CAF: Core to Core Communication Acceleration Framework

Yipeng Wang[1], Ren Wang[2], Andrew Herdrich[2], James Tsai[2], and Yan Solihin[1]

[1]ECE, North Carolina State University
{ywang50, solihin}@ncsu.edu

[2]Intel Corporation
{ren.wang, andrew.j.herdrich, james.tsai}@intel.com

ABSTRACT

As the number of cores in a multicore system increases, core-to-core (C2C) communication is increasingly limiting the performance scaling of workloads that share data frequently. The traditional way cores communicate is by using shared memory space between them. However, shared memory communication fundamentally involves coherence invalidations and cache misses, which cause large performance overheads and incur a high amount of network traffic. Many important workloads incur significant C2C communication and are affected significantly by the costs, including pipelined packet processing which is widely used in software-based networking solutions. In these workloads, threads run on different cores and pass packets from one core to another for different stages of processing using software queues.

In this paper, we analyze the behavior and overheads of software queue management. Based on this analysis, we propose a novel C2C Communication Acceleration Framework (CAF) to optimize C2C communication. CAF offloads substantial communication burdens from cores and memory to a designated, efficient hardware device we refer to as Queue Management Device (QMD) attached to the Network on Chip. CAF combines hardware and software optimizations to effectively reduce the queue-induced communication overheads and improve the overall system performance by up to $2-12\times$ over traditional software queue implementations.

Keywords

Hardware queue; multicore communication; hardware accelerator

1. INTRODUCTION

New trends in network design and architecture, including Software Defined Networking (SDN) and Network Function Virtualization (NFV), are leading to rapid development of software-based packet processing on general purpose servers, which often requires high speed communication among multiple cores with packets passing through different processing stages [2]. Other workloads, such as job distributors and in-memory map-reduce algorithms used in machine learning [32] also involve frequent core-to-core (C2C) communication. Our study shows that C2C communication becomes a critical performance bottleneck in contemporary chip multi-processors (CMPs) for such workloads. The bottleneck can be attributed to the increasing number of processing elements (e.g., cores, accelerators, etc.) on a chip, that run workloads with cooperating threads that require multiple cores to exchange data frequently. In a typical CMP, the software queue structure is an important and commonly-used construct to realize C2C communication. However, software queues incur substantial performance overheads in modern data centers running SDN and NFV workloads, where significant core resources are dedicated for dequeue and enqueue operations to exchange data between cores. There is a clear demand for a more efficient communication method on the general purpose platforms.

To point out an example, Algorithm 1 shows a simplified pseudo code of a dequeue operation for a single-producer single-consumer queue. The algorithm is derived from a widely used software queue implementation, and has been optimized using a lock-free design (to avoid synchronization), batching (to amortize queue management overheads), and localizing global variables (to reduce coherence-related communication). Generally, each queue object has an array and four bookkeeping variables. The array represents the queue structure itself, which is shown as the variable *ring*. It contains pointers pointing to the actual data content. The four bookkeeping variables are producer head, producer tail, consumer head, and consumer tail respectively. A consumer thread will read and write the consumer head and tail, while only reading the producer tail. A producer thread will read and write the producer head and tail, while only reading the consumer tail. While optimized, the algorithm still suffers from significant performance overheads and scalability bottlenecks.

One source of overheads and bottleneck come from the communication triggered by the coherence protocol. The code involves multiple memory accesses, and each memory access may cause

Algorithm 1 SW queue dequeue function.

```
1:  procedure DEQUEUE(dataArray, bulkSize)
2:      localHead = consumerHead;
3:      availEntries = producerTail − localHead;
4:      if availEntries < bulkSize then
5:          return ERROR;
6:      consumerHead+ = bulkSize;
7:      index = localHead;
8:      for i ← 0, bulkSize − 1 do
9:          dataArray[i] = ring[index];
10:         next(index);
11:     consumerTail = consumerHead;
12:     return OK;
```


PACT '16, September 11-15, 2016, Haifa, Israel

DOI: http://dx.doi.org/10.1145/2967938.2967954

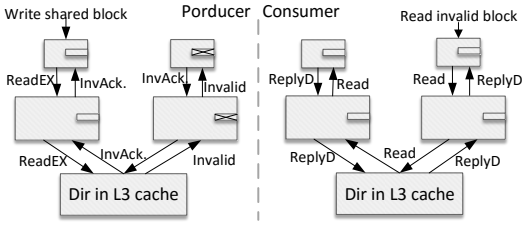


Figure 1: Illustrating the coherence-related communication overheads.

coherence-related communication. For example, when the consumer reads out of the *ring* array, the entries are likely still residing in the producer core's private L1 or L2 cache; in which case, coherence is triggered to forward the data to the target core, which could cost more than one hundred cycles. Thread contention will introduce even more overheads. In a more complex situation involving multiple consumers and multiple producers (code not shown), bookkeeping variables and data pointers are written and read by multiple threads. For example, in the dequeue algorithm, the *consumerHead* is read at line 2 but updated at line 6. In the multi-consumer situation, if this code section is not properly protected with critical section, one thread may use a stale *consumerHead* and update it with a wrong value. In the lock-free queue algorithms, dedicated instructions like compare and swap (CAS) are used to commit or retry the critical section. The critical section code is not scalable with more threads.

We refer to the cost of manipulating queue entries and bookkeeping variables (and to ensure thread-safety) as *control plane overhead*. To alleviate the control plane overhead, the algorithm employs bulk-processing, i.e., the application enqueues or dequeues a batched set, or a "bulk size" of data items each time to amortize the maintenance overhead. While this approach improves throughput through overhead amortization, it comes with a costly trade-off: higher and more unpredictable delays/jitters, due to each thread waiting for the bulk completion. Consequently, many real applications prefer the smallest bulk size that achieves a target throughput.

Another type of overhead inherent in C2C communication is what we call as *data plane overhead*. Data plane overhead is not explicitly shown in the pseudo code but becomes evident when the dequeued items are actually consumed. In common packet processing applications, the queue itself is merely an array of pointers pointing to actual data items, and the data items are the payload of the network packets that are passed from one processing stage to the next. The overhead for a core to access the data items that are pointed by the queue entries is referred as the data plane overhead. The data plane overhead may be large because the payload of a packet may be large, and the data items usually reside in another core's private cache. Figure 1 shows the communication overhead of a consumer reading an block residing in the producer's private cache, and a producer writing a shared block. Coherence flow is triggered to forward or invalidate the cache blocks from another core's local cache. The round-trip communication is very costly compared to accessing data directly from one core's local cache or the shared last level cache (LLC).

In this paper, we propose the C2C **Communication Acceleration Framework** (CAF) as a co-optimized software and hardware solution to improve C2C queue-based communications. CAF includes a new hardware structure called the *queue management device* (QMD). QMD is a device that is attached to the Network on Chip (NoC) of a CMP and acts as an accelerator for various queue management functionalities. A core can request services from the QMD through the Instruction Set Architecture (ISA), i.e. executing special enqueue and dequeue instructions. QMD achieves several significant benefits. First, it makes queue operations fast. Instead of executing hundreds of instructions at the core to manage a software queue, a core can execute an enqueue or dequeue instruction, with QMD handling the rest. Consequently, QMD frees up the core to work on more useful jobs. Second, QMD can handle multiple producers and consumers without requiring locks or synchronizations. Third, QMD removes most coherence-related communication incurred in software queue implementations, both in the control plane and in the data plane. The last two benefits increase the scalability ceiling vs. software queues. Furthermore, the scalability ceiling of QMD can be further lifted by making QMD distributed. Our results show up to $2-12\times$ throughput improvement compared to a fully optimized software queue structure. Finally, QMD can provide new functionalities. By using credit-based management system that controls the traffic flow for each thread, QMD can support Quality of Service (QoS). We also extend the functionality of QMD to accelerate more universal C2C communication applications, such as for job distributor.

The rest of the paper is organized as follows. Section 2 discusses related works. Section 3 describes the design specifics of CAF. Section 4 evaluates the proposal and Section 5 discusses some extensions to the basic design of CAF. Section 6 concludes the work.

2. RELATED WORK

Prior techniques in queue-based C2C communication in general fall into two categories: software and hardware approaches. For software approaches, numerous software queue algorithms have been proposed since decades ago until recently [18, 21, 10, 26, 34, 12, 33]. Those algorithms try to improve the scalability and throughput of the software queue by optimizing memory allocation or reducing thread contention. Meanwhile, one widely used queue algorithm in data centers for packet processing on a general processor is the Intel's DPDK [2] library. The queue algorithm from DPDK is well optimized for x86 systems and implements various optimizations, such as bulk processing. However, even a well-optimized software algorithm still requires substantial core resources to maintain the queue data structure and cannot avoid the costly thread contention. Packet processing workloads running in modern data centers still require significant core resources to be dedicated for pure dequeue and enqueue operations. Our proposed CAF offloads all queue maintenance work from the core to a dedicated hardware engine.

For hardware approaches, there are studies focusing on hardware-accelerated task queues [17, 20, 31], which reduce software overheads in task scheduling and load balancing. They are very different from CAF according to their hardware design and use cases. However, we will show that CAF can be used for task scheduling as well with an extra load balancing module. The most related work to ours is HAQu [22]. Like CAF, HAQu proposes hardware acceleration for queue operations. Although sharing the same purpose, CAF is fundamentally different from HAQu in several ways. HAQu maintains the queue information in the core, whereas CAF maintains the queue information in a NoC-attached device. In HQAu, cores spend many cycles to bookkeep queue information, and queue items still reside in cache, triggering memory access and coherence overheads. In contrast, QMD is attached to the NoC; this relieves the core and cache hierarchy from bookkeeping and keeping coherence. Furthermore, HAQu does not support multi-producer multi-consumer communication, because each core has its own local copy of the queue's head and tail. Extending

HAQu to support multi-producer multi-consumer requires keeping these heads and tails coherent across cores, which incurs a problematic new coherence problem. In contrast, CAF centralizes the queue structures, hence it does not incur new coherence issues between cores. CAF keeps modifications to the core minimum (adding new instructions). CAF allows multi-producer and multi-consumer queue communications, which is a critical support for packet processing.

In industry, Freescale designs a set of hardware components called DPAA [1] on a system-on-chip (SoC) to accelerate C2C communication. The detailed design and performance specification of DPAA is unknown to the public. However, to our best knowledge, the hardware queue manager of DPAA is not closely integrated with the core architecture. The potential latency is longer and the complexity of communication steps between the core and the queue manager is much higher than our proposed hardware design.

3. SYSTEM ARCHITECTURE AND DESIGN

The basic idea of the proposed Communication Acceleration Framework (CAF) is to offload the queue-related communication maintenance from the core to an efficient device. CAF consists of two major parts: 1) the Queue Management Device (QMD), which is attached to the NoC and works like an accelerator core to perform various queue management functionalities; and 2) three new instructions added to the ISA to accelerate dequeue, enqueue, and data movement operations. We will first discuss the new instructions, then discuss QMD design.

3.1 New Instructions

CAF adds three new instructions to the ISA:

- `enqueue reg, qid_reg`, where `reg` is a 64-bit operand containing the data to enqueue, and `qid_reg` is the concatenation of queue ID, priority, or other operation flags. The flags can be used to specify various operation modes. The QMD can decode the request by using simple masks.
- `dequeue reg, qid_reg`, where `reg` is the destination register to hold the value returned by QMD. `qid_reg` contains the queue ID, priority, and other operation flags.
- `prepush mem`, where `mem` specifies memory block that will be invalidated in the local cache and pushed to the shared last level cache.

enqueue and *prepush* are used for C2C communication through the QMD. They differ slightly from store instructions. As store-like instructions, the new instructions execute non-speculatively and go through the store buffer. When the instructions arrive at the head of the reorder buffer (ROB), if their operands are ready, they can be retired. Retired *enqueue* and *dequeue* instructions are held in the store buffer until they get responses from the QMD. After getting responses, they are regarded as completed and can be removed from the store buffer. In contrast to regular stores, dequeue uses the writeback stage to write data to the destination registers. In addition, enqueue and dequeue do not access memory so they do not incur page faults.

prepush request behaves like the inverse of a *prefetch* request, hence the name. Instead of fetching a cache block from a lower level cache to the upper level caches, *prepush* will push a cache block to the shared cache and invalidate the copy in its private caches. *prepush* is similar to stores and updates from literature [3, 30], speculative push [29], or forwarding write [16], pushing stores [27], however *prepush* is non-binding and because it only pushes data block to the shared cache, it reuses the writeback path of the cache hierarchy. Basic *prepush* is similar to early write back [19], but applied only to private caches.

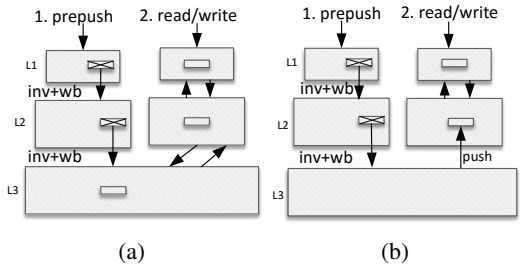


Figure 2: Two prepush schemes: basic (a) and advanced (b).

After *prepush*, another core that reads or writes to the pushed block can read it from shared cache instead of triggering coherence-based data forwarding or invalidation requests, saving a large number of cycles and NoC bandwidth. The process is illustrated in Figure 2(a). An alternative way to implement *prepush* is to push the cache block directly to the target core's local cache as shown in Figure 2(b). This will be discussed more in Section 5. We implement *prepush* as non-speculative. Similar with a store instruction, *prepush* also goes through store buffer.

To illustrate the use of the new instructions, consider network applications where producers and consumers rely on these enqueue or dequeue to communicate the pointers to the network packets. *prepush* can be used to reduce the data plane overhead incurred during queue-based communication. After the pointer of a packet is enqueued, the producer core can prepush the content of the packet to the shared cache so that the consumer core can get data from the shared cache and avoid expensive coherence and data forwarding.

Memory ordering. We assume a system implementing relaxed consistency model which relaxes load-to-store ordering (e.g. Intel IA-32 and SPARC total store order (TSO)). This is also the consistency model assumed by gem5 [4] simulator, which we used for our simulation. Generally speaking, the x86-TSO ordering principles include: a) a store is not reordered with respect to other stores; b) a store is not reordered with older loads; c) a load may be reordered with older stores to different locations. With our new instructions, additional ordering principles should be followed: a) a dequeue request is not reordered with other dequeue requests and an enqueue request is not reordered with other enqueue requests if they access the same queue; b) an enqueue request is not reordered with older stores; c) a store can be reordered with a *prepush* if the prepushed cache block address is different from that of the store.

Queue requests from a thread (or core) for a particular queue must arrive at the QMD in the program order. One way to achieve this is to use an ordered NoC, i.e. using routing that guarantees ordered message delivery such as deterministic single-path routing. It is also possible to use non-ordered NoC with an additional mechanism (e.g. scratch buffering resource and packet reordering based on sequence numbers) to restore the message order. At the processor pipeline, queue requests are not reordered with earlier stores. For example, if the program first stores data to a memory location, then the latter enqueue must happen after the earlier store is completed. As a result, the enqueued address always points to the updated data. Similarly, *prepush* is not reordered with an older store to the same block address so that the pushed block will have the latest value. Additional ordering between queue instructions and others can be enforced by using memory fence instructions.

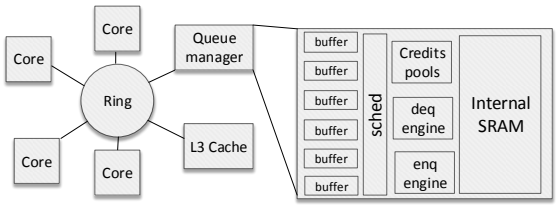


Figure 3: Queue management device architecture.

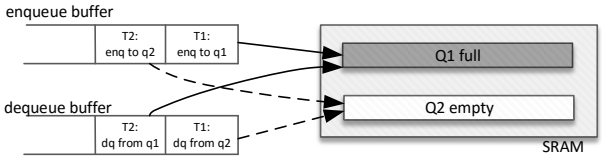


Figure 4: Illustration of a possible deadlock situation.

3.2 Queue Management Device

As shown in Figure 3, QMD is attached to the NoC. The figure shows a ring as an example but other NoC topologies are possible. When an application wants to use a queue, it makes a system call to request a queue to be allocated. The OS checks QMD for available hardware resources. If the allocation succeeds, the OS returns a queue ID to the application. Otherwise, an error message is returned. The application also makes a system call to deallocate a queue.

After queue allocation, the application can start using the queue. Enqueue and dequeue requests sent out by a core will be routed to the QMD. After each successful enqueue or dequeue operation, QMD sends a response back to the core. We describe QMD's hardware components in detail as below.

1. *Request buffers.* Every enqueue and dequeue request will be put into one of the request buffers. These buffers hold incoming requests temporarily until the QMD is able to serve them. In a basic design, enqueue requests go to one buffer and dequeue requests go to another. In a more advanced design, to enable fine-grained Quality of Service (QoS), we can associate each buffer with a priority. With n priorities, there are n enqueue buffers and n dequeue buffers.

2. *Scheduler.* The scheduler is responsible for removing requests from the buffers and send them to the enqueue or dequeue engine. In each cycle, a buffer is selected, and then the scheduler removes the request from the head of the selected buffer. The buffer selection can be accomplished using various policies, such as Round Robin (RR) or Weighted Round Robin (WRR). With WRR, the scheduler consider priorities of buffers into account. After scheduling, the enqueue or dequeue engine reads from or writes to the internal SRAM where the queue is managed. If the queue is full (for an enqueue), the request tries again in the next cycle. Similarly, if the queue is empty (for an dequeue), the request tries again in the next cycle. One caveat with this scheme is that one thread's requests may potentially block another thread's requests in the same buffer, thus a deadlock can occur as illustrated in Figure 4. In the figure, thread T1's enqueue request blocks thread T2's enqueue request while T1's dequeue request can only be performed after T2's requests is completed. To address this, one option is to have a dedicated set of buffers for each core. Alternatively, we can have a single set of virtual buffers for each core, but physically all the cores share a set of physical buffers. A simple arbitration (e.g. round-robin) between virtual buffers is sufficient.

3. *Queue storage.* The internal SRAM supports a configurable number and size of queues. In our configuration, each queue entry is 8 bytes, equal to the size of a pointer in x86-64. Thus, a 16KB SRAM can support one big queue with 2K entries or 16 smaller queues each with 128 entries. QMD maintains internal registers for the head and tail of each queue, which will be updated after processing a queue request. After successfully completing a queue request, a response is sent out to the response port, which is then routed back to the core that sent out the request.

4. *Credit management module.* In the basic mode, QMD operates without credit management. In this mode, queues behave as blocking when empty or full so that threads can keep sending requests until the cores stall. However, non-blocking behavior may be preferred in some cases. In this scenario, the credit management feature is used to inform the program how many resources in the QMD can be used by each thread. The number of credits given to a producer corresponds to the number of empty entries in the target queue. A thread should obtain credits before sending any enqueue or dequeue requests. Each thread may be granted different numbers of credits to control their traffic. We will discuss the credit management in Section 3.5.

5. *Interrupt.* QMD can send interrupts to cores. This is required to avoid the situation that when one thread crashes, the corresponding threads that is working on the same queue might be frozen. At the software side, a guarding thread may be used to inform QMD to send out an interrupt whenever necessary. The core receiving the interrupt drains all the incomplete queue requests. A timer in the QMD may be used to determine a time out when QMD may have blocked some cores. For example, if no request is successfully fulfilled for a long time from one buffer, QMD may drop requests in the buffer and send out an interrupt to the corresponding core.

6. *Others.* Other modules can be integrated with QMD to provide additional features such as data reducing and job distribution. The additional modules can be used and configured by enqueue/dequeue instructions with special flags or bit masks as the second operands. In Section 3.5, we will discuss several possible additional modules.

3.3 Power and Area Overhead

We use analytical power and area modeling tools to estimate the power and area overheads of QMD. We use both CACTI [36] and ORION [15] together to model the QMD. CACTI estimates power and area of the internal SRAM, and ORION estimates the power and area of the buffers, the scheduler, and the crossbar in the QMD. We leave out special modules such as the credit manager and the load balancer since they are optional. For a 45 nm technical node, we model a 32KB SRAM with 8 byte block size, and 2 routers (one for enqueue and one for dequeue) each with 3 buffers and a round robin scheduler. Our modeling tool shows 0.04 watts for the routers and 0.09 watts for the internal SRAM at full load. For the total area, the routers consume $0.03mm^2$ and the SRAM requires $0.71mm^2$, which means that the area cost is on a par with a core's first level cache.

3.4 Distributed QMD

Although QMD can offload a huge amount of work from core and memory, saving power, bandwidth, CPU cycles, etc, it still has limited hardware resources to handle all the requests. Like any shared hardware component, large amount of requests may saturate QMD. Fortunately, the QMD can easily be scaled by distributing it. One can have several QMDs on the network with each as a

complete functional QMD as in the centralized design. Each QMD will contain numbers of physical queues in its internal SRAM just like in the centralized QMD. Each queue of the QMDs will have its unique queue ID for thread to access. The number of QMDs is determined by the area and power budget during the chip design process. We will discuss more on this topic in the evaluation and extension sections.

3.5 Advanced Features

The basic QMD design can accelerate simple queue operations. Some advanced features can help QMD do work that might be costly or impossible to be done in software. In this section, we discuss examples of advanced features that can be added to the QMD.

Credit based flow control. The first feature is *thread level credit based flow control*. Sometimes, programmers may want to control the throughput of certain threads due to power, priority, or fairness concerns. Thread-level flow control is not easy to implement in software but can be added to the QMD credit management module. With credit management, a thread must reserve credits before sending out a request. The credits represent the resources allocated to a thread. If a thread sends more requests than the number of credits it has obtained, QMD will drop the extra requests silently. Software is responsible for recording credits it has obtained or consumed. For implementation, the dequeue instruction with a special flag is used to obtain credits for a certain queue. Although this mode adds additional credit reservation and bookkeeping overheads, it has certain benefits: a) it is useful for programs requiring non-blocking behavior, b) misbehaved threads cannot take over all the resources of QMD and starve other threads, and c) the credit mechanism enables another level of flow control among all threads, i.e., by giving a different number of credits to each individual thread, the speed of a thread can be controlled precisely. Thus, the maximum service time for each thread can be guaranteed.

The credit management module is designed to maintain multiple credit pools for each queue, including local and global credit pools, in its internal hardware registers. A thread can only obtain credits from its own local credit pool. There is also a global credit pool which maintains credits that have not been distributed to the local credit pools yet. In other words, the local credit pool represents the amount of resources a thread can use, while the global credit pool indicates the resources QMD holds. When a producer or consumer thread requests credits, it can only request from its local credit pool. The local credit pool will reduce the number of credits from its pool accordingly. In our current design, we let the local credit pool give out all its credits to any credit obtaining request to maximize throughput. When the enqueue or dequeue takes place in QMD, the global credit pool will be replenished first. For example, an item that is enqueued will increase the global dequeue credits by one. Then, QMD decides how credits will be distributed from the global credit pool to local credit pools. Different credit distribution policies can be used. For example, a WRR policy distributes credits to local credit pools in a weighted round robin fashion based on predefined thread priority. Under such a policy, no matter how fast or slow one thread is, the ratio of the maximum amount of service time for each thread is fixed. The credit distribution and replenishment are handled by hardware, so no software overheads are incurred.

Load balancer. Besides pipelined packet processing, another common use of queue is for distributing jobs (tasks) to different worker threads [17, 5, 31, 20]. The goal of a job distributor is to keep all cores busy. There are several ways to achieve this. One way to implement this is to have a single job queue. All workers draw jobs from this queue. After completing a job, a worker thread will dequeue another job. This keeps all workers busy at all time. However, using a single job queue means the jobs are not distinguishable. If certain jobs can only be handled by certain workers, a different scheme is needed. A single queue also suffers from scalability issue even though queue operations are processed by hardware. So, an alternative method is to allocate one queue to each worker thread. Special jobs can be enqueued to the queues for specific threads. Multiple queues also makes it more scalable with a large number of threads. However, the job producer needs to do load balancing in software. Without the knowledge of the status of each queue, it is difficult for the producer thread to distribute jobs to different queues optimally. To solve this issue, QMD could include a load balancer to help balance the requests for different queues. Figure 5(a) shows the concept of this component. In the load balancing mode, queue requests will be automatically distributed to different queues. For example, the load balancer will search for the queue with the smallest number of items and send the next enqueue request to it. With this feature, programmer can create a job distributor quickly without much software complexity. A configuration step needs to be performed to set up the distribution mapping.

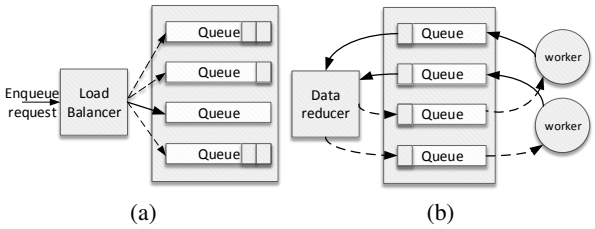


Figure 5: Illustration of load balancer (a) and data reducer (b).

Data reducer. If power and area budget allows, a more aggressive feature is to build a data processing unit into the QMD. For example, a data reducer is designed to help with in-memory Map Reduce [8, 7, 14] applications. Map Reduce is a popular programming model for parallel computing such as machine learning algorithms. In the Map Reduce model on a CMP, a master core is responsible for assigning (mapping) data to different workers. After the workers complete computation, the master collects the results from each worker and reduce them. In a large class of machine learning algorithms, the training procedure consists of a loop of a Map Reduce process [32]. The loop is iterated until the parameters converge. Thus, the machine learning algorithms may incur a lot of inter-core communications, especially with a high core count. Usually, the reducing function is simple and can be directly handled by hardware accelerators. Using a general purpose core to handle the task is wasteful. QMD would be a good accelerator for such task. Figure 5(b) shows a data reducer engine built into the QMD to act as the master core. Each worker enqueues the map results to the QMD and retrieve new data from the QMD for the next round of calculation.

Other features for QMD are possible but beyond the scope of this paper. We will evaluate certain advanced features and show QMD's effectiveness in the evaluation section.

4. EVALUATION

4.1 Modeling Details

We use the detailed mode of a full-system cycle-accurate simulator based on gem5 [4] to model CAF. We added the three new instructions into the x86-64 ISA and all their pipeline stages and memory behaviors are modeled at the cycle-accurate level. We use gem5 classic cache model but tune access latencies to approximate

Table 1: Baseline configuration.

Core	16 cores, each 2GHz, x86-64 ISA, 8-wide out-of-order superscalar Num. of entries: IQ:64 LQ:72 SQ:64
Cache	Non-inclusive property, MOESI protocol 64B block size, LRU replacement policy Private L1-D: 32KB, 8-way, 4-cycle latency Private L2: 256KB, 8-way, 10-cycle latency Shared L3: 24MB, 24-way, 45-cycle latency
QMD	50ns SRAM access latency, fully pipelined, 1GHz
DRAM	Single channel 8GB DDR3

the behavior of an Intel directory-based multi-core processor. We attach a centralized QMD to a dedicated interconnect which directly connects to each core. We use 50 CPU cycles as the round trip time between cores and the QMD, which is larger than the measured round trip time to the L3 cache on a real platform. Buffers and schedulers are modeled at the cycle-accurate level so that contention of requests at QMD is properly accounted for. The enqueue and dequeue engines take 50 ns to access its internal SRAM, which is rather conservative compared to access time of a cache. To make the simulation time manageable, we limit our detailed simulation to a system with 16 cores connected to a shared bus. We also simulate a 64-core distributed QMD design connected to a 2D mesh. Table 1 shows the baseline architecture configuration we use for detailed simulation throughout the evaluation section.

4.2 Queue Application Results

We wrote micro-benchmarks to evaluate the performance of CAF against a state-of-the-art software queue algorithm from the DPDK [2] library, which is widely used in today's data centers for packet processing workloads. A simplified version of the algorithm is shown in Algorithm 1. Other popular software queue algorithms in research literatures are more or less variants of this algorithm and are not promising better performance, especially on an x86 system. The algorithm also uses bulk processing to amortize queue maintenance overhead, but as we mentioned in Section 1, real applications always prefer small or no bulk. Therefore we use bulk size up to 8 in most of our experiments. Furthermore, going from a bulk size of 8 to 32, the software throughput improves by only 40%, due to diminishing returns.

The queue algorithm was modified minimally to run in our simulation environment [1]. We also compare CAF with HAQu in the single-producer single-consumer case since HAQu does not support concurrent queues. To see the raw performance improvement for different applications, we use the basic operating mode of QMD without advanced credit management mechanism. We tested the credit management in Section 4.5.

1. **Pure QMD dequeue speedup test:** We use a micro-benchmark to test the speedup of pure dequeue operations of the QMD. We pre-populated the queue structures and then used multiple threads to dequeue all the items. After each dequeue, the items are discarded by the consumer, thus the test does not incur any data plane overhead. We fixed the number of total items and increase the number of threads. We used a bulk size of 8 to amortize the control plane overhead of the software implementation. We recorded the time until the queue became empty, i.e. the execution time of the slowest thread, normalized to the software queue's 2-thread case,

[1]We use gem5's magic instructions to protect critical code section in the multi-producer multi-consumer queue case, due to gem5's lack of support for modeling x86's atomic instruction, at the time we conducted the research. The magic instruction is expected to be faster than an atomic instruction.

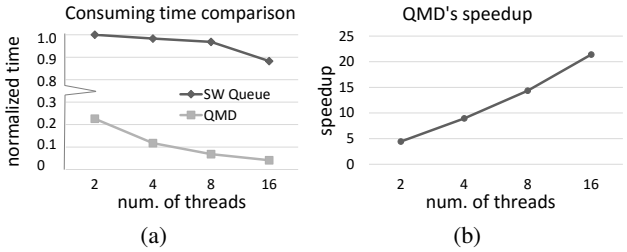


Figure 6: Normalized execution time of CAF vs. software queue (a), and CAF speedup over software queue (b).

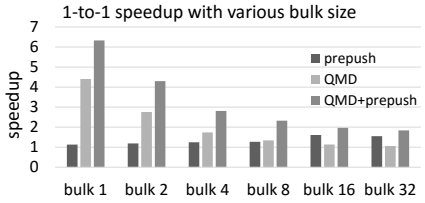
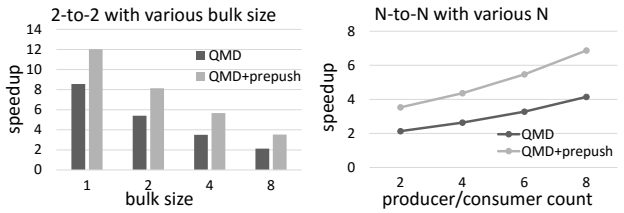


Figure 7: Speedup of CAF for a single-producer single-consumer workload.

and show the results in Figure 6(a). By measuring the execution time of the slowest thread (until the queue becomes empty), the speedup we measure is also throughput speedup. For the rest of the paper, we measure throughput speedup in a similar fashion. We may guess that the execution time will be inversely proportional to the number of threads as more threads will consume the queue faster. However, the software queue execution time only decreases by about 10% going from 2 to 16 threads. This is due to the contention caused by multiple threads accessing the global variables. On the other hand, CAF's execution time decreases by about 80%, or a speedup factor of 5×. This is because the software-related contention is eliminated when queue operations are offloaded to the hardware. For comparison, Figure 6(b) shows the speedup of CAF relative to software queue on equal numbers of threads, with varying number of threads. Confirming the previous observation, the speedup increases as the number of threads increases, reaching as high as 20× speedup in the 16-thread case.

2. **One-to-One communication:** In order to consider a more realistic workload that includes data plane overheads, we first created a benchmark to represent a single-consumer single-producer use case. In this packet processing benchmark, a memory pool is managed in a queue from which the producer can allocate new packets. After the packets are allocated, the pointer to the packet is inserted into another queue waiting for the consumer to dequeue. The consumer dequeues the pointers and fetches the pointed packets to perform a simple arithmetic operation. Afterwards, the memory space used by the packets is returned to the memory pool. This use case represents basic packet processing applications. In the software queue implementation, we enable bulk processing with various bulk sizes to amortize the control plane overhead. In the CAF implementation, we use QMD for queue management and also issue *prepush* following each enqueue request. Figure 7 shows the speedup of CAF compared to the software queue implementation with various bulk sizes. The figure shows QMD's speedup (representing control plane speedup) and the *prepush*'s speedup (representing data plane speedup) separately. The results show that as the bulk size increases, the control plane speedup becomes less significant as it is increasingly amortized by the larger bulk size. When



(a) (b)

Figure 8: The N-to-N benchmark workload's speedup with CAF with various bulk sizes (a), and with increasing numbers of threads (b).

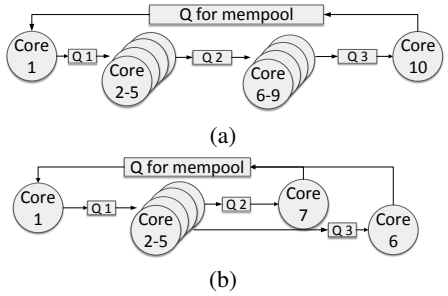
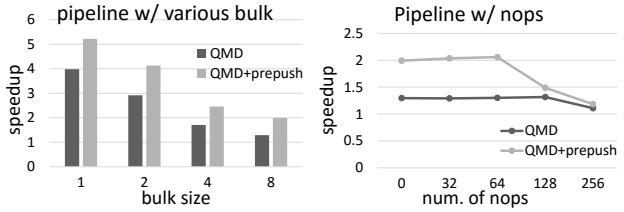


Figure 9: Pipelined/clustered hybrid application (a), and firewall application (b).



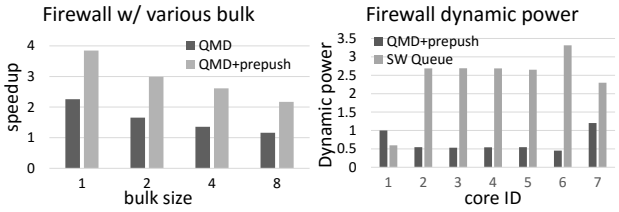
(a) (b)

Figure 10: For the pipeline benchmark, the speedup of CAF with various bulk sizes (a), and the speedup with various nops inserted (b).

the software queue does not use bulk, CAF achieves more than 6× speedup. This means that CAF is very effective for more real-time applications such as telecommunication applications which prefer no bulk. With a bulk size larger than 8, CAF still achieves about 2× speedup, however a smaller bulk is preferred in real applications since a larger bulk may lead to a longer latency where critical packets may be delayed for a long time waiting for the composition of a large block of packets.

3. **N-to-N communication:** This is similar to the single-producer single-consumer case, but with multiple producers and consumers. First, we used two producers and two consumers but vary the bulk size from 1 to 8. Figure 8(a) shows that the speedup is up to 12×. Then we fixed the bulk size at 8 and varied the number of producers and consumers. The speedup results are shown in Figure 8(b). Due to the high overhead of global variable contention in software queues, CAF achieves about 7× speedup in the case of 8 producers and 8 consumers. QMD-only also achieves about 4× speedup. This again shows that CAF is especially helpful in the multi-producer multi-consumer case, and it scales better than software queue.

4. **Hybrid packet processing: pipeline of interconnected clusters.** Figure 9(a) shows a more complex packet processing scenario, with two clusters of threads working at different pipeline stages. The job is thus pipelined and distributed to different cores to maximize throughput. As noted in [9], this is a common packet processing scheme on general purpose x86 platforms. Figure 10(a) shows the speedup with various bulk sizes. CAF shows up to 5× speedup when we use small bulk sizes. Then we fix the bulk size at 8 and insert *nop*s in each worker thread to mimic different amounts of packet processing work. The results are shown in Figure 10(b). We can see that when the workload is light, CAF can achieve 2× speedup. When the amount of work is large, the benefit of CAF diminishes. This is because the packet processing work itself eventually becomes the bottleneck of the benchmark and CAF is not designed to accelerate the core's execution speed. However, in a largely distributed application, the amount of work for each core is expected to be relatively light.



(a) (b)

Figure 12: Performance evaluation of the firewall application on CAF with various bulk sizes (a), and the per core dynamic power consumption (b).

5. **Firewall packet filtering:** In this benchmark, we emulate the behavior of a firewall when processing incoming network traffic. The application scheme is shown in Figure 9(b). Core 2 to core 5 are responsible for filtering the packets based on firewall policies. Any suspicious packet is rerouted to core 6 for further analysis or bookkeeping, and then dropped. Normal packets are passed to core 7. To mimic real use cases, each packet occupies a 2KB memory buffer, with the first three 64-byte blocks in the 2KB buffer representing the packet header. The packet header will be passed from core-to-core for examination. In our experiments, on average every 8 packets, core 1 sends out a "suspicious" packet with one of the header locations set to a special value. One can regard the value as an IP address or a port number. The workers (core 2 to core 5) check this value to see if it is suspicious. Similar to the previous test cases, core 6 and core 7 return the used memory buffers back to the memory pool when packets are dropped by core 6 or consumed by core 7.

Figure 12(a) shows the speedup. We vary the bulk size from 1 to 8 and find up to 4× speedup. To further show the benefit of CAF, we fix the bulk size as 8 and collect power and cache performance statistics. We use McPAT [24] to model the dynamic power consumption of each core. Figure 12(b) shows the dynamic power consumption and Figure 11 shows each core's detailed performance data including dynamic instruction count, data cache access count, data forwarding count, and data invalidation count. Since a core could be stalled instead of spin waiting for available queue resources when queue is blocking, dynamic instruction count and data cache access count are significantly reduced by using CAF, halving the dynamic power of the whole processor. Also, CAF helps eliminate a large amount of coherence-based communication overhead including both data forwarding and data invalidation, which means considerable NoC bandwidth and power could also be saved.

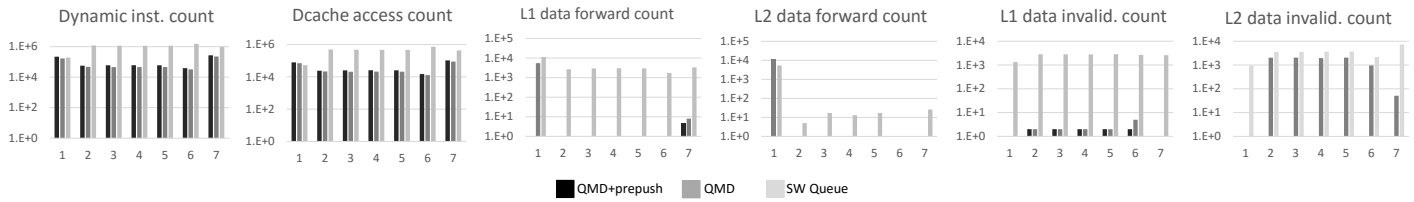


Figure 11: Comparison of QMD (with and without prepush) against the software queue implementation using various statistics. The x axis shows the core ID corresponding to Figure 9 (b). The y axis is shown in logarithmic scale.

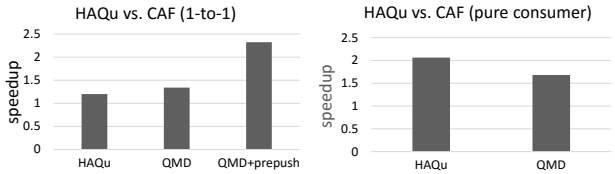


Figure 13: Comparing HAQu and CAF throughput on the single-producer single-consumer benchmark evaluation (a) and pure consumer benchmark (b).

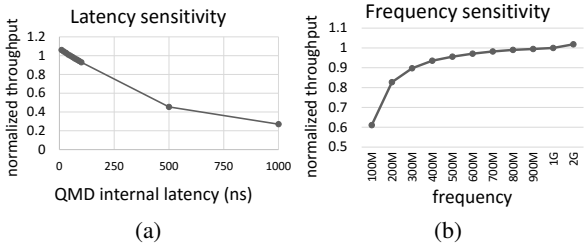


Figure 14: Throughput with various latencies (a), and with various clock frequencies (b).

6. **Comparison with HAQu:** HAQu [22] is a state-of-the-art hardware queue acceleration approach, but it does not support the multi-producer multi-consumer applications or any advanced QMD features. Thus, we only compared our design with HAQu for the single-producer single-consumer benchmark. Figure 13 shows the comparison of CAF and HAQu. Fundamentally, HAQu does not eliminate any overhead associated with accessing the queue heads and tails. In other words, heads and tails are still stored and managed in shared memory. Thus, in the single-producer single-consumer case in which the heads and tails of the queue are updated frequently, HAQu performs worse than QMD (1.2× vs. 1.35×). However, HAQu does not have the latency overhead to access the queue content given that the queue information is kept in the core. Dequeue requests in HAQu only require the overhead of a regular cache access. In the pure consumer test, there are no frequent updates to the heads and tails, therefore the HAQu performs better than the QMD (2.1× vs. 1.7×). However, the pure consuming benchmark is less realistic. Also, since HAQu does not support multi-producer multi-consumer cases and requires substantial modifications to cores, CAF is a more appealing solution for general purpose platforms.

4.3 Sensitivity Study

For sensitivity study, we vary the latency, i.e. time for the enqueue or dequeue engine in QMD to access its internal SRAM, from 10 ns to 1 ms, while keeping the clock frequency at 1GHz. Then, we change the frequency from 100MHz to 2GHz, while keeping the latency at 50 ns. We use the single-producer single-consumer benchmark as our test case. Figure 14(a) shows the throughput changes by changing the internal latency and Figure 14(b) shows the throughput changes by changing the frequency. For the latency sensitivity results, the throughput changes almost linearly when the internal latency is less than 500 ns. There is about 10% difference between 100 ns and 10 ns latency. The number is not large because multiple dequeues and enqueues can be in-flight simultaneously, which amortize the latency for each request. For the frequency sensitivity results, we see that beyond 500MHz, the throughput does not increase much, but from 300MHz to 100MHz, there is a big drop. This is because frequency determines how fast the QMD can pick up requests from the buffers. Only when buffers are stressed, increasing frequency will improve the system's throughput.

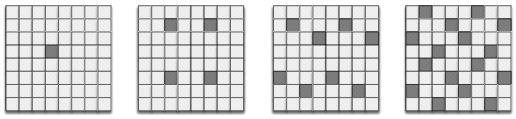


Figure 15: A possible placement of 1, 4, 8, and 16 QMDs on a 64-core mesh.

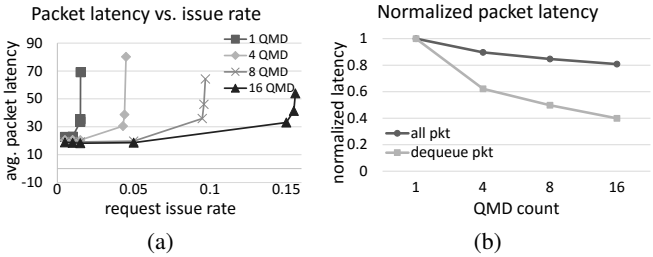


Figure 16: The average packet latency with various issue rates (a), and the average latency comparison for all packets and dequeue-only packets (b).

4.4 Distributed QMD

As we mentioned in Section 3.4, a centralized QMD may be saturated by a large amount of requests, which will affect its performance. The result in Figure 14 also shows the effect: when QMD's speed decreases by 10 times from 1GHz (similar to a requests rate increaseing by 10 times at 1GHz), the performance decreases by 40%. To solve this problem, we introduce the distributed QMD design. One can regard our previous 16-core evaluation as part of a larger system with distributed QMDs.

To evaluate the benefit of distributed QMDs over a centralized

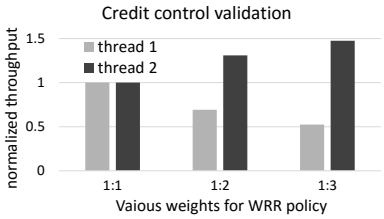


Figure 17: Throughput of two threads with a weighted round robin policy.

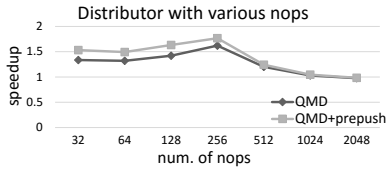


Figure 18: CAF speedup for the distributor application.

QMD, we model a 64-core system based on the BookSim [13] simulator. Each core is connected to a router in a 8×8 mesh. We model each core to issue enqueue and dequeue requests to the QMDs with certain issue rates. We let the number of enqueue to be equal to the number of dequeue requests and assume a policy that the physical queues used are always located close to the consumer cores. In other words, enqueue could be issued to any QMD, but all the dequeues will be issued to the nearest QMD. We assume this policy because dequeue instructions usually stay in the critical path of an application. We use 1, 4, 8, and 16 QMDs placed as shown in Figure 15. For 8 and 16 QMDs, the placement is optimized for the minimum hop distance according to [37]. We vary the average issue rate of each core and collect the average packet latency. Figure 16(a) shows the average packet latency against the issue rate with different numbers of QMDs. When the latency increases dramatically, it means the QMD cannot finish processing each request fast enough so that the network is saturated. By using 16 QMDs, the system can support more than $10\times$ higher requests issue rate and achieve lower packet latency, meanwhile sustain the throughput of each thread. Figure 16(b) shows the average packet latency of all packets and dequeue-only packets for different numbers of QMDs. We found that the dequeue spend 60% less time on the network scaled to include 16 QMDs.

These results show that if there is no high volume of queue requests, a centralized QMD would be enough, although hop distance could be high for certain cores. Meanwhile, a high volume of queue access requests could saturate the network and in turn affect the performance. In future many-core CMP systems, a distributed QMD design may be necessary.

4.5 Advanced Features

1. **Evaluation of credit management system:** The credit management system enables *thread-level flow control* and non-blocking behavior as discussed in Section 3.5, although an extra credit obtaining step is needed. To test the effectiveness of the flow control, we run a test with two producers and one consumer working on the same queue. Each producer sends out a credit obtaining request first. According to the credits it obtains, it will then issue a matching number of enqueue requests. After credits are consumed, it will request for credits again. We set the credit control system to implement a weighted round robin policy for distributing the credits to the local credit pools according to their weights. Other policies are also possible. Figure 17 shows the normalized throughput of the 2 producer threads. By setting the weights to be 1:1, 1:2, and 1:3, the throughputs of the two threads are changed accordingly. Note that there is no good way to control traffic in software unless using expensive synchronizations and communications.

2. **Distributor application:** By including a queue balancer, we can use the QMD to work as a job distributor as we discussed in 3.5. To show how fast that QMD can optimally distribute jobs to different workers, we first implement a software solution in which the producer loops around worker threads to distribute jobs to the idle ones. To avoid unnecessary overhead, each worker thread only has one shared variable with the producer to store one incoming job. When one worker finishes its job, it will mark itself as idle, so that the producer will find and put a new job in its shared variable. Without maintaining queues, this software distributor is simple and fast. We will compare the throughput with our QMD implementation.

With QMD, we set up a specific queue for each worker. In the load balancing mode, the load balancer module automatically distributes jobs to the worker with the least jobs waiting. To compare the results, we use four workers and insert different numbers of *nop*s to the workers to mimic different amounts of workload. Figure 18 shows the speedup of QMD against the software implementation. For these tests, we see up to $1.8\times$ speedup. Notice that we do not even use queues for buffering packets in the software implementation. If queue management and queue status check overhead is involved, the speedup of CAF may be even higher.

3. **Machine learning algorithm:** As discussed in Section 3.5, we may add a data processing engine into QMD to accelerate applications with Map Reduce programming model. To show the benefit, we use stochastic gradient descent (SGD) to solve logistic regression (LR) [6, 28] algorithm which is a typical method used in classification and language processing. Although we only test a single algorithm, many other machine learning algorithms could be abstracted into a similar form [35]. The algorithm we are using is:

$\theta_j := \theta_j + \varepsilon \sum_{i=1}^{m} (y^i - h_\theta(x^i)) x_j^i$

In this equation, θ is the vector of the parameters (or weights). ε is the step size. y^i is the corresponding label for data vector $\vec{x^i}$. h is the hypothesis function. In each iteration, a new θ will be calculated and used for the next iteration. With Map Reduce, each worker works on a part of the data vectors. The master core sums the results from each worker and updates θ. In the software implementation, we synchronize the workers and the master at the end of each iteration so that they proceed in lock-step. In the QMD implementation, we let each worker enqueue its result to the QMD. We add a data reducing engine into QMD to do a reducing (summation) of all the enqueued data. The engine is also responsible for copying the reduced data back to each queue. Then the workers will dequeue the new data and proceed to the next iteration. For the implementation, we consider one cycle for the data reducer to read one item, one cycle for an arithmetic operation, and one cycle to store the data back. We input a set of vectors each having 13 attributes from the Statlog (Heart) Data Set [25]. We change the number of workers and the number of input vectors handled by each worker for evaluating the benefit.

Figure 19 shows that QMD achieves 15% speedup when worker count is as large as 16, and each worker only works on a small subset of the data (one row of the input matrix). This is because the communication overhead will take a larger portion of the overall overhead in this case. Meanwhile, if the communication overhead does not dominate, the benefit is small. We believe QMD has the

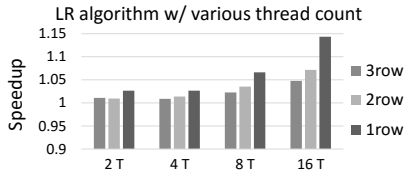


Figure 19: Throughput of the Map Reduce application.

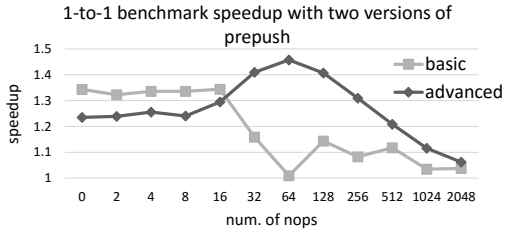


Figure 20: The speedup comparison between basic and advanced *prepush* implementation.

potential to benefit machine learning applications in future many-core system with a large number of processing elements.

5. EXTENSIONS

We already discussed the general scheme and advanced features of CAF. In this section, we will discuss some possible extensions to the current design and provide some preliminary results. These extensions include a more advanced *prepush* implementation, queue protection, and queue virtualization.

5.1 Advanced Prepush

In Section 3.1, we introduced the *prepush* instruction with a basic implementation of pushing data into the shared LLC. A more advanced implementation of *prepush* is to push the data directly into the destination core's private caches (illustrated in Figure 2(b)). New store or update instructions proposed by [27, 16, 23, 3, 11, 30] have a similar behavior. One choice we made is to push data to the private L2 cache of the destination core instead of the private L1 cache, in order to avoid unnecessary pollution to small L1 cache. One major concern of this mechanism is how to recognize the core to push the data to. In this discussion, we will assume a simple approach which allows the programmer or compiler to specify the destination core, and we set up core-affinity for the threads. Compared to the previous *prepush*, the new *prepush* instruction now takes the destination core ID as another operand.

We implemented the advanced *prepush* and evaluated it with the single-producer single-consumer benchmark as our test case. In this benchmark, the producer is initially slower than the consumer, so we add *nop*s to the consumer thread to change the relative speed of the producer and the consumer. Figure 20 shows that when the producer is relatively slow, the advanced *prepush* does not benefit more than the basic *prepush*. This is because the new *prepush* only speeds up the consumer, however in this case the producer is the bottleneck. Also, since the advanced *prepush* needs to read the core ID and wait for the data to be pushed to another core, it slows down the producer. As we add more *nop*s to the consumer, the advanced *prepush* begins to show larger benefits. The benefit is from the cycles saved by the consumer to access private L2 cache instead of the shared LLC. Although there is an advantage with the advanced *prepush*, issues such as core ID determination and cache pollution need to be carefully addressed. Furthermore, the advanced *prepush* also complicates the hardware design and the coherence protocol, whereas the basic *prepush* utilizes the write back path of the cache hierarchy.

5.2 Queue Protection and Queue ID Virtualization

In complex general-purpose systems, multiple processes may co-execute and access queues concurrently. Previously, we assume there are no misbehaving processes so that we do not need to protect the queues used by one process from another process. With this assumption, a software queue allocation mechanism may be enough to distribute hardware queue resources to multiple processes. However, without proper protection, one misbehaving process may starve another process or steal items from a queue that does not belong to it. To enforce process isolation and enable queue protection, a hardware component is necessary to monitor all the enqueue and dequeue requests, which we refer to as queue ID management unit (QIMU). Queue ID ranges and permissions can be set up in QIMU for regulating the queue usage for each process.

Another benefit of introducing QIMU is that we can use it for run-time queue ID translation which provides queue ID virtualization support. With queue ID virtualization, the physical queue being used can be migrated from one QMD to another dynamically without being discovered by the user level program. After migration, the OS just needs to re-map the logical queue to the new physical queue. Dynamic queue migration enables the optimization opportunities of packet latency, NoC traffic, and QMD workload balance. For example, when a thread migrates from one core to another, the physical queue it uses should also migrate to the new core's nearest QMD. Analogous to a memory management unit (MMU), one issue of using QIMU for translation is that translating each queue request is costly. To solve this issue, a queue translation lookaside buffer (QTLB) is needed. Operations for TLB also apply to QTLB. Each queue access request will look up QTLB for virtual queue ID to physical queue ID mapping. If the translation is found, the access will be translated directly. Otherwise, the mapping in memory will be looked up and brought into the QTLB. Like a TLB, context switch requires flushing the QTLB.

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed and evaluated a core-to-core Communication Acceleration Framework (CAF) to significantly reduce software queue communication overhead and improve overall system performance. We proposed new instructions to accelerate control and data plane communication, and new hardware component to accelerate control plane communication. Our gem5 based simulation model shows up to $2-12\times$ speedup in test cases representing real usage patterns compared to an optimized software queue algorithm. We also show that CAF can support various advanced features to extend its usage to more universal applications like job distributor and machine learning algorithms.

7. ACKNOWLEDGMENT

Much of the work was done when Yipeng Wang was a research intern at Intel Labs and in collaboration with his Ph.D. advisor Yan Solihin. We would like to thank the anonymous reviewers for their helpful feedback.

8. REFERENCES

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

[2] Data Plane Development Kit: Programmer's Guide. Technical report, Intel Corp, 2015.

[3] W. Berke. A cache technique for synchronization variables in highly parallel, shared memory systems. In *Courant Institute of Mathematical Sciences*, 1988.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2), 2011.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[6] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics*, Paris, France, August 2010. Springer.

[7] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[9] C. F. Dumitrescu. Design Patterns for Packet Processing Applications on Multi-core Intel Architecture Processors . Technical report, Intel Corp., 2008.

[10] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, 2008.

[11] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[12] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin. Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers*, 2013.

[13] N. Jiang, D. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. Shaw, J. Kim, and W. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, April 2013.

[14] W. Jiang, V. Ravi, and G. Agrawal. A map-reduce system with an alternate api for multi-core environments. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing(CCGrid)*, 2010.

[15] A. Kahng, B. Li, L.-S. Peh, and K. Samadi. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In *Design, Automation Test in Europe Conference Exhibition*, April 2009.

[16] D. A. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(12), Dec 1996.

[17] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[18] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2), Apr. 1983.

[19] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[20] J. Lee, C. Nicopoulos, H. G. Lee, S. Panth, S. K. Lim, and J. Kim. Isonet: Hardware-based job queue management for many-core architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(6), June 2013.

[21] P. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *Proceedings of International Symposium on Parallel Distributed Processing*, 2010.

[22] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck. Haqu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *Proceedings of 17th International Symposium on High Performance Computer Architecture*, 2011.

[23] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The stanford dash multiprocessor. *Computer*, 25(3), 1992.

[24] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd International Symposium on Microarchitecture*, Dec 2009.

[25] M. Lichman. UCI machine learning repository, 2013.

[26] M. M. Michael and M. 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*, 1996.

[27] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim. Location-aware cache management for many-core processors with deep cache hierarchy. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[28] D. Pregibon. Logistic regression diagnostics. *The Annals of Statistics*, 9(4), 1981.

[29] R. Rajwar, A. Kägi, and J. R. Goodman. Inferential queueing and speculative push for reducing critical communication latencies. In *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[30] E. Rosti, E. Smirni, T. D. Wagner, A. W. Apon, and L. W. Dowdy. The ksr1: Experimentation and modeling of poststore. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993.

[31] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In

Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, 2010.

[32] B. Schölkopf, J. Platt, and T. Hofmann. Map-reduce for machine learning on multicore. In *Proceedings of Conference of Advances in Neural Information Processing Systems*, 2007.

[33] T. R. Scogland and W.-c. Feng. Design and evaluation of scalable concurrent queues for many-core architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015.

[34] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 2001.

[35] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[36] S. Wilton and N. Jouppi. Cacti: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5), May 1996.

[37] T. Xu, P. Liljeberg, and H. Tenhunen. Optimal memory controller placement for chip multiprocessor. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis*, Oct 2011.