HAQu: Hardware-Accelerated Queueing for Fine-Grained Threading on a Chip Multiprocessor*

Sanghoon Lee, Devesh Tiwari, Yan Solihin, James Tuck
Department of Electrical & Computer Engineering
North Carolina State University
{shlee5,dtiwari2,solihin,jtuck}@ncsu.edu

Abstract

Queues are commonly used in multithreaded programs for synchronization and communication. However, because software queues tend to be too expensive to support fine-grained parallelism, hardware queues have been proposed to reduce overhead of communication between cores. Hardware queues require modifications to the processor core and need a custom interconnect. They also pose difficulties for the operating system because their state must be preserved across context switches. To solve these problems, we propose a hardware-accelerated queue, or HAQu. HAQu adds hardware to a CMP that accelerates operations on software queues. Our design implements fast queueing through an application's address space with operations that are compatible with a fully software queue. Our design provides accelerated and OS-transparent performance in three general ways: (1) it provides a single instruction for enqueueing and dequeueing which significantly reduces the overhead when used in fine-grained threading; (2) operations on the queue are designed to leverage low-level details of the coherence protocol ; and (3) hardware ensures that the full state of the queue is stored in the application's address space, thereby ensuring virtualization. We have evaluated our design in the context of application domains: offloading fine-grained checks for improved software reliability, and automatic, fine-grained parallelization using decoupled software pipelining.

1. Introduction

Chip multiprocessors (CMPs) are now ubiquitous, but achieving high performance on a wide range of applications is still a big challenge. It is important to consider new architectural features that can expand the programmability and performance of CMPs. In this paper, we consider such a feature by adding architectural support for a *single-producer, single-consumer queue*.

Queues are a commonly used programming construct in multithreaded programs for synchronization and communication. They are effective when the granularity of parallelism is large enough to amortize the cost of enqueueing and dequeueing. But software queues tend to be too expensive to support fine-grained parallelism. For this reason, hardware queues have been proposed to reduce the overhead of communicating between cores. However, hardware queues tend to be unattractive to designers. They come at a large design cost as they require modifications to the processor core and need a custom, global interconnect. They also pose difficulties for the operating system because their state must be preserved across context switches. These challenges grow with ever increasing processor virtualization.

To solve these problems, we propose a hardware accelerated queue, or HAQu (read like *haiku*). HAQu adds hardware to a CMP that accelerates operations on software queues. Rather than adding a custom hardware queue, our design implements fast queueing through an application's address space with operations that are compatible with a fully software queue. Our design provides accelerated and OS-transparent performance in three general ways. (1) It provides instructions for enqueueing and dequeueing which significantly reduce the overhead of fine-grained threading. These instructions are critical for achieving fast queueing since they eliminate the high instruction count involved in queueing operations. (2) Operations on the queue are designed to leverage low-level details of the on-chip caches and coherence protocol to provide fast communication. (3) Furthermore, the full state of the queue is stored in the application's address space, thereby providing virtualization, queues in nearly unlimited quantity, and queues of a large size. On the whole, HAQu is no worse for programmability than a software queue.

We have evaluated our design in three contexts. First, we evaluate our mechanism using a micro-benchmark designed to show the peak performance of our architecture compared to state-of-the-art queueing algorithms in software. Our system is able to achieve a 6.5× speedup over an efficient software implementation proposed by Lee *et al* [9]. Secondly, we show how our design can facilitate fine-grained

*This work was supported in part by NSF Award CNS-0834664.

parallelization by parallelizing Mudflap, a pointer-checking utility built into GCC. Using HAQu, we can afford to parallelize fine-grained, inlined pointer checking codes. Using 16 cores, we achieve an average speedup of 1.8×, calculated using geometric mean, and a max speedup of 3.3× on a set of SPECint applications. Finally, we demonstrate the potential of HAQu for automatic parallelization using decoupled software pipelined (DSWP). On a small set of SPECint applications, we show that our mechanism enables fully automatic parallelization.

The rest of this paper proceeds as follows: Section 2 discusses the overall idea of our approach; Section 3 describes the architecture of the queue, and Section 4 provides the implementation. Section 5 describes our experimental setup, and Section 6 provides the full evaluation in the context of three case studies. Section 7 discusses related work, and Section 8 concludes.

2. Main Idea: Hardware Accelerated Queue

2.1. Motivation

Queues that support fine-grained parallelism on chip-multiprocessors must meet several criteria. We have identified four that would a make a queue acceptable for a wide range of uses. Throughout this article, we restrict our discussion to single-producer/single-consumer queues. First, (*Criterion I*) enqueue and dequeue operations must be made as efficient as possible. Since these operations may occur frequently, it's necessary that the total latency is low. Second, (*Criterion II*) the delay through the queue (i.e. the time between an enqueue and dequeue of the same item) must be low enough that concurrent execution proceeds efficiently. Third, (*Criterion III*) they should be no worse to program than software queues. For example, there should be enough queues available to support a wide variety of applications, and their synchronization constraints should be no harder to understand than for fully software implementations. Finally, (*Criterion IV*) the queue must work seamlessly on an unmodified operating system (OS).

The current state-of-the-art implementations for queueing are fully in software. Figure 1 shows an implementation of a Lamport queue designed for sequential consistency; part (a) shows the queue definition, and part (b) and (c) show the enqueue and dequeue operations respectively. The enqueue and dequeue operations shown here are non-blocking and lock-free; a blocking implementation is trivial to construct using these operations. For weaker consistency models, a fence is needed to properly read and update the queue state.

While these implementations are not well suited for fine-grained parallelism, they do meet most of the criteria above. In particular, they provide low latency for communicating through the queue because they often exploit the fast on-chip interconnect. This latency can be optimized through careful consideration of on-chip caches and the coherence protocol. They meet *Criterion III* trivially because they are implemented in software. *Criterion IV* is met trivially as a result of *Criterion III*; since the queues are stored in virtual memory, they are trivially virtualized by any OS. Software queues, however, do not provide enqueue and dequeue operations that are efficient enough for fine-grained parallelism; so, they do not meet *Criterion I*. Even with efficient non-blocking and lock-free implementations for enqueue and dequeue, the cost of these operations tend to be high — 10 instructions per queue operation.

Hardware queues have appeared in a variety of proposals [13, 11, 6]. Due to the similarities, we will treat them the same in this discussion. These designs support *Criterion I* and *Criterion II* . Using special instructions or special hardware, data is inserted directly into the hardware queue, and this makes enqueue and dequeue highly efficient. In fact, the latency of enqueue or dequeue can be hidden by out-of-order execution. Furthermore, the storage for the queue is provided by special buffers and an interconnect between two or more cores. This provides very low latency communication. However, these mechanisms are costly, and there are practical concerns over their programmability and the limits on the number of queues that can be supported (*Criterion III*). Furthermore, these hardware queues have not been designed with the OS in mind, and often do not consider the challenges of context virtualization. For example, how is the state of the queue preserved when one of the threads, either the producer or consumer, is context switched. Furthermore, allocation of global queueing resources may need to be managed by the operating system. Given the complexity of a modern OS, hardware that requires such changes in the OS is considerably less desirable.

Neither software nor proposed hardware mechanisms meet all of the criteria for fine-grained threading on current CMPs. However, the shortcomings of these software and hardware mechanisms point the way to a hybrid design that leverages strengths from each one.

2.2. Our Proposal: Hardware-Accelerated Queue

The Hardware-Accelerated Queue (HAQu) aims to exploit the critical benefits of hardware queueing while retaining the many desirable features of software queues. In a sense, we start with a software implementation and speedup the features that are critical for faster and more efficient fine-grained threading. Using hardware to accelerate queueing is not new to this proposal. The VAX [4] also supported hardware accelerated queues (see comparison in Section 7). However, we will show that HAQu provides a simple, lean implementation that integrates well into modern superscalar architectures; and it can provide much higher throughput and lower latency compared to optimized software implementations.

HAQu provides two new instructions for enqueue and dequeue operations to meet *Criterion I*. These new instruc-

```
struct queue {
   Item *head;
   Item *tail;
   uint  size; // = SIZE;
   Item  buffer[SIZE];
}

typedef struct queue * queue_p;
```

(a)

```
int enqueue(queue_p q, Item val)
{
   Item *next = NEXT(q->head);
   if (q->tail == next)
      return BLOCK;

   *head = val;
   // memory barrier
   q->head = next;
   return SUCCESS;
}
```

(b)

```
int dequeue(queue_p q, Item *val)
{
   Item *tail = q->tail;

   if (tail == q->head)
      return BLOCK;

   // memory barrier
   *val = *tail;
   q->tail = NEXT(tail);
   return SUCCESS;
}
```

(c)

Figure 1. Lamport's queueing operations single-producer, single-consumer queue on a sequentially consistent processor.

tions keep the code footprint low, and their implementation will work much like software enqueue and dequeue operations. However, because they are implemented in hardware, we incorporate detailed knowledge of the memory consistency model, coherence protocol, and on-chip resources to provide a highly optimized enqueue or dequeue operation.

HAQu stores the queue's state in the application's address space. As a result, *Criterion III* and *Criterion IV* nearly trivially are met. This means that we can support a large number of queues efficiently, and we can design HAQu to provide synchronization semantics that programmers expect. Also, since the state of the queue is held in virtual memory, a context switch poses almost no problems since the queue need not be saved or restored. Furthermore, HAQu can leverage the existing on-chip interconnect and cache coherence protocol for transfering data from producer to consumer. This usually provides low enough latencies to meet *Criterion II*.

Arriving at an efficient implementation is not as straightforward as it may seem. *Criterion I* and *Criterion II* are difficult to obtain without leveraging queue specific information in the hardware. We find that some information about the queue must be tracked in hardware. However, we can do so only if we preserve OS transparency and programmability concerns. Our architecture for HAQu balances the competing concerns of achieving a high-throughput implementation (*Criterion I* and *Criterion II*) while still providing an implementation that can be deployed on conventional CMPs that support a wide range of applications (*Criterion III* and *Criterion IV*) and programmability concerns.

3. System Architecture

To explain the overall architecture of our proposal, we will begin with a software queue and modify it to support enqueueing and dequeueing operations in hardware. There are three main components to this discussion: the allocation and layout of the queue in memory, enqueue and dequeue operations, and making these operations efficient for fine-grained threading.

3.1. Allocation and Layout

Our queue will reside in memory but will be operated on directly using hardware. Therefore, the layout of the queue is determined *a priori*[1]. For now, we will assume a layout similar to Figure 1(a). This is not an optimized configuration but is sufficient for the following discussion[2]. A client will allocate a queue using a memory allocator like *malloc* (in C) and prepare it for use. The head, tail, size, and buffer fields of the queue are located at fixed offsets from the beginning of the queue. We presume that the queue data structure and its initialization can be provided through a runtime library. Furthermore, since the buffer is the last field of the struct, the queue can be of arbitrary size.

3.2. Enqueue and Dequeue Operations

A key goal of HAQu is reducing the cost of enqueue and dequeue operations by adding two new instructions to carry out these tasks. A naive design directly implements Lamport's algorithm [8] in hardware.

Consider an enqueue operation. When an enqueue instruction is executed, it is passed the base address of the queue as an input argument. Using the base address, the head and tail fields of the data structure are read. From this point on, hardware implements Lamport's algorithm.

Figure 2 shows a possible timeline for the execution of queue operation. Since enqueue and dequeue operations are similar, we only describe the enqueue operation in detail. In cycle 1, the head and tail are loaded into temporary registers (these are not architectural registers or part of the register file). For this example, we assume that both memory operations hit in the L1 cache and return 2 cycles later. In cycle 3,

[1]This is a limitation of our proposal, but a handful of layout configurations could be supported and configured with firmware with little additional cost.

[2]An optimized implementation will pad the head and tail to fit on their own lines to prevent false sharing.

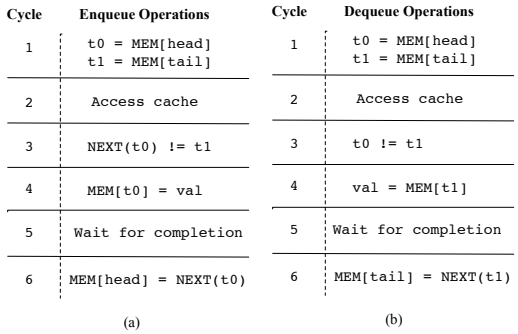


Figure 2. Enqueue (a) and dequeue (b) operations. Assumed cache hits with a dual-ported, 2-cycle access L1 cache.

Mnemonic	Description
benq{b,h,w,d} Rs, $QAddr$	Enqueue Rs into $QAddr$, block on queue full.
bdeq{b,h,w,d} Rd, $QAddr$	Dequeue from $QAddr$ to Rd,block on queue empty.
enq{b,h,w,d} Rt,Rs,$QAddr$	Enqueue Rs into $QAddr$, Rt returns status.
deq{b,h,w,d} Rt, Rd,$QAddr$	Dequeue from $QAddr$ to Rd, returns status in Rt.

Figure 3. Enqueue and dequeue instructions.

we perform the test to validate that the queue is not full. In this timeline, we assume it was not and proceed with cycle 4 to store the value in the queue. We will refer to this store as the *enqueueing-store* since it actually places the value in the queue. (Similarly, we refer to the operation in cycle 4 for dequeue as the *dequeueing-load*.) At this point, we have to wait for the store to complete before updating the head to the next position. Note, this store ordering is required for Lamport's algorithm to function properly and without races.

The enqueue and dequeue operations are shown in Figure 3. Since we target a MIPS architecture in our evaluation, these instructions are shown using MIPS-like mnemonics and implementation nuances. We support variants of these operations for each size of memory load or store supported in MIPS to ensure efficient code generation. Each enqueue instruction takes two input arguments: (1) the base address of the queue, and (2) the data to be enqueued. Dequeue operations only require the base address of the queue as input. Non-blocking instructions return an error code in their destination register.

3.3. Making Fine-Grained Queueing Efficient

So far, our design does not consider the interaction of queue operations with the cache coherence protocol or on-chip caches. It is critical to do so in order to achieve high throughput using our queue. Consider the implications of the proposed timeline in Figure 2 in a state-of-the-art CMP when used for fine-grained threading. First, if enqueueing and dequeueing operations are happening concurrently,

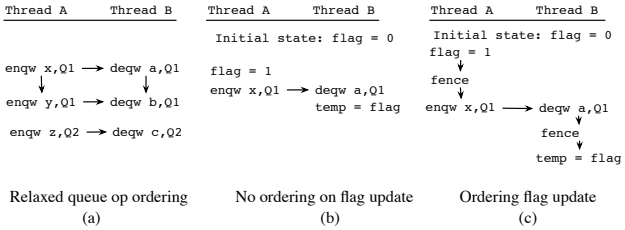


Figure 4. Relaxed memory ordering for queue operations.

then the head and tail variables for a queue are being read frequently by two different cores. This results in frequent transitions between exclusive state and shared state for the cache line holding each variable. This will significantly lengthen the time needed to decide if an enqueue or dequeue can proceed. Second, even in the case where the threads are not operating concurrently, the queue operations that can occur per unit time are determined by the amount of ILP that can be extracted from these instructions. The amount of parallelism is limited by the number of ports on the cache, the cache access time, and by the degree of ILP provided in the microarchitecture for enqueue and dequeue operations. Overall, these effects could serve to drastically reduce the throughput of our mechanism. To improve the efficiency of our mechanism, we adopt a few key measures: relaxed memory ordering for queue operations, reduced accesses to shared variables, and pipelined queue operations.

3.3.1. Relaxed Memory Ordering for Queue Operations

To enhance the parallelism available in the microarchitecture, we will assume a system with weak memory ordering, and we allow queueing instructions to be re-ordered dynamically as long as they operate on different queues. Since each queue operation actually involves multiple memory accesses, we are allowing more aggressive reordering of memory operations than would be available in a software implementation. This scenario is shown in Figure 4(a). The first enqueue and dequeue operations on Q1 must be properly ordered, otherwise FIFO ordering may not be preserved. This is indicated through the ordering arrows. However, the operations on Q2 can be re-ordered with respect to those on Q1, and this is shown by the lack of arrows to the enqueue or dequeue on Q2. For queueing operations on different queues that must occur in a given order, a memory fence is necessary to guarantee their order.

Furthermore, the relative order of other memory operations with respect to queueing operations are also relaxed. In Figure 4(b), a flag is set to 1 before the enqueue in Thread A, and the flag is read in Thread B after the corresponding dequeue. In this case, there is no guarantee that Thread B observes the value 1. However, this can be enforced using a fence before the enqueue in Thread A and after the dequeue in Thread B, as shown in Fig 4(c).

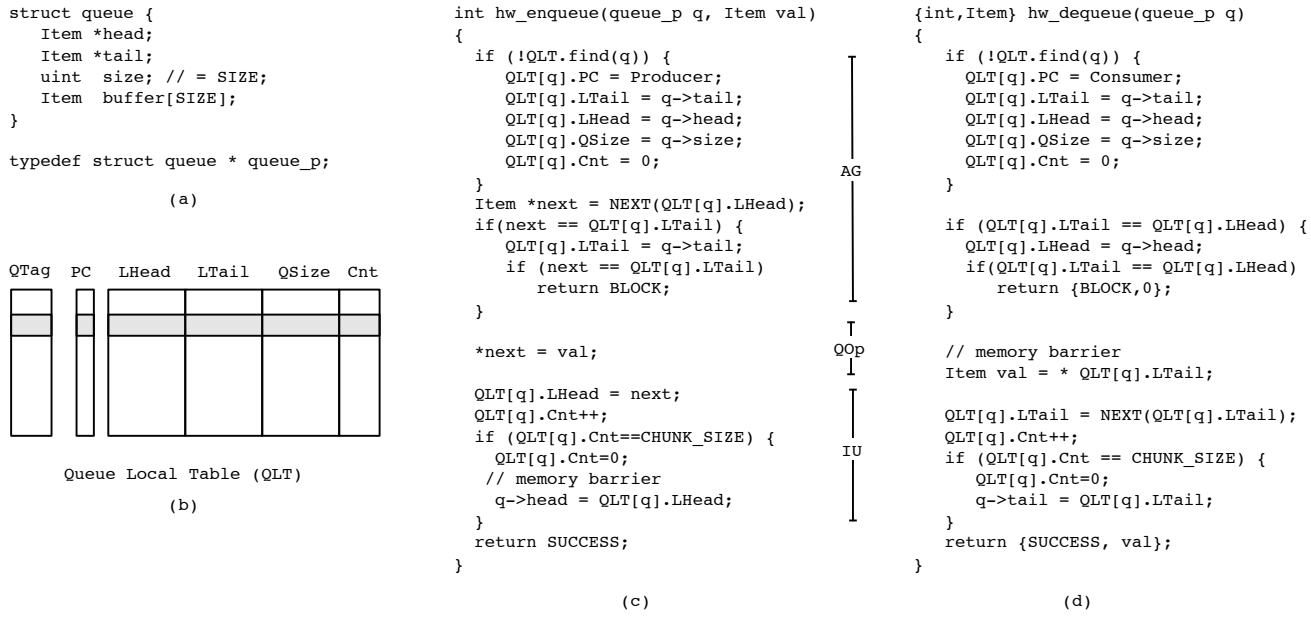


Figure 5. Algorithm for reducing accesses to shared variables.

If fences are needed between queueing operations, they must be inserted into the code. If a programmer is directly coding using HAQu, then she is responsible for inserting the fence. However, a parallelization framework or language may use these queues, and in that case, the compiler or a runtime library can provide automatic fence placement.

3.3.2. Reduced Accesses to Shared Variables

We modify our queueing algorithm so that we no longer need to access shared variables on each enqueue or dequeue operation. This simplification can eliminate the need to access the cache in advance of the *enqueueing-store* or *dequeueing-load*.

Our optimized enqueueing and dequeueing algorithms are shown in Figure 5 and are inspired by Lee *et al* [9] (abbreviated as P.Lee from here on). We adopt an innovation that avoids accesses to the shared head and tail through use of a local copy of the head and tail that is private to each thread. In our algorithm, these private variables are located in a hardware table, called the Queue Local Table (QLT), associated with each queue. (In P.Lee's approach, these values would be held in general purpose registers; we are simply moving them to dedicated registers.) The QLT and its fields are shown in Fig 5(b). Consider the case of enqueuing in Figure 5(c). Most of the time, the head of the queue is tracked using `QLT[q].LHead` (LHead for short), which is the head of the queue as stored in the QLT for queue, `q`. As long as LHead has not caught up with LTail, there is room to enqueue and the value of LHead is correct to use. Afterwards, the *enqueueing-store* is performed and LHead is updated.

We can also avoid the final update to the head of the queue. In this case, we can delay the update of head until enough items are enqueued so as to amortize the cost of coherence transactions on *head* and *tail*. The `CHUNK_SIZE` denotes the number of enqueues needed and this value is dependent on the block size of the cache and, ideally, is a multiple of the cache block size. We will discuss the interaction of this policy with the memory consistency model in Section 3.4. Also, we discuss a potential deadlock caused by this policy in Section 4.

The dequeueing operation is analogous to that of the enqueue. However, it is worth pointing out that the private copy of LHead and LTail are not shared by the producer and consumer. If a single core were both enqueuer and dequeuer, a separate table entry for enqueueing and dequeueing operations would be needed.

3.3.3. Pipelined queue operations

We decompose queue operations into three stages: *address generation*, *queueing-op*, and the *index-update*. Figure 5 shows the stages for the enqueue and dequeue operations between parts (b) and (c). In address generation (AG), the location of the *enqueueing-store* or *dequeueing-load* is calculated. With the availability of the *LHead* and *LTail*, generating multiple addresses in a single cycle is feasible. This enables high throughput issue of enqueue or dequeue operations. The *queueing-op* (QOp in the figure) performs the operation on the queue's buffer in memory. It is treated like a regular load or store for dequeueing and enqueueing, respectively. Consequently, this stage is executed by the processor as efficiently as it would a load or store operation and

under the same constraints. The QOp phase must wait until the proper stage in the pipeline to execute the load or store operation. Finally, the *index-update* stage (IU) occurs last to bring the state of the *head* or *tail* up to date. If multiple queue operations on the same queue are ready to retire in a single cycle, the IUs can be merged into a single update.

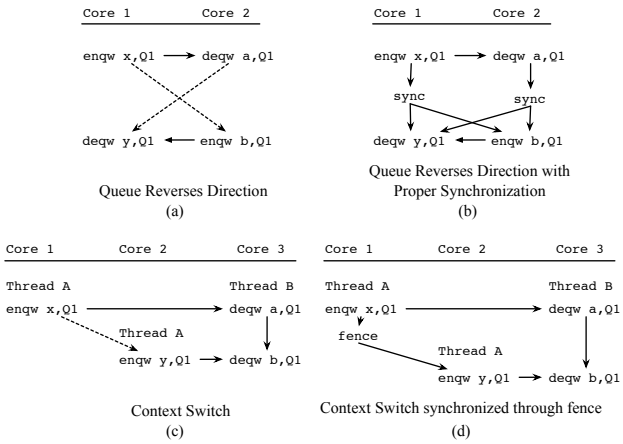


Figure 6. Important cases for providing OS transparency and programmability.

3.4. Ensuring Programmability and OS Transparency

To achieve high performance, we have significantly altered the behavior of the queue as compared to its software version. For example, we buffer the head and tail in hardware and allow them to be reused by multiple queue instructions. Since these values are now cached in hardware, we must be careful to preserve a clear programming model and one that remains OS transparent. The key questions we need to answer are: (1) when and for how long can we keep the local head and tail values buffered, and (2) under what circumstances is it allowable to delay the index-update operation.

To better explain the problems, Figure 6 shows a few scenarios that are particularly important for the functioning of our mechanism. Fig 6(a) shows the case where a queue is used to communicate from Core 1 to 2, and then after some time, it reverses direction. At any point in time, there is only one producer and consumer, but there is a transition point between these modes of operation. By the time the producer and consumer have switched roles, both must have the same view of the queue state and that should reflect all prior queue operations. But, if we are not careful, two cores may be holding simultaneously a QLT entry for either consuming or producing, which would violate our assumptions; and, worse, we have compromised the programmability of the mechanism. Finally, Fig 6(c) shows the case of a context switch. Initially, Thread A is running on Core 1, but, eventually, it is context switched to Core 2. Thread A should have a consistent view of the queue after moving to Core 2. If this case fails to work, we have sacrificed OS transparency. The common requirement among these cases is that all cores must retain a consistent view of the queue data structure.

To make the cases in Fig. 6(a) and (c) function properly, we must conservatively guarantee that all hardware state is flushed to memory before the producer or consumer switches cores. Because we cannot automatically detect these cases in hardware, we rely on a memory fence to force all information in the QLT to memory. On a weakly ordered system, we require that any pending index updates must be completed before a fence or synchronizing instruction can retire. Furthermore, no new queue operations can begin until the fence completes. Figure 6(b) and (d) show the proper orderings after synchronizing these events properly. Note, that in Fig. 6, an additional barrier operation (labeled as a sync) is needed to force ordering across threads, but this would also be required in an equivalent program using a software queue implementation; so, our requirements are no worse than the software queue case.

4. Implementation

4.1. Microarchitecture Description

To support HAQu, we add a Queue Functional Unit (QFU) to each core of a CMP, as shown in Figure 7. The QFU is made up of two structures. The Queue Local Table stores information about pending queue operations. In particular, it tracks the local head and tail variables described in Section 3.3.2. Also, it includes the Address Generation Buffer and the Queue Op Buffer (QOB) which contain a record of pending queue operations; we use them to order queue operations, determine the necessary updates to the head and tail variables, and recover from misspeculation. We now provide a brief overview of our micro-architectural mechanism. Since it involves straightforward changes to modern superscalar architectures, we will not discuss the logic in depth.

4.2. Pipelined Execution Description

Address Generation Stage. When a HAQu instruction is renamed, an entry is created for it in the Address Generation Buffer. The AGB contains all queue instructions (pInst in AGB) in program order. Instructions wait in this buffer until the source registers are available and have been read. Once the queue's base address has been resolved, the queueing operation can begin.

Address generation proceeds by checking the QLT for a valid entry for the specified queue. If no such entry exists, a new one must be created by loading the head, tail, and size of the queue. We provide the QFU direct access to the cache to obtain the needed values. Once the head and tail are known, a new entry in the QLT is initialized and it is

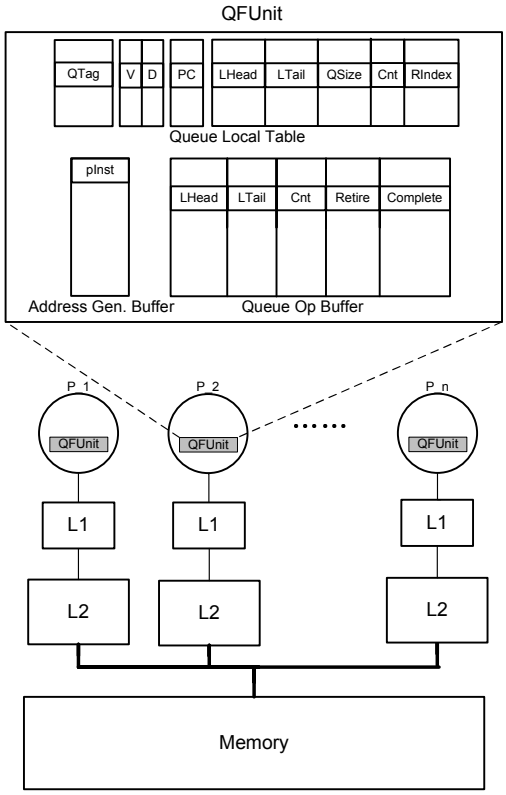


Figure 7. Hardware Accelerated Queue added to a CMP.

used to generate an address for the queue operation. The entry is set to valid (V bit in table).

QOp Stage. At this time, a store or load address can be generated for the enqueue or dequeue, respectively. We assume that we can generate multiple addresses in a single cycle. However, we need not generate more addresses than can be used to access the cache in a single cycle. Once the address is known, the information for the operation is moved to the QOB until it retires and completes. For an enqueue operation, the enqueueing-store cannot execute until it reaches the head of the re-order buffer. A dequeueing-load can execute as soon as its address is known.

Index Update Stage. When an instruction finally retires, we update several fields of the QLT. The RIndex field of the QLT is updated with the location in the queue it either read or wrote (LTail for enqueue or LHead or dequeue). This serves as a record of the last retired operation on the queue. The dirty (D) bit is set to indicate that it needs to eventually perform an index update. Also, the Cnt field is incremented; if Cnt is more than `CHUNK_SIZE`, the index update occurs immediately and Cnt is reset. Once the RIndex value is written to memory, the D bit is cleared.

Our policy to delay *head* and *tail* updates can introduce deadlock. If we wait until `CHUNK_SIZE` enqueues occur, the other thread waiting for a head update may be indefinitely stalled. Therefore, we add some additional logic that will force an update of a dirty entry after a fixed interval of time, t_{update}. When this timeout is triggered, we steal an idle cache-port and update the dirty entry. We use a separate timer for all entries in the QFU since it is a small table.

Memory Ordering Issues. Only a memory fence alters timing of queueing instructions relative to other memory operations. When a fence is decoded, a signal is sent to the QFU. The fence is then added to the AGB and tracked like other queue operations. The fence will remain in the AGB until all instructions in the QOB complete and all dirty entries in the QLT are flushed to memory and invalidated; and, queueing instructions that are added to the AGB after the fence must stall until the fence completes. Once the fence retires, it is removed from the AGB and queueing operations will begin again.

Note, we implement fence in this way to guarantee the ordering of operations discussed in Section 3.4. Indeed, our mechanism will suffer high overheads if fences occur frequently; however, in that case, the queues are unlikely to be the principal bottleneck due to limited re-ordering of memory operations.

Misspeculation and Precise Exceptions. Speculative execution and precise exceptions are properly handled. In the event of a branch misspeculation, the QLT must roll back to the last address assigned before misspeculation. After discarding speculative instructions from the QOB and AGB, the LHead and LTail in the QLT must be brought up to date with any pending non-retired instructions in the pipeline. This can be accomplished by walking the QOB and updating the QLT to the youngest values found there. In the event of an exception on any of the memory operations within the queueing instruction, we require the entire instruction be re-executed. In this way, all structures and resources associated with that instruction can be reclaimed precisely.

4.3. Managing QFU Resources

The QLT is a limited resource and can only support a limited number of entries. If the QLT is full with valid entries and a new queue is referenced, an action must be taken to reclaim resources within the QLT. Fortunately, this case is easy to handle. If there is a non-dirty entry, it can be immediately displaced and claimed for the new queue. If all entries are dirty, then an entry is picked at random to displace (though it is best to choose an entry with no pending operations). To displace a dirty line, we must write back the RIndex to the queue's *head* or *tail* depending on its status as either producer or consumer, respectively.

Because non-dirty lines can be evicted at any time, it is possible than an operation on a displaced queue entry is in the QOB but has not yet retired. In this case, the RIndex update simply writes directly to memory rather than updating the QLT.

4.4. Instruction Implementation Details

The queueing instructions are more complex than typical instructions in RISC processors today because they may need to access up to three memory locations: the head, tail, and buffer location. In the worst case, this requires access to three separate pages in memory and could generate three exceptions in the worst case. If the OS cannot guarantee multiple pages stay in memory, this could generate livelock. For OS compatibility and to reduce the potential complexity of these operations, we can require the head and tail to be allocated on the same page. Now, the instruction will access no more than two pages. This is no worse than the case of unaligned loads in current systems which access two pages.

5. Experimental Setup

To evaluate HAQu, we have integrated our design into SESC [12], an event-driven performance simulator, that accurately models CMP architectures. Table 1 shows the pertinent details of the simulated architecture. The core, L1, and L2 cache are common for all architectures we evaluated. The core is an aggressive 3-issue superscalar processor. We model a cache coherent CMP design with private L1 and L2 caches. Coherence is supported at the L2 cache over a ring interconnect. HAQu is supported with a 16 entry QLT per core, and t_{update} value of 500 cycles.

We support two different link bandwidths because our proposal is sensitive to interconnect design. The NarrowRing runs at 25.6GB/s per link and represents the capabilities of a currently available CMP; this is the default. We also model a futuristic interconnect, WideRing, with 102.4GB/s; we believe such a design could be on the market in the very near future.

For two studies, we will compare HAQu to an aggressive hardware queue. The hardware queue uses HAQu's instructions but maps them into a high speed buffer connecting two cores. We assume 1 cycle to read/write from/to the queue. Also, we used 4 KB for each hardware buffer and allow the same number that HAQu can use (which is unlimited in theory but under 16 our studies).

Tables 2 and 3 show the different application binaries we study. (They will be described in more detail later.) For all of our application binaries, we used GCC 4.5 targeted to the Mips32 architecture. To accurately compare the performance of different binaries, simply timing a fixed number of instructions cannot be used. Instead, "simulation markers" are inserted in the code of each binary, and simulations are run for a given number of markers. After skipping the initialization, a certain number of markers are executed so that the binary graduates between 100 and 500 million instructions.

Table 1. Architecture simulated. All cycle counts are in processor cycles.

No. of Cores: 2 to 16 Frequency 3.2 GHz Fetch width 6 Issue width 3 Retire width 3 Weak Ordering Consistency Model Branch predictor: Mispred. Penalty 14 cycles BTB 2K, 2-way	ROB 126 I-window 68 LD/ST queue 48/42 Mem/Int/Fp unit 1/2/1 Hybrid GShare & Local
Private L1 Cache: Size 64KB, assoc 4 Line 64B Latency 2 cycles	Private L2 Cache: Size 2MB, assoc 8 Line 64B Latency 9 cycles Protocol MESI Memory: Latency 300 cycles Bandwidth 25.6GB/s
Ring interconnection: NarrowRing Link Bwd (8 data bytes/cycle): 25.6GB/s WideRing Link Bwd (32 data bytes/cycle): 102.4GB/s	
QLT Size: 16 entire t_{update} = 500 cycles	

Table 2. Mudflap configurations.

Name	Description
Base	No Mudflap,-O2
Mflap	Base + Mudflap
HAQu	Parallelization with HAQu
HWQ	Parallelization with Hardware Queue
SWQ	Parallelization with Software Queue

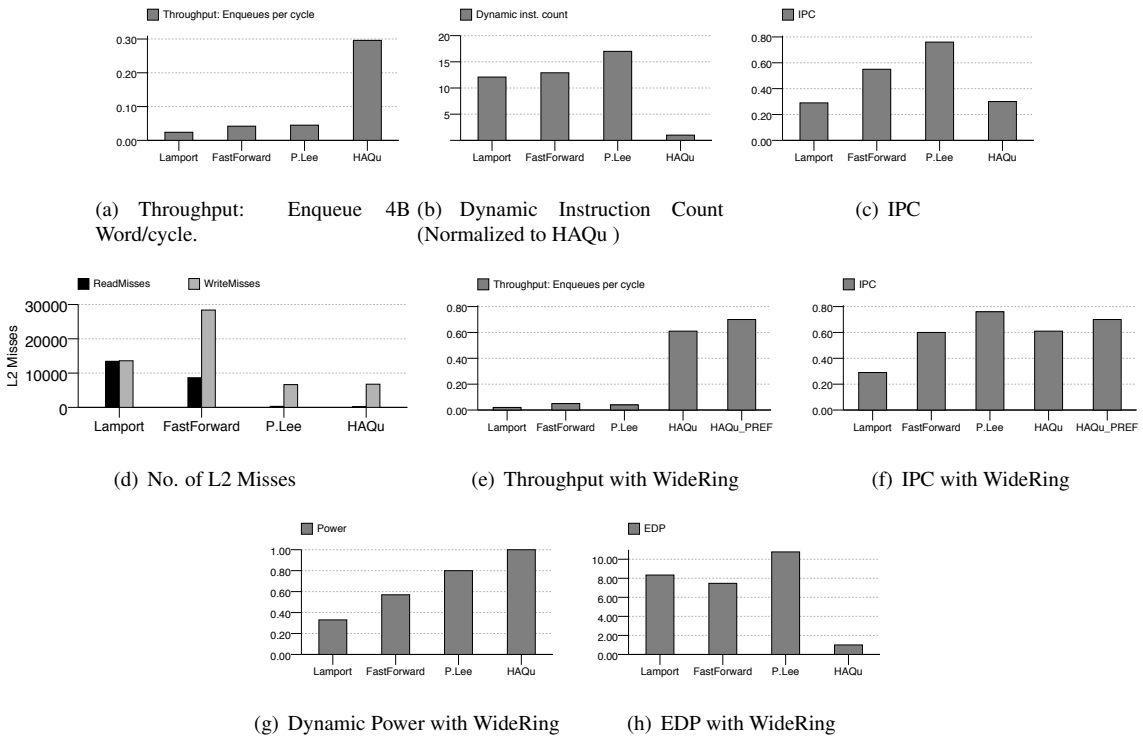
Table 3. DSWP configurations with varied queues.

Name	Description
Base	No DSWP,-O3
HAQu	Base + DSWP with HAQu
HWQ	Base + DSWP with Hardware Queue
SWQ	Base + DSWP with Software Queue

6. Case Studies

6.1. Micro Benchmarks

We wrote three micro benchmarks to evaluate the efficiency of HAQu compared to software queue implementations. The three algorithms selected are Lamport's [8], FastForward [2], and P.Lee's [9]. For each one, we kept the queueing data structure as similar as possible and minimized false sharing by placing the head, tail, and data buffer in different cache lines. The key difference in these implementations are the degree of false sharing they allow and how they check for empty and full status. Lamport's queue suffers much more false sharing than FastForward or P.Lee's because head and tail are referenced on each enqueue or dequeue operation. P.Lee's queue is most similar to the HAQu implementation. FastForward differs from the others because it uses the value stored at the next entry to decide if the queue is empty/full. For example, if the next entry is non-zero, it can be dequeued, otherwise it must



(a) Throughput: Enqueue 4B Word/cycle.

(b) Dynamic Instruction Count (Normalized to HAQu)

(c) IPC

(d) No. of L2 Misses

(e) Throughput with WideRing

(f) IPC with WideRing

(g) Dynamic Power with WideRing

(h) EDP with WideRing

Figure 8. Micro benchmarks using queues.

wait. This eliminates contention for a shared head and tail.

In each run, enqueue and dequeue are executed 500 times in a loop with 200 iterations, and we used 4-byte elements for each queue operation. In total, we transport 400,000 bytes. Each enqueue or dequeue is inlined. For the software queues, memory fences are inserted for a weak ordered consistency model. For all the queues, we use queue buffers holding 1024 elements (4KBytes). For each micro benchmark, simulation begins after the producer and consumer reach a barrier at the beginning of the loop.

Figure 8(a) shows the throughput for each queue in terms of enqueues per cycle. Overall, HAQu achieves 12.5× speedup over Lamport's queue, 7.0× speedup over FastForward, and 6.5× speedup over P. Lee's. This is possible because it generates an enqueue operation with a single instruction while the other implementations require several, usually dependent, instructions. Figure 8(b) shows the dynamic instruction count for each queue; the other queues use 12× to 17× more dynamic instructions. Furthermore, there is little ILP in any of the queues, on average. Figure 8(d) shows the L2 miss rate which is mostly coherence misses for these runs, and Figure 8(c) shows the IPC. P.Lee has the highest IPC due to few coherence misses, but even it does not reach one instruction per cycle in our simulations. This shows that an aggressive superscalar cannot extract much queue-level parallelism from the optimized software queues, but it can with HAQu.

Fig. 8(d) shows some interesting behaviors. Both P.Lee and HAQu have very few read misses, and this is a direct result of caching a local copy of the head and tail and only updating it when necessary. Figure 8(d) also shows that HAQu avoids cache thrashing by updating the head and tail only after `CHUNK_SIZE` updates. Its behavior is as good as P.Lee's which it is modeled after. However, on the NarrowRing HAQu is not living up to its full potential. Figure 8(e) and 8(f) show throughput and IPC for the WideRing interconnect. With additional bandwidth, the speedup of HAQu soars to 25.5× that of Lamport's queue and 13.0× over FastForward. In addition, the IPC is nearly as good as FastForward. Furthermore, if we add prefetching support to HAQu, we take advantage of even more bandwidth and boost the speedup over Lamport's queue to 29×. HAQu is an interesting design on current models (NarrowRing), but it is even more valuable as interconnects scale to support higher bandwidth links.

Figure 8(g) and Figure 8(h) show the normalized dynamic power and the normalized energy delay product (EDP) respectively. Figure 8(g) shows that HAQu consumes a bit more dynamic power compared to the other queues. This is because HAQu is able to do much more work in a given unit of time. Due to its fast execution, HAQu has a significantly better EDP. In a power constrained system, we could match the power of a software queue by running the entire core a little slower but still provide a performance gain.

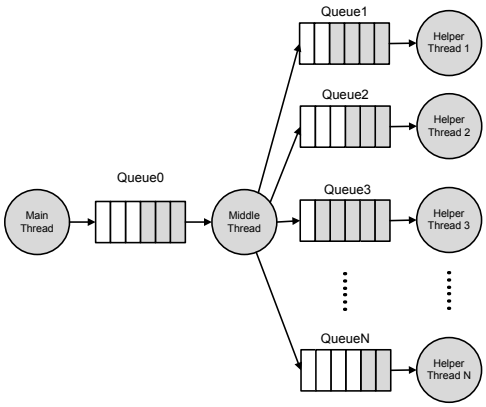


Figure 9. Parallelizing Mudflap using Hardware Accelerated Queue.

6.2. Pointer-use Checking Acceleration

Mudflap [1] is a pointer-use checking utility integrated into GCC [14]. When a program is compiled with Mudflap, error checking code is inlined at all memory references to validate the memory access against a database of allowed memory regions in the stack, heap, static variable section, and others. These checking codes add considerable overhead to any program that uses them [5]. It would be nice if, instead of adding these codes to the critical path of the program, they could be performed in a helper thread. Using HAQu, we offload these fine-grained checks to another thread. We adopt a pipelined architecture for parallelizing Mudflap as shown in Figure 9. Because address-check operations occur frequently, we want to keep the cost of enqueueing as low as possible. Therefore, we enqueue into a single queue which must be emptied quickly by the Middle Thread. The Middle Thread removes the requested operation from Queue0 and then assigns it to one of the Helper Threads for completion. We implemented parallelized versions with a software queue using P.Lee's algorithm, a hardware queue, and HAQu as shown in Table 2.

In this section, we evaluate HAQu's performance with several SPECint applications running the parallelized Mudflap algorithm on a 16-core CMP. We used P. Lee's queue for the software queue. Figure 10(a) shows the speedup for each application over Mflap running on the NarrowRing design. Overall, SWQ shows the worst performance, while the HWQ achieves a 1.3× speedup over HAQu. HAQu uses many fewer processor cycles per enqueue, enabling it to use those cycles for the main application.

HAQu uses 24× fewer dynamic instructions and 2.6× fewer L2 misses than SWQ. The performance difference between HWQ and HAQu comes primarily from the lower bandwidth of the NarrowRing compared to the high throughput hardware queue. However, given more bandwidth, HAQu is very competitive with HWQ. Figure 10(b) shows that performance normalized to Mflap on the WideRing. Here, HAQu can achieve nearly equivalent speeds as HWQ. Furthermore, the overall speedup compared to Mflap jumps up to 2.2×.

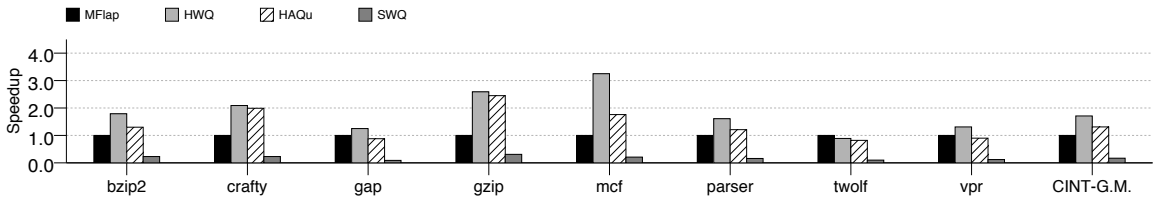
On WideRing, our parallelized Mudflap is only 1.85× slower than serial execution without Mudflap. We would have liked to reduce this more, but most of the remaining performance gap comes from bottlenecks unrelated to queueing. A key problem is load imbalance among the helper theads. Due to the load distribution policy, one helper thread may be busy for a long time, and its queue may fill up and prevent the Middle Thread from making progress. We studied several load balancing algorithms and found that round-robin works the best, on average. All of our results are based on this policy. However, some applications perform better with dynamic load balancing algorithms.

6.3. Decoupled Software Pipelining

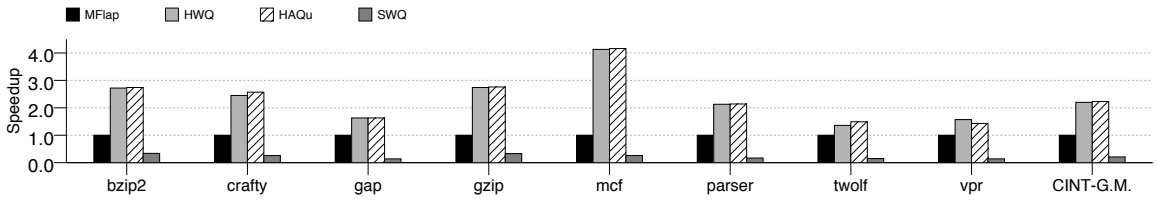
Decoupled Software Pipelining (DSWP) [11] is an automatic parallelization technique that partitions a program, usually just a loop nest, into a pipeline. Queues are inserted between the pipe stages to serve as communication channels for live-ins and live-outs of each stage. To achieve good performance with this parallelization approach, the cost of communication must be small with respect to the work performed in each pipe stage. If fine-grained queueing can be made efficient and common, many loops with fine-grained parallelism could be parallelized using this technique.

We have implemented a fully-automatic, profile-directed DSWP pass in GCC to evaluate the potential of HAQu in a challenging fine-grained threading environment. We have not incorporated any speculation techniques into our implementation; pipe stages must be fully independent: no true, anti, or output dependences can cross stages. We select any loop for parallelization that appears to offer a speedup based on profiling data, and we generate a parallel version with two threads. For all communication between pipestages, we assign a dedicated queue. After the parallelized region, the main stage must wait for the helper stage to finish. Table 3 shows the experimental configurations for DSWP.

Figure 11(a) shows speedups of each queue normalized to Base. In the figure, CINT-G.M. means SPECint geometric mean. Overall, HWQ achieves 10% speedup over Base. This speedup comes from mcf which achieves a 57% speedup over Base, and vpr which achieves 25% over base; bzip2 attains a modest speedup around 4%. The other applications except for gap do contain at least one parallelized loop that turns out to be unprofitable, which explains their degraded performance. HAQu is much better than the SWQ, but it achieves an overall slowdown of 10% on this configuration. Compared to the HWQ implementation, HAQu pays a much bigger latency to communicate between cores. This turns out to be very costly in DSWP because the main stage must wait for the helper stage to finish; this places queue latency on the critical path.



(a) On NarrowRing.



(b) On WideRing.

Figure 10. Speedup of Parallelized Mudflap.

In order to distinguish the effects of latency and bandwidth, we also evaluate DSWP on the more aggressive WideRing interconnect. In this system, HAQu is able to leverage the higher bandwidth and break even with serial. mcf attains a 45% speedup, vpr reaches 5%, and bzip2 attains 1%. Furthermore, gzip, twolf, and parser are penalized less for their attempted parallelization than on the NarrowRing. Clearly, DSWP is sensitive to latency and bandwidth, and this poses some challenges for HAQu. However, considering the flipside, without a HAQu-like-mechanism, achieving good performance using a DSWP algorithm is very difficult. With HAQu's modest architectural requirements, auto-parallelization techniques for challenging integer applications might be attainable on future systems.

7. Related Work

Software Queueing In order to support concurrent lock free queueing, researchers have proposed a variety of queues [10, 8, 2, 3, 9]. They have primarily focused on providing lock-free queueing algorithms which are cache conscious. More discussion of these queues is included in Section 6.1. Other work in software queueing has focused on multiple-producer, multiple consumer queues. Currently, HAQu does not support such queues, but it is a challenging problem and a focus of our future work.

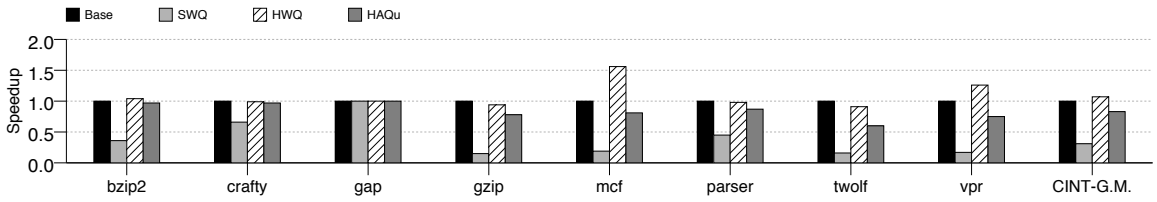
Some software-only alternatives do exist for reducing overheads associated with fine-grained queueing; Tiwari *et al.* [15] used speculation to hide the latency and bulk enqueues to reduce the frequency of queue operations.

Queueing Support in the ISA The VAX [4] supported hardware accelerated queues similar in concept to HAQu. However, their implementation was considerably more complex than HAQu. The VAX instructions INSQHI, INSQTI, REMQHI, and REMQTI allowed insertion or removal at the head or tail of a self-relative doubly-linked list. These instructions were interlocked and could not be interrupted; this prevented other processors from updating the queue at the same time. These complexities made it very difficult to achieve high performance and programmers often resorted to implementing their own software queues. Hence, the VAX design failed to meet *Criterion I* and *Criterion II*. HAQu intentionally avoids these pitfalls by adopting a hardware design that mimics successful software queueing algorithms. Furthermore, HAQu's design builds on top of the microarchitectural features that enable precise exceptions and branch misspeculation, thereby ensuring ease of integration in aggressive superscalar designs.

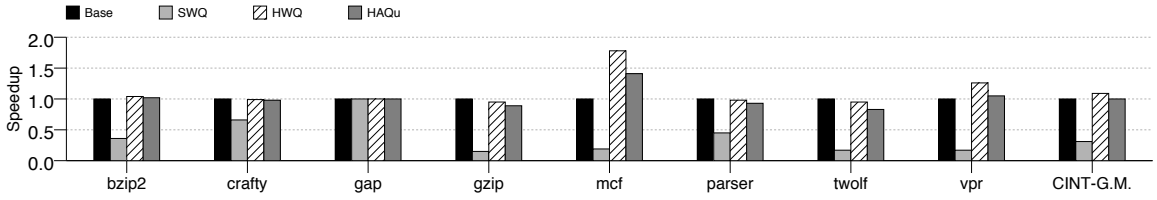
Hardware Queueing Hardware queues [13, 11, 6] have been proposed to support low latency between a producer and consumer. Heapmon [13] uses a hardware queue to offload the monitoring of heap memory bugs to a helper core. DSWP [11] leverages a high speed Synchronization Array to support fine grained parallelization. Carbon [6] uses a hardware queue to accelerate dynamic task scheduling. While these works meet *Criterion I* and *Criterion II* they do not consider *Criterion III* and *Criterion IV*(Section 2).

8. Summary

Queues are a commonly used programming construct in multithreaded programs for synchronization and communication. Queues are effective when the granularity of parallelism is large enough to amortize the cost of enqueueing and dequeueing. But, software queues tend to be too expensive to support fine-grained parallelism. To solve these



(a) On NarrowRing.



(b) On WideRing.

Figure 11. Speedup of DSWP over Base using 2 cores.

problems, we propose a hardware accelerated queue, or HAQu. HAQu adds hardware to a CMP that accelerates operations on queues in memory. Rather than adding a custom hardware queue, our design implements fast queueing through an application's address space with operations that are compatible with a fully software queue. Our design provides accelerated and OS-transparent performance.

We evaluated HAQu on queueing micro benchmarks, the parallelization of Mudflap, and Decoupled Software Pipelining. Compared to FastForward [2] and Lee *et al* [9], HAQu achieved speedups of 7× and 6.5×. We found that HAQu can deliver high throughput and low latency queueing, and that it carried over to application level studies. The low instruction footprint of each queue operation frees the processor to perform other tasks, thereby enabling fine-grained parallelism. Furthermore, HAQu is able to scale-up and attain high throughput on next generation interconnects. In conclusion, HAQu is a modest architectural extension that can help parallel programs better utilize the interconnect and processing capability of future CMPs.

References

[1] F. Eigler. Mudflap: Pointer use checking for C/C++. In *Proc. of the GCC Developers Summit*, 2003.

[2] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[3] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009.

[4] Vax macro and instruction set reference manual. URL, 2009. http://h71000.www7.hp.com/doc/73final/4515/4515pro_021.html.

[5] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, 2009.

[6] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[7] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*

[8] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[9] P. P. Lee, T. Bu, and G. P. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *Parallel and Distributed Processing (IPDPS), 2010 IEEE International Symposiumon*, 2010.

[10] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 1998.

[11] G. Ottoni, R. Rangan, A. Stoler, and D. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, November 2005.

[12] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005.

[13] Rithin Shetty, Mazen Kharbutli, Yan Solihin, and Milos Prvulovic. HeapMon: a Low Overhead, Automatic, and Programmable Memory Bug Detector. *Proc. of IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (P=ac2)*, Oct 2004.

[14] T. G. Team. GNU Compiler Collection. URL: http://gcc.gnu.org. 2008.

[15] D. Tiwari, S. Lee, J. Tuck, and Y. Solihin. Mmt: Exploiting fine-grained parallelism in dynamic memory management. In *IPDPS*, 2010.