1b). None of these approaches attempt to speculate when a message request will arrive (e.g., a *pop* operation). *SPAMeR* places data ahead to the receiver that needs it but have not requested yet (Figure 1c), in order to further reduce the load-to-use latency.

Traditional speculative methods to reduce load-to-use latency rely on prefetching what data comes next based on a learned pattern, however, when communicating between asynchronous threads, it is unclear which data should be fetched next in the general case (consider the case of *M:N* communicating threads, which one should receive the next message?). When a hardware queue is used, it knows exactly what data is next (using the property of the queue). In the case of a hardware queue, perhaps a better speculation approach is to have the hardware queue guess when to push data to the consumer, in anticipation of a request. This would remove the added latency for the consumer data request, however, speculating *when* to push data and not what data to fetch is a new problem to solve, one to which we contribute a solution. Another problem, which is different from a traditional prefetch algorithm, is that we also must select which receiver (assume *M:N*) will need the next message, something that we also contribute to.

As Virtual-Link is the state-of-the-art hardware message queue architecture, we adopt *VL* as the underlying hardware queue, and then improve upon it, noting that there are additional data network transactions that could be removed if speculation (which we provide) was adopted. To our knowledge, there are no prior works that speculatively push data within a message channel (such as created by *VL*).

The primary contributions of this paper are:

(1) We design and implement a speculative message queue architecture that we call *SPAMeR*. This architecture can prepush data amongst asynchronous communicating threads with little coherence overhead, anticipating future consumer data requests, therefore further reducing load-to-use (e.g., *pop*-to-use) and enabling more communication to overlap.
(2) We evaluate the performance of *SPAMeR* comparing it to state-of-the-art hardware queues and explore the benefits of speculative message passing. The *gem5*-based full system simulation demonstrates a speedup of 1.33× for 8 task-parallel workloads over state-of-the-art.

The rest of the paper is organized as follows: first we present a short overview of the state-of-the-art hardware queue mechanisms in the background section (§ 2). We then present the design of *SPAMeR* (§ 3), followed by evaluation of the *SPAMeR* implementation (§ 4), related work (§ 5), and lastly the conclusions(§ 6).

2 BACKGROUND

This paper presents a speculative pre-push mechanism, *SPAMeR*, for hardware queue architectures. We pick one of the state-of-the-art hardware queue solutions, Virtual-Link (*VL*) [48] as the base to build on, because *VL* has addressed the scalability issue caused by coherence traffic on shared states and achieved good performance. However, *VL* does not touch the problem that how to speculate the message request and hide the load-to-use latency, which is issue that *SPAMeR* targets on. We first present an overview of *VL* in this section. Other prior works are described in Section 5, and the techniques we develop on top of *VL* are likely apply to other hardware queues.

VL is a light-weight communication mechanism with hardware support to facilitate *M:N* lock-free data movement [48]. *VL* attaches a routing device to the coherence network, as shown in Figure 2, to facilitate transmission of data (cache lines) from producer to consumer. The routing device enables *VL* to “link” unique producer

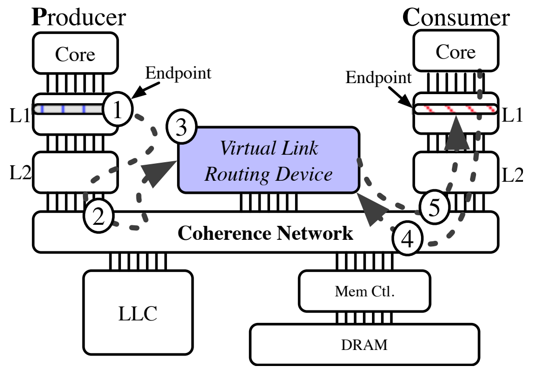


Figure 2: The Virtual-Link hardware queue [48]

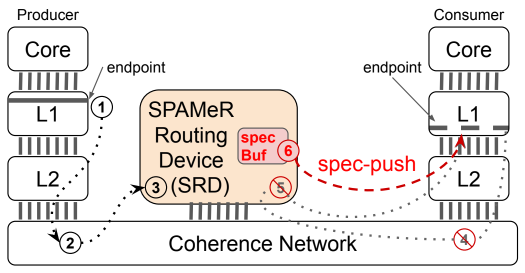


Figure 3: Overview of *SPAMeR* architecture. The *specBuf* added by *SPAMeR* in the routing device enables (6), speculative pushes. Speculative pushes can replace (4) and (5) in the baseline design and deliver the data with reduced latency.

and consumer "endpoints" together via a Shared Queue Identifier (*SQI*). Endpoints subscribe to a *SQI* to form a *M:N* message channel; each *SQI* can be thought of as a single queue "object". Messages are sent from each producer non-coherently and they are copied over to a unique target memory location at the consumer. In doing so, the producer gives the illusion of a shared memory message passing connection, but with no true sharing. Figure 2 illustrates a complete flow for *VL*. At (**1**) a cache line moves from the producer, at its own unique address location, to an indirection layer in hardware at (**2**) that is addressed based on the *SQI*. That indirection layer, the Routing Device, matches the *SQI* at (**3**) to a consumer based on consumer endpoint demand, which is registered by (**4**). The Routing Device forwards data to the target consumer buffer on a totally different memory address at (**5**). From (**4**) and (**5**), we can see *VL* delivers data in an on-demand manner, that wait until the consumer requests then push the data to where it is needed.

VL reduces the amount of coherent shared state, a bottleneck for many approaches, to zero. *VL* provides further latency benefits by keeping data on the fast path (i.e., within the on chip interconnect). *VL* enables directed cache-injection (stashing) between PEs on the coherence bus, reducing the latency for core-to-core communication. *VL* is particularly effective for fine-grain tasks on streaming data. This paper strives for more improvement over *VL* through reduced load-to-use latency and enhanced message pipelining.

3 DESIGN OF *SPAMER*

The *SPAMeR* design is an extension of the Virtual-Link [48] architecture. We first describe how the prior Virtual-Link mechanism is extended to give the functionality of *SPAMeR* (§ 3.1). We then describe our specific micro-architecture (§ 3.2), ISA (§ 3.3), and library (§ 3.4) contributions. We explore the design space in terms of speculation algorithms (§ 3.5); we also discuss potential security vulnerabilities and their mitigation (§ 3.6).

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

Figure 3 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 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 paper). 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 3), 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 3). 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 3), 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 3, 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 4 is analogous to the *VLRD* of [48] with several exceptions highlighted in red part (will be explained in Section 3.2). 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 4 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

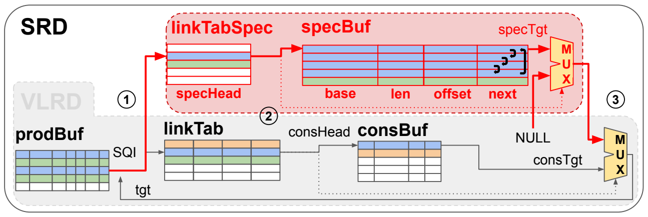


Figure 4: *SPAMeR* Routing Device (*SRD*) includes the *VLRD* components from Virtual-Link Routing Device design (with the grey background), which we shrink for brevity, and some newly-added structures (highlighted in red) to enable speculative push. Cell filling colors indicate different *SQI* occupancy. Numbers in circle mark the three stages of the address mapping process.

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 4, 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 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*.

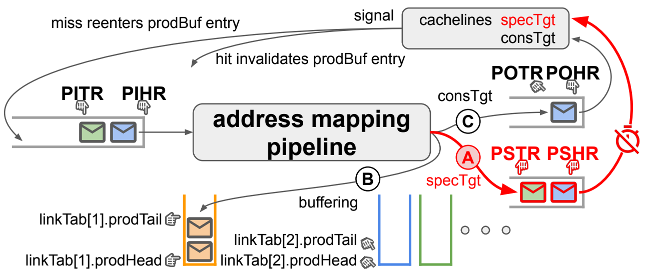


Figure 5: There are three different possible outcomes from the address mapping pipeline (marked with letter A, B, C in circles), and the consumer cachelines give hit/miss response signals for every push (on-demand or speculative). Path and structures related to speculative push are highlighted in red, 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.

Figure 5 (except the red parts) shows what happens after address mapping in the original Virtual-Link architecture (the same process is used in the *SRD*). There are two possible outcomes: the data finds a target and gets into the sending queue (Path (**C**) of Figure 5); 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). Please note that the queues shown in Figure 5 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 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* of [48] and our *SRD*. What makes our *SRD* different is described next.

3.2 *specBuf*

SPAMeR adds an additional speculative push path ((**6**) in Figure 3), 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 4) 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 we will explain in Section 3.3. 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` × 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` × 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). After some delay, the *SRD* sends the data to the target cacheline. The delay is key for efficient speculation as we discuss further in § 3.5 and § 4.3.

3.3 Speculative Address Registration Instruction

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*.

3.4 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 3.3. 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). We revise the original *VL* library code [43] 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 3, 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.

3.5 Speculation Algorithms

Just like prefetching, speculative pushing could be history-based[33], profiling-guided[30], heuristic-oriented[50], or perceptron-style[8]. For *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 (§ 3.2), 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 0-delay, which does not add any additional delay, but lets the speculative push go as soon as possible. The 0-delay algorithm can maximize the performance, because as long as there are available producer data in *SRD*, it keeps trying speculative pushes. This lets the 0-delay 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 adaptive delay algorithm. The adaptive 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 adaptive 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 adaptive algorithm approach is too simple to fully model the consumer behavior (as we will show in Section 4.3).

We come up with a tuned delay prediction algorithm tuned for the benchmark which analysis suggests has the greatest potential (§ 4). The intuition for the design of the tuned algorithm is to take 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 calculates 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 6 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

```
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;
  }
}
```

```
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 1: Tuned delay prediction algorithm.

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. On the right of Listing 1, 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 6. Function `lookupSpecTab` on the left side of Listing 1, along with the left part of Figure 6, elaborates how the algorithm generates the delay time to send data. The orange Greek letters in Figure 6 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 § 4.4). 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 ($\zeta = 256$, $\tau = 96$, $\delta = 64$, $\alpha = 1$, $\beta = 2$), then as a cross-validation, apply the parameterized algorithm to all the benchmarks in the evaluation. As future work, we could search to find a more optimal set of parameters for each benchmark and reconfigure those parameters dynamically.

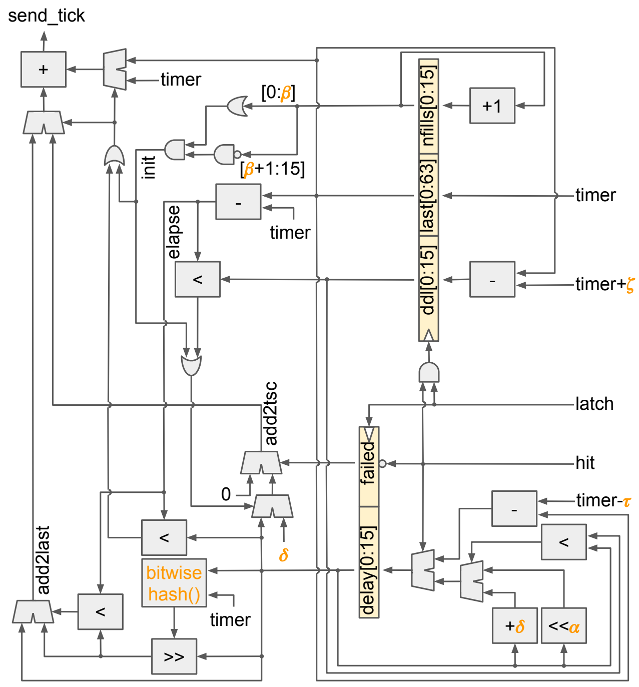


Figure 6: The example hardware logic implementation of the tuned delay prediction algorithm. All "timer" label in the diagram refer to the time stamp counter (tsc), while "timer+ζ" and "timer-τ" input ports on the right could be from two other time stamp counters configured to have constant offsets (i.e., ζ, $-\tau$) from tsc.

3.6 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 [9, 39]. 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 [3], while *SPAMeR* does not use content for prediction. 3) Attacker could derive memory layout from prefetching latency [18]. 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 [19], 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.

4 EVALUATION

4.1 Methodology

Table 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
SRD	64 entries per *prodBuf*, *consBuf*, *linkTab*, and *specBuf*

We evaluate the proposed *SPAMeR* architecture using full-system simulation. The approach was to implement *SPAMeR* on top of the *gem5* [11] code base from Virtual-Link [48] repository [43], and enhance it with the proposed *SPAMeR* Routing Device (*SRD*, § 3.2). *SPAMeR* works as an extension to the AArch64 architecture. The simulation settings used are shown in Table 1.

Table 2 lists the benchmarks we use in our evaluation. *ping-pong*, *halo*, *sweep*, and *incast* are the common communication patterns derived from the Ember benchmark suite [40]. *pipeline*, and *firewall* represent the styles of many network packet processing workloads [46]. *FIR* exists in many Digital Signal Processing workloads, and *bitonic* is a sorting algorithm with plenty of parallelism for hardware to exploit. The benchmarks cover different types of queues (one-to-many, many-to-one, many-to-many etc.). Such queue information is marked as (*M*:*N*)×*k* at the end of each row, where *M* denotes the number of producers to the queue, *N* denotes the number of consumers of the queue, and *k* denotes the number of such queue instances. For example, *firewall* has 4 message queues, three of which are one-to-one queues, and there is another queue having two producers and one consumer. This software structure design is influenced by the work from Wang et.al. [46]. The benchmarks show diversity on the number of threads too, ranging from 2 to 16. Each thread is assigned to a core in order to reduce the migration overhead in the experiments. All the benchmarks are compiled with '`-O3`' level optimization using '`gcc-8.2.0`'.

Table 2: Benchmarks.

Benchmark	Description, (#producer:#consumer) × #queue
ping-pong [40]	data back and forth between two threads (1:1)×2
halo [40]	exchange data with neighboring threads (1:1)×48
sweep [40]	data sweeps through a grid of threads corner to corner (1:1)×48
incast [40]	all threads sending data to the master thread (4:1)×1
pipeline [46]	4-stage pipeline with middle stages multi-threaded (1:4)×1+(4:4)×1+(4:1)×1+ (1:1)×1
firewall [46]	filter and dispatch packages (1:1)×3+(2:1)×1
FIR	data streams through 10-stage FIR filter (1:1)×9
bitonic [5]	bitonic sort with varying number of threads (1:N)×1+(M:1)×1

4.2 Message Queue Workload Tracing

In order to get a better sense of how *SPAMeR* would reduce the cross-core communication latency, we trace a few key events of each message queue transaction in *incast* (which can have the simplest queue setting so it is relatively easy to reasoning), then visualize the transactions for detailed analysis. Figure 7 presents an example trace where we can observe a mix of different types of message queue transactions. In order to make the example easy to follow, the trace is from *incast* that is configured to have a single message queue, a single consumer cacheline, and single producer thread. From the overview chart at the top of Figure 7, we can see two phases: 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 generates a burst of data and the consumer becomes the bottleneck.

The bottom chart of Figure 7 zooms in to reveal more details at the transition of these two phases. The markers joined by lines are the different events in a transaction. For each marker, its x -axis

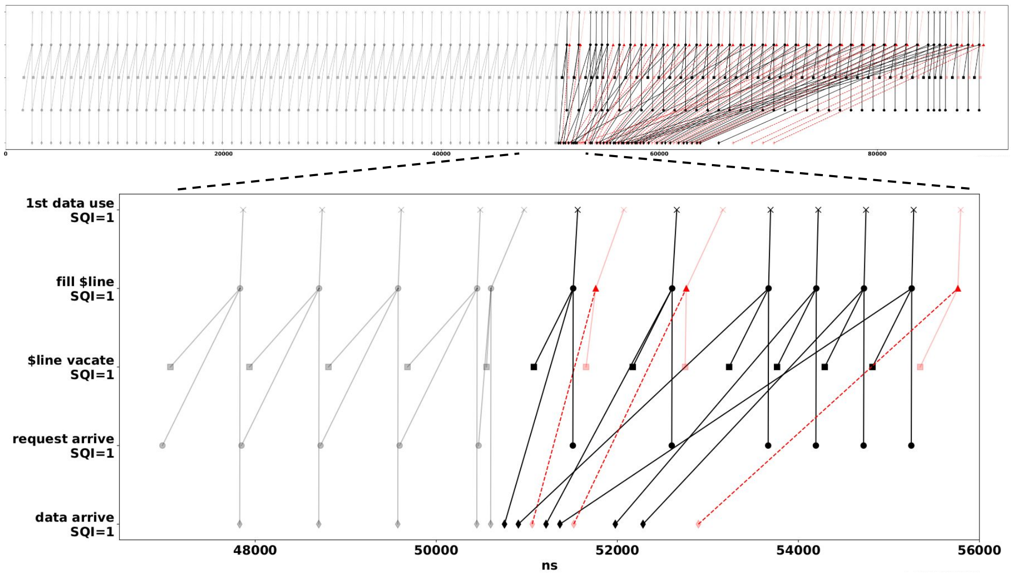


Figure 7: One of the simplest traces showing different message queue transactions in *incast* benchmark (single *SQI*) with single consumer cacheline, single producer thread. For each marker, its x axis value is the timestamp, and it is vertically located according to the event type. Red dashed lines 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.

value is the timestamp, and it is vertically located according to the event type as indicated on the y -axis. 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 (×) for the consumer's first use of the data. For on-demand pushes (solid lines in the chart), data arrival, request arrival, and cacheline vacation must precede the cacheline fill, while speculative pushes (red dashed lines) have no request arrival event associated to the transaction. We highlight some of the on-demand push transactions in dark black, because in these transactions, filling the consumer cacheline with data is hindered by the request arrival, the latest one among the three events that an on-demand transaction requires. If a speculative push was triggered, the delivery of the data could have gone earlier as soon as both data arrival and cacheline vacate events happen. The potential speculative push saving in those transactions are calculated as the difference between the cacheline fill timestamp and the latest of data arrival or cacheline vacation.

We also notice a "prerequest" behavior in the trace that a transaction (e.g., leftmost 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 we observe its random impact on the performance of *VL* (§ 4.3). *SPAMeR* replaces such "prerequest" with educated speculation.

4.3 *SPAMeR* Performance

As mentioned in Section 3.4, 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 8 compares the performance of *SPAMeR* against the baseline, Virtual-Link. As we can see, with the aggressive 0-delay 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 adaptive delay algorithm obtains performance improvement fairly close to the 0-delay algorithm. This is because the *FIR* worker threads could switch between fast path and slow path, and the adaptive 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 0-delay algorithm, the adaptive and the tuned delay algorithm get 1.45×, 1.25× and 1.33× speedup, respectively.

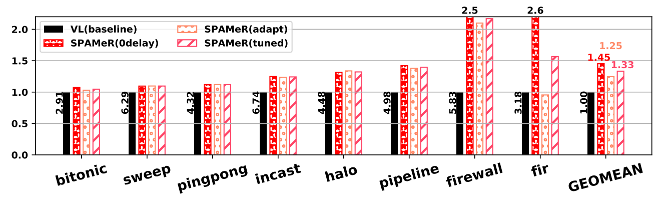


Figure 8: Performance improvement *SPAMeR* gains over Virtual-Link (prior work), the higher the more performant. Execution time normalized to the baseline (labeled on the left of the black solid bar in millisecond).

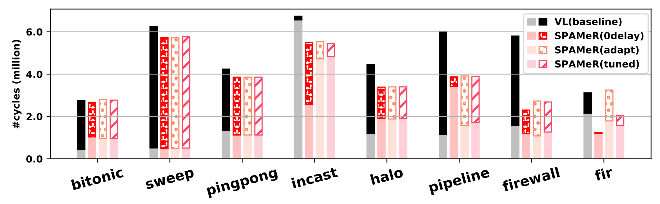
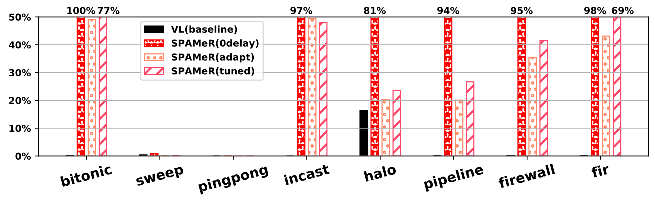


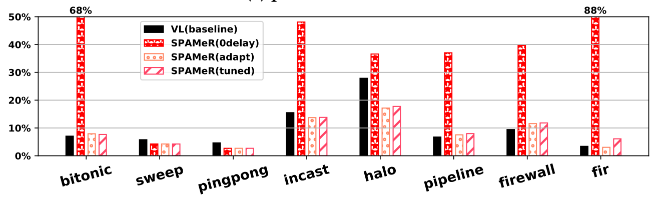
Figure 9: Execution time breakdown: the top of the bars stands for average consumer cacheline empty cycles, and the bottom of the bars is for non-empty.

Figure 9 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 9 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 once hit peak consumer throughput, for example, *bitonic* and 0-delay 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 0-delay might quickly fill up all 32 cachelines then blocked on one (round-robin as designed in § 3.5), until the consumer thread uses up all data in other cachelines. This pattern causes half of the cycles in *incast* with 0-delay 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.

On the other side, the 0-delay algorithm costs more bus utilization and energy than other delay algorithms due to its higher failure rates. In Figure 10a, we compare how many pushes (counting both on-demand pushes and speculative ones) fail out of total across four different settings. *VL* has 0% failure rate on almost all the benchmarks. One apparent exception is *halo*, where multiple threads in the grid might frequently request data again and over again from their neighboring threads, leading to 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



(a) push failure rate



(b) bus utilization

Figure 10: The push failure rates and bus utilization of Virtual-Link, and *SPAMeR* with different delay algorithms. The higher the less efficient.

fail. However, due to the plenty speculation opportunities exist in *halo*, *SPAMeR* gets 1.33× speedup on *halo*, and such "prefetches" is also overall beneficial to the the performance of the *VL* baseline (without "prefetches" *VL* would be 0.94× slower on *halo*). *SPAMeR* with 0-delay 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, the consumer cacheline of the node is probably ready ahead. Therefore, *ping-pong* and *sweep* are the only two 0-delay algorithm which does not make many failures. The adaptive 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 10b). The failure rate for the tuned algorithm is slightly higher than the adaptive algorithm. Achieving near-zero speculative push failure rate for arbitrary workload is no 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% [45].

The higher the push failure rate, the more wasted traffic on the bus. Figure 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 0-delay algorithm consumes much more bandwidth than others on most of the benchmarks. *SPAMeR* with the adaptive 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 adaptive 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.

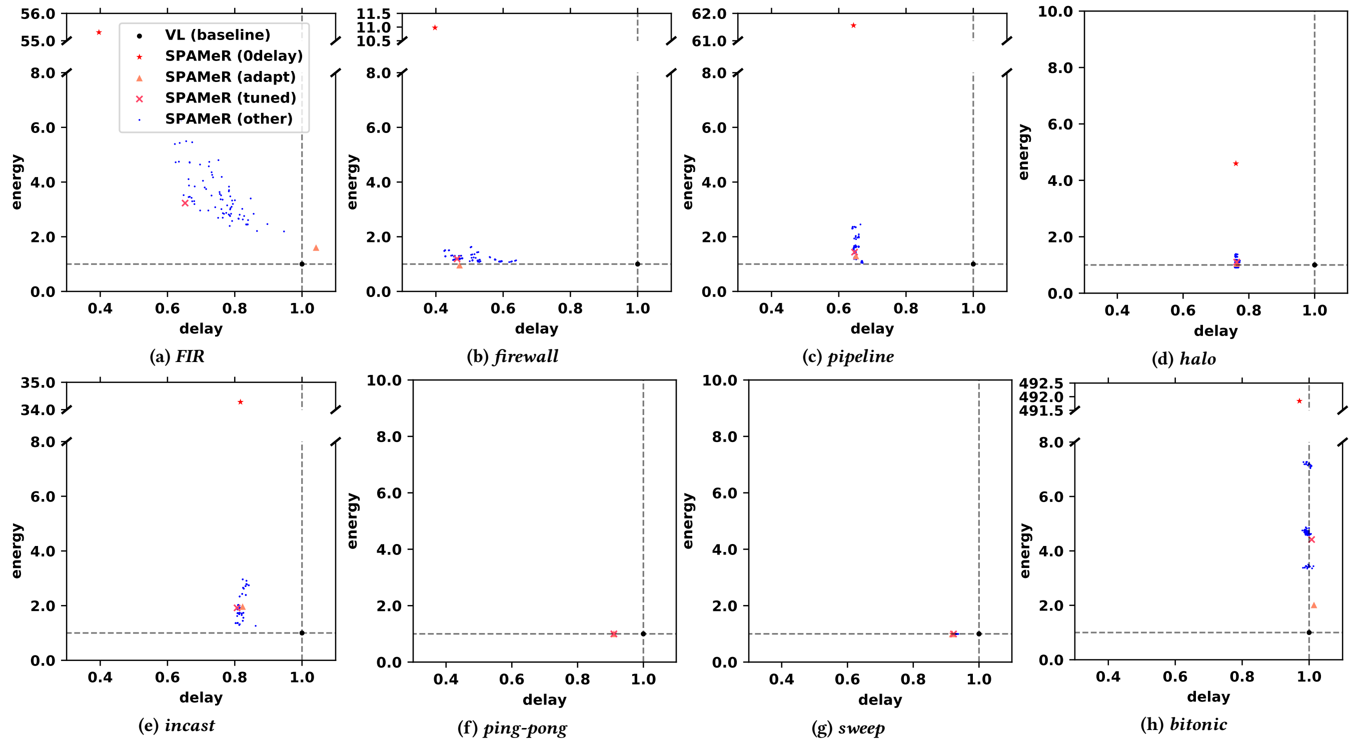


Figure 11: Execution time v.s. the dynamic part of energy consumed by *SRD* pushes. Both axis are normalized to *VL* baseline. Other combinations of the tuned algorithm parameters are included to show how is the algorithm sensitive to the parameters.

4.4 Sensitivity Study

There are several parameters (i.e., ζ, τ, δ, α, β) in the tuned algorithm design, so we explore the sensitivity to the parameter combinations as shown in Figure 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 dynamic energy consumed by *SRD* pushes (denoted as energy), respectively. Different benchmarks have different communication patterns therefore react differently to the varying parameters. In order to sense the parameter sensitivity consistently across benchmarks, we keep the scale 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 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 11a illustrate. Because the set of parameters ($\zeta = 256$, $\tau = 96$, $\delta = 64$, $\alpha = 1$, $\beta = 2$, 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 11b to Figure 11h show, the chosen parameter setting might be suboptimal 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 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.

4.5 Area and Power Estimation

The Virtual-Link [48] work estimated the area cost of the *VLRD* by developing RTL code and scaled the synthesis result on the FreePDK 45 nm library [42] to 16 nm technology node [41]. 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 [47], 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 0-delay algorithm. Different delay prediction algorithms (e.g., adaptive 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 0-delay algorithm would yield too much higher power to be realistic, while the adaptive 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 [47], 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 RELATED WORK

Software message queues range in implementation and complexity from the very basic lock-free queues of Michael and Scott [31] to more recent implementations of these structures such as [32] and [23]. More recent software works focus on lock-free, application specific data structures for increased performance and lower-latency versus a more general queueing solution such as that proposed by *SPAMeR* (e.g., Kite [17]). One thing many of these software frameworks have in common is that they rely on demand data access and generic prefetchers to place the data as close to the receiver as possible. Some software works that attempt to pre-push data include [44] and [16]. While these works address some variations on pre-pushing, they still rely on atomic operations and coherence structures which have scaling issues [14, 48], *SPAMeR* does not. Dataflow, streaming, and event-based languages/runtimes allow programmers to write applications described as a graph, these frameworks enable the compiler and runtime to place prefetch ahead of time for indirect buffers pointed to by messages received, examples include [6, 10, 15]. These techniques do nothing for the synchronization points, however they do improve performance for accessing the indirect buffers.

Moving beyond atomic operations to mitigate scaling issues seen in modern coherent systems, computer architects and researchers have attempted to provide hardware support for communicating threads. These solutions range from instructions to facilitate direct memory transfer (or register to register) to hardware-software solutions. Domain specific solutions such as the TILE64 [7], digital signal processors such as the IBM Cell [12], the Freescale DPAA [36], and others provide data movement operators to send data directly from *PE*-to-*PE*, often in the form of direct memory access transfers (i.e., DMA). More recent works such as HAQu [28], CAF [46], the Intel DLB [22], the RISC-V based "moving compute" hardware channels model [14], and "Virtual-Link" [48], which this work extends, all provide hardware acceleration that reduce core-to-core message latency and increase overall throughput. A key to almost all of these frameworks is that they decouple the coherence coupling between producer and consumer, reducing the ping-pong effect of repeated shared-to-exclusive cache-line upgrades. None of these works provide a means of speculatively injecting data to target consumers.

The domain of software prefetch and pre-population of data caches in general, is quite rich. Concepts such as run-ahead threads [37] and customized data structures to allow easier software prefetch [49] exist in the literature. None of these solutions target communicating threads which would likely destroy these optimizations. Concepts such as decoupled access-execute [26] could produce higher hit rates with communicating threads, at the cost of much higher contention within the interconnect; such solutions still have the same scalability bottlenecks as traditional coherent systems, which *SPAMeR* specifically addresses. Special purpose hardware that drives targeted prefetch has also been proposed [2]. More recent works have looked specifically at prefetching in the face of communicating threads [24, 38]. While these works look to understand and mitigate the impact of prefetch across synchronization boundaries, *SPAMeR* pre-pushing removes the synchronization boundary while simultaneously netting the benefit of placing the data as close to the core as possible.

6 CONCLUSION

In this paper, we present 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× speed up over a state-of-the-art hardware message queue architecture on 8 task-parallel benchmarks. We also use *gem5* to study the benchmarks, and perform a detailed analysis on the message queue communication overhead. We believe the proposed architecture would assist the effectiveness of multi-core systems handling task-parallel dataflow workloads.

ACKNOWLEDGMENTS

This research was supported in part by NSF grant numbers 1745813, and 1763848, and funding from Arm. We also acknowledge the computing servers donated from Ampere Computing. Any opinions, findings, conclusions or recommendations are those of the authors and not of the National Science Foundation or other sponsors.

REFERENCES

[1] Martín Abadi, Michael Isard, and Derek G Murray. 2017. A computational model for TensorFlow: an introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 1–7.

[2] Sam Ainsworth and Timothy M Jones. 2016. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*.

[3] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-like Attacks by Capturing Speculative State. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 132–144. https://doi.org/10.1109/ISCA45697.2020.00022

[4] Mohammad Bakhshalipour, Seyedali Tabaeiaghdaei, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Evaluation of hardware data prefetchers on server processors. *ACM Computing Surveys (CSUR)* 52, 3 (2019), 1–29.

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

[6] Jonathan C Beard, Peng Li, and Roger D Chamberlain. 2017. Raftlib: A C++ template library for high performance stream parallel processing. *The International Journal of High Performance Computing Applications* 31, 5 (2017), 391–404.

[7] S. Bell, B. Edwards, J. Amann, R. Conlin, K. 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. 2008. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*. 88–598. https://doi.org/10.1109/ISSCC.2008.4523070

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

[9] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. 2016. 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* (Seoul, Republic of Korea) *(HASP 2016)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/2948618.2948624

[10] Tiwei Bie, Changchun Ouyang, and Heqing Zhu. 2020. Virtio. In *Data Plane Development Kit (DPDK)*. CRC Press, 229–250.

[11] 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. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[12] Thomas Chen, Ram Raghavan, Jason N Dale, and Eiji Iwata. 2007. Cell broadband engine architecture and its first implementation—a performance view. *IBM Journal of Research and Development* 51, 5 (2007), 559–572.

[13] Iacopo Colonnelli, Barbara Cantalupo, Roberto Esposito, Matteo Pennisi, Concetto Spampinato, and Marco Aldinucci. 2021. HPC Application Cloudification: The StreamFlow Toolkit. In *12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[14] Halit Dogan, Masab Ahmad, Brian Kahne, and Omer Khan. 2019. Accelerating synchronization using moving compute to data model at 1,000-core multicore scale. *ACM Transactions on Architecture and Code Optimization* 16, 1 (2019), 1–27.

[15] Alan AA Donovan and Brian W Kernighan. 2015. *The Go programming language*. Addison-Wesley Professional.

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

[17] Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, and Arpit Joshi. 2020. Kite: efficient and available release consistency for the datacenter. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–16.

[18] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 368–379. https://doi.org/10.1145/2976749.2978356

[19] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang. 2022. Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1550–1550. https://doi.org/10.1109/SP46214.2022.00121

[20] W Daniel Hillis. 1989. *The connection machine*. MIT press.

[21] Ali R Hurson and Krishna M Kavi. 2007. Dataflow computers: Their history and future. *Wiley Encyclopedia of Computer Science and Engineering* (2007).

[22] Intel. 2020. *Queue Management and Load Balancing on Intel® Architecture*. Retrieved February 2022 from https://intel.ly/3hY0Zy8

[23] Giorgos Kappes and Stergios V Anastasiadis. 2021. A lock-free relaxed concurrent queue for fast work distribution. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 454–456.

[24] Engin Kayraklioglu, Michael P Ferguson, and Tarek El-Ghazawi. 2018. LAPPS: Locality-aware productive prefetching support for PGAS. *ACM Transactions on Architecture and Code Optimization* 15, 3 (2018), 1–26.

[25] Andi Kleen. 2009. Linux multi-core scalability. In *Proceedings of Linux Kongress*.

[26] Konstantinos Koukos, Per Ekemark, Georgios Zacharopoulos, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2016. Multiversioned Decoupled Access-execute: The Key to Energy-efficient Compilation of General-purpose Programs. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. https://doi.org/10.1145/2892208.2892209

[27] Ben Lee and Ali R Hurson. 1993. Issues in dataflow computing. In *Advances in computers*. Vol. 37. Elsevier, 285–333.

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

[29] Thorben Louw and Simon McIntosh-Smith. 2021. *Using the Graphcore IPU for traditional HPC applications*. Technical Report. EasyChair.

[30] Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P. Geoffrey Lowney, and Robert Cohn. 2002. Profile-Guided Post-Link Stride Prefetching. In *Proceedings of the 16th International Conference on Supercomputing* (New York, New York, USA) *(ICS '02)*. Association for Computing Machinery, New York, NY, USA, 167–178. https://doi.org/10.1145/514191.514217

[31] Maged M Michael and Michael L Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 267–275.

[32] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. BQ: A lock-free queue with batching. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 99–109.

[33] K.J. Nesbit and J.E. Smith. 2004. Data Cache Prefetching Using a Global History Buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. 96–96. https://doi.org/10.1109/HPCA.2004.10030

[34] Davide Pasetto, Massimiliano Meneghin, Hubertus Franke, Fabrizio Petrini, and Jimi Xenidis. 2012. Performance evaluation of interthread communicationmechanisms on multicore/multithreaded architectures. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. 131–132.

[35] Raghu Prabhakar and Sumti Jairath. 2021. SambaNova SN10 RDU: Accelerating Software 2.0 with Dataflow. In *2021 IEEE Hot Chips 33 Symposium*. IEEE, 1–37.

[36] DPAA QorIQ. 2012. *Primer for Software Architecture*. Technical Report. Technical report, Freescale Semiconductor Inc.

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

[38] Isaac Sánchez Barrera, David Black-Schaffer, Marc Casas, Miquel Moretó, Anastasiia Stupnikova, and Mihail Popov. 2020. Modeling and optimizing numa effects and prefetching with machine learning. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–13.

[39] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 131–145. https://doi.org/10.1145/3243734.3243736

[40] sstsimulator. 2020. *Ember Communication Pattern Library*. Retrieved October 2020 from https://bit.ly/3k9egUV

[41] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration* 58 (2017), 74 – 81. https://doi.org/10.1016/j.vlsi.2017.02.002

[42] 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. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. 173–174. https://doi.org/10.1109/MSE.2007.44

[43] UT-LCA. 2021. *GitHub Virtual-Link*. Retrieved November 2021 from https://github.com/UT-LCA/near-data-sim

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

[45] Haoyuan Wang and Zhiwei Luo. 2017. Data Cache Prefetching with Perceptron Learning. arXiv:arXiv:1712.00905

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

[47] Scoot Wasson. 2015. *Inside ARM's Cortex-A72 microarchitecture*. Retrieved February 2022 from https://bit.ly/3sf0a9h

[48] Qinzhe Wu, Jonathan C. Beard, Ashen Ekanayake, Andreas Gerstlauer, and Lizy K. John. 2021. Virtual-Link: A Scalable Multi-Producer Multi-Consumer Message Queue Architecture for Cross-Core Communication. *2021 IEEE International Parallel and Distributed Processing Symposium* (2021), 182–191.

[49] T. Yamada, S. Hirasawa, H. Takizawa, and H. Kobayashi. 2015. A Case Study of User-Defined Code Transformations for Data Layout Optimizations. In *2015 Third International Symposium on Computing and Networking (CANDAR)*. https://doi.org/10.1109/CANDAR.2015.96

[50] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. 2019. Improving Cache Performance for Large-Scale Photo Stores via Heuristic Prefetching Scheme. *IEEE Transactions on Parallel and Distributed Systems* 30, 9 (2019), 2033–2045. https://doi.org/10.1109/TPDS.2019.2902392