# Amortizing Software Queue Overhead for Pipelined Inter-Thread Communication

Ram Rangan David I. August
Department of Computer Science
Princeton University
{ram, august}@cs.princeton.edu

## ABSTRACT

Future chip multiprocessors are expected to contain multiple on-die processing cores. Increased memory system contention and wire delays will result in high inter-core latencies in these processors. Thus, parallelizing applications to efficiently execute on multiple contexts is key to achieving continued performance improvements. Recently proposed *pipelined multithreading (PMT)* techniques have shown significant promise for both manual and automatic parallelization. They tolerate increasing inter-thread communication delays by enforcing acyclic dependences amongst communicating threads and pipelining communication.

However, lack of efficient communication support for such programs hinders related language and compiler research. While researchers have proposed dedicated interconnects and storage for inter-core communication, such mechanisms are not cost-effective, consume extra power, demand chip redesign effort, and necessitate complex operating system modifications. Software impelementations of shared memory queues avoid these problems. But, they tend to have heavy overhead per communication operation, causing them to negate parallelization benefits and worse still, to perform slower than the original single-threaded codes. In this paper, we present a simple compiler analysis to coalesce synchronization and queue pointer updates for select communication operations, to minimize the *intra-thread* overhead of software queue implementations. A preliminary comparison of static schedule heights shows a considerable performance improvement over existing software queue implementations.

## 1. INTRODUCTION

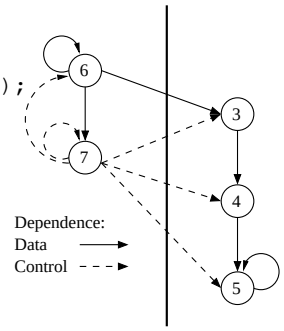
Parallelizing individual tasks into multiple threads is key to performance improvement on chip multiprocessors. High inter-core communication delays have made the notion of thread extraction almost synonymous with the search for long-running threads with minimal communication. While this strategy has had some success for scientific applications, it has impaired similar efforts for general-purpose applications (both manual *and* automatic). Recently however, language and compiler *pipelined multithreading (PMT)* techniques (StreamIt [9, 3], Decoupled Software Pipelining (DSWP) [5, 4], and others [1, 2]) have shown promise as viable methods to expose thread-level parallelism. They can handle more codes because they embrace inter-thread dependencies (albeit acyclic dependencies) by partitioning applications into concurrent, long-running producer and consumer threads. They place fewer demands on interconnect latency because they easily tolerate long-latency inter-thread communication by pipelining acyclic communication (also known as streaming communication). For example, consider the loop in Figure 1a and its low-level code in Figure 1b. A PMT partitioning of this loop is shown in Figure 1c. Notice that all inter-thread dependencies flow from thread 1 to thread 2. Inter-thread queues can be used to take advantage of the acyclic flow to buffer values and provide decoupled pipelined communication.

```
1  int sum = 0;
2  do {
3  sum += ptr->val;
4  } while(ptr=ptr->next);
```

(a) Pointer chasing loop

```
1  r4 = 0
2  loop:
3  r2 = r1 + 4
4  r3 = M[r2]
5  r4 = r3 + r4
6  r1 = M[r1]
7  br r1 != 0, loop
```

(b) Assembly code




(c) Dependence graph

**Figure 1: PMT example**

While PMT techniques show promise, current architectures are without sufficient architectural and operating system support for streaming communication. Although researchers have proposed dedicated microarchitectural structures for inter-core communication (synchronization array [5], FIFOs [6], scalar operand networks (SONs) [8]), their adoption in commercial processors has been hindered for several reasons. Dedicated structures, such as the ones mentioned above, result in sub-optimal use of hardware, since they are used exclusively for inter-thread operand transfers. Memory traffic, for instance, cannot be multiplexed on these dedicated interconnects. Besides resulting in sub-optimal use of hardware, such dedicated structures (interconnect and storage) consume extra power, demand chip redesign effort, and often necessitate complex operating system modifications.

Software queues avoid these problems. The default memory consistency and cache coherence implementation of any machine provide the complete hardware support needed for such communication. Memory based synchronization and communication obviates the need for any OS modifications. Thus, shared memory software queues may seem like an attractive communication alternative.

The code sequence to implement either a produce or a consume communication operation with shared memory software queues in-

```
void produce(int value) {
  // spin while queue is full
  while(occupancy == q_size);
  // occupancy < q_size
  q[tail].data = value;
  occupancy++;
  tail = (tail+1)%q_size;
}
```

```
int consume() {
  // spin while queue is empty
  while(occupancy == 0);
  // occupancy > 0
  value = q[head].data;
  occupancy--;
  head = (head+1)%q_size;
  return value;
}
```

**Figure 2: Occupancy counter based software queues.**

```
void produce(int value) {
  // spin until tail empty
  while(q[tail].full);
  // q[tail].full == 0
  q[tail].data = value;
  q[tail].full = 1;
  tail = (tail+1)%q_size;
}
```

```
int consume() {
  // spin until head full
  while(!q[head].full);
  // q[head].full == 1
  value = q[head].data;
  q[head].full = 0;
  head = (head+1)%q_size;
  return value;
}
```

**Figure 3: Condition variable based software queues.**

volves three fundamental steps - synchronization to ascertain whether a producing store or a consuming load can execute, the store or load itself to effect the data transfer and queue pointer update. Of these, synchronization and queue pointer update instructions are overhead instructions. Even though PMT codes tolerate inter-core latencies very well, they are highly sensitive to the intra-thread overhead of communication. Even highly tuned code sequences have significant recurring intra-thread overhead and tend to negate any benefits from PMT parallelization. While synchronization and queue pointer update are necessary evils, we observe that they can be coalesced into one per group of queue accesses instead of one per individual queue access and hence, the overhead can be effectively amortized over multiple queue accesses.

In this paper, we present a compiler analysis to automatically identify queue operations for which synchronization and queue pointer update can be coalesced. The analysis is implemented in an automatic DSWP[4] compiler. We find that, by amortizing synchronization and queue pointer overhead over multiple queue accesses, we can improve profile-weighted static schedule height of parallelized program sections by as much as 241% (and 85% on the average) over existing software queue implementations, across a range of benchmarks. DSWP implementations can either use a unique queue to handle each inter-thread dependence or may choose to merge two or more dependencies into a single queue. Each has its pros and cons. While the former leads to queue addressability concerns, the latter constrains the scheduler by requiring the order of queue accesses be strictly the same in all communicating threads. We shall assume a queue per dependence implementation in this paper. Further, we shall assume finite-sized queues with equal number of entries in each queue.

## 2. SOFTWARE QUEUES

In this section, we shall review software queue implementations and understand the overhead arising from a naïve use of such implementations to support communication in DSWP'ed codes. To keep the discussions streamlined, we shall focus only on single-producer single-consumer scenarios. However, the mechanisms discussed in this paper can easily be extended to other communication patterns as well.

Code for queues based on simple occupancy counters for a single-producer, single-consumer queue is shown in Figure 2. The queue is composed of a head index, a tail index, an occupancy counter and a shared memory array of data items. The tail (head) index is updated exclusively by the producer (consumer). The occupancy counter is updated by both. To access the queue, the producer (consumer) spins until the occupancy counter is less than the queue size (greater than 0) indicating the queue is not full (not empty). Once past the spin loop, the producer (consumer) can write (read) the data to (from) the queue and increment (decrement) the occupancy counter. Once the data item is written (read), the tail (head) index should be updated to point to the new tail (head) slot. Since only a single thread will produce data into each queue and only a single thread will consume data from each queue, the head and tail pointers can be stored locally on the consumer and producer cores respectively. Additionally, no mutexes are required to protect the queue (although, the appropriate memory fence instructions are required to enforce the correct ordering of operations). Mutexes are required to protect accesses to the occupancy counters. Since both threads read and write the occupancy counter variable, such an approach will work well only on an SMT core where both threads can share the L1 cache. When executing on multiple cores, accesses to the occupancy counter will lead to cache line ping-ponging between private caches resulting in poor performance.

We can solve the problem of cache line ping-ponging by introducing fine-grained condition variables. Code for such an implementation is shown in Figure 3. Here, the queue is composed of a head index, a tail index, and a shared memory array of condition variable, data item pairs. The tail (head) index is updated exclusively by the producer (consumer). To access the queue, the producer (consumer) spins until the condition variable for the tail (head) queue slot indicates the slot is empty (full). Once the queue slot becomes available, the producer (consumer) can write (read) the data to (from) the queue and signal the condition variable that the slot is now full (empty). Once the data item is written (read), the tail (head) index should be updated to point to the new tail (head) slot. Such fine-grained condition variables allow for an efficient implementation of software queues [7].

Variants of the above approaches often use multiple queue buffers or coarser-grained signaling to minimize cache line ping-ponging or amount of storage used for synchronization or both. Regardless, the main drawback of software queues is that the code sequences to produce and consume a single datum are quite lengthy. The C code shown in Figures 2 and 3 will likely expand into many instructions. For simplicity, we shall assume that access to each and every queue slot has to be explicitly synchronized. A naïve use of any of the above software queue implementations to handle communication in DSWP by reproducing the entire code sequence (comprising synchronization, data transfer and queue pointer update instructions) for every single communication operation to each queue will naturally lead to performance inefficiencies. However, in code regions with two or more accesses to "parallel" queues, the overhead arising from synchronization and queue pointer update instructions can be amortized across these multiple accesses. Coalescing synchronization leads to an increase in critical section size, which may be unacceptable in most conventional scenarios. However, since DSWP pipelines communication and synchronization, increasing the critical section size will at worst manifest itself as an increase in pipeline fill cost. The next section expands upon this intuition and presents an analysis to automatically identify parallel queue accesses for which synchronizations and queue pointer updates can be coalesced.

## 3. ANALYSIS

In this section, we shall start with an intuitive algorithm to determine which queue accesses to coalesce synchronization for and refine the algorithm progressively to take into account various cor-

rectness constraints. We shall use the notation `ACQUIRE n` and `RELEASE n` to denote acquire and release of a condition variable $n$. The term *synchronization number* ('syncno' for short) will be used to abstractly refer to a condition variable. The analysis operates on a machine-independent intermediate representation (IR). While the exact code sequence for an `ACQUIRE n` or `RELEASE n` operation may vary slightly depending on whether it synchronizes a produce or a consume operation, for the discussion below, it suffices to know that `ACQUIRE n` is a spinlock loop which will prevent the enclosing thread from making forward progress until the spinlock succeeds. Since coalescing synchronization is inherently more complex than coalescing queue pointer updates, the analysis is driven from a synchronization standpoint. Section 4 will explain how queue pointer update coalescing can be piggybacked on synchronization coalescing during code-generation.

We define *synchronization equivalence group (SEG)* as a group of queues for which synchronization can be coalesced. To start with, let us consider only innermost loops in 2-thread DSWP. For pipelined communication between two innermost loops, we intuitively see that by acquiring and releasing synchronization at the beginning and end of the loop body respectively, we can correctly synchronize all communication operations in the loop body. However, when we consider a two-deep loop nest that has been DSWP'ed such that there are communication operations in both the outer and inner loops, our simple strategy will no longer work. All synchronization cannot be coalesced at the outer loop boundaries as that will leave communication operations in the inner loop without any synchronization. In Figure 4, notice that there is no `ACQUIRE n` or `RELEASE n` operation for queue accesses in the inner loop. This can cause produce operations to run over the allotted buffer space or consume operations to prematurely read stale data. Thus, upon coalescing, `ACQUIRE` and `RELEASE` operations for a queue access can move to a less restrictive control flow condition (like outside an "if" statement), but cannot be hoisted out of their original loops. This condition ensures that every dynamic queue access operation is guarded by at least one `ACQUIRE n` and one `RELEASE n` operation.

But this condition is *not* sufficient. For example, in Figure 5, synchronization is acquired and released at the beginning and end of each loop nest level. Even though this satisfies the condition stated above, it fails to provide correct synchronization because, assuming a queue to contain 32 entries, the producer thread's `ACQUIRE 1` will spinlock trying to produce beyond 32 queue items. However, the consumer thread will spinlock in `ACQUIRE 0` in its outer loop, since the producer's outer loop will never get a chance to execute its synchronization release, `RELEASE 0`. Since the consumer is spinlocking in `ACQUIRE 0`, it will never reach its inner loop, thus leading to a deadlock.

Figure 6 highlights another potential pitfall if `ACQUIRE` and `RELEASE` operations for a syncno are not control-equivalent. In this example, if the loop were to proceed down the "If" path, the `RELEASE` operation for syncno 0 would never execute. This in turn would cause the consumer thread to spin loop in its `ACQUIRE` operation and prevent it from making any progress.

To summarize, the necessary and sufficient conditions for correct synchronization coalescing are:

1. For each syncno $n$, dynamically, there be a many-to-one or one-to-one mapping of synchronization operations (`ACQUIRE`s and `RELEASE`s) to queue accesses, for each queue in its $SEG(n)$[1].
2. There be no circular inter-thread dependence among overlapping critical sections in any thread.
3. For each syncno, dynamically, there be a one-to-one correspondence between `ACQUIRE`s and `RELEASE`s.

To satisfy all the above conditions, we shall define a single-entry single-exit acyclic region with control equivalent entry and exit points called a *loop region*. `ACQUIRE` and `RELEASE` operations for all queue accesses in this region can be coalesced at the region entry and exit points. The *acyclic* clause satisfies conditions 1 and 2 and the *control equivalence* clause satisfies condition 3.

The first step of the analysis is to form loop regions. Initial loop region entries are defined as points in a loop's static CFG where control flow is transferred *into* the loop, including from inner loops and fall-through from loopback branches. Similarly, initial loop region exits are points in a loop's static CFG where control flow is transferred *out of* the loop, including to inner loops. The source of a loop backedge is a special loop region exit and likewise a loop header is a special loop region entry.

The algorithm starts by marking initial loop region entries and exits as defined above as entry and exit nodes respectively in the given CFG. Then, for every node in the CFG it computes its latest post-dominator and earliest dominator, taking into account the new entry and exit nodes. All latest post-dominators are marked as exit nodes and all earliest dominators are marked as entry nodes. Source nodes of in-edges into earliest dominators are marked as exit nodes and destination nodes of out-edges from latest post-dominators are marked as entry nodes. The algorithm iterates till no new entry and exits nodes are marked on the CFG. This procedure gives us maximal loop regions. The set of queues accessed in each loop region forms an $SEG$. The analysis also remembers the directionality (i.e. produce or consume) of each $SEG$. If both produce and consume queue accesses occur in a particular loop region, the analysis groups them into two different $SEG$s.

Now, all queue operations in a given loop region can theoretically have their `ACQUIRE` and `RELEASE` operations coalesced at the region's entry and exit nodes respectively. However, due to the inter-thread nature of synchronization, it is important to make sure corresponding queue accesses in other threads all fall into the same loop region in those threads. The second step of the analysis compares $SEG$s across all communicating threads and ensures that for each queue, the $SEG$ it is a member of is exactly the same across all threads accessing that queue. If this is not the case, the corresponding $SEG$s are split until the condition is true. Once this is done, each globally unique $SEG$ is assigned a globally unique syncno.

For example, let us consider a three-way partitioning of a loop (with no inner loops) wherein thread 1 produces into queues 1, 2, 3 and 4, thread 2 consumes from queues 3 and 4 and produces to queues 5 and 6 and thread 3 consumes from queues 1, 2, 5 and 6. The first step of the analysis creates $SEG$ [1,2,3,4] for thread 1, $SEG$s [3,4] and [5,6] for thread 2 and $SEG$ [1,2,5,6] for thread 3. The second step of the analysis will split the $SEG$s such that thread 1 has [1,2] and [3,4], thread 2 has [3,4] and [5,6] and thread 3 has [1,2] and [5,6]. Now, there are three globally unique $SEG$s - [1,2], [3,4] and [5,6] and they are assigned syncnos 0, 1 and 2 respectively.

The first step of the analysis is a local analysis. The second step has to make a pass over all threads to determine the correct mapping from each syncno to its $SEG$ and is a global analysis. For

[1]Note, to take into account coarse-grained signaling implementations, the condition can be modified slightly to ensure a many-to-one or one-to-one mapping with every $k$th queue access, where $k$ is the signaling granularity.

```
        Producer Thread                 Consumer Thread

Outer:  ACQUIRE  0              Outer:  ACQUIRE 0
        produce [4] = r5                consume r5 = [4]
Inner:  produce [7] = r6        Inner:  consume r6 = [7]
        r6 = r6 + 1                     r6 = r6 + 1
        br r6 < 100, Inner              br r6 < 100, Inner
        r5 = r5 + 1                     r5 = r5 + 1
        produce [9] = r5                consume r5 = [9]
        RELEASE 0                       RELEASE 0
        br r5 < 100, Outer              br r5 < 100, Outer
```

**Figure 4: Coalescing at the outer loop.**

```
        Producer Thread                 Consumer Thread

Outer:  ACQUIRE  0              Outer:  ACQUIRE 0
        produce [4] = r5                consume r5 = [4]
Inner:  ACQUIRE  1              Inner:  ACQUIRE 1
        produce [7] = r6                consume r6 = [7]
        r6 = r6 + 1                     r6 = r6 + 1
        RELEASE 1                       RELEASE 1
        br r6 < 100, Inner              br r6 < 100, Inner
        r5 = r5 + 1                     r5 = r5 + 1
        produce [9] = r5                consume r5 = [9]
        RELEASE 0                       RELEASE 0
        br r5 < 100, Outer              br r5 < 100, Outer
```

**Figure 5: Coalescing at loop entry and exits.**

each procedure, the analysis outputs the syncnos used in the procedure, the direction of synchronization (i.e. produce or consume), the `ACQUIRE` and `RELEASE` points for each syncno and its $SEG$.

# 4. CODE GENERATION

The code generation phase first creates memory locations for all syncnos handed out. Then, for each procedure, for each syncno used in that procedure, it uses analysis information to insert `ACQUIRE`s and `RELEASE`s for that syncno at the specified points. It uses direction information for each syncno to determine the exact condition variable values (or occupancy counter operations) to use while generating the `ACQUIRE`s and `RELEASE`s. Either concurrently or as a later pass, `produce` and `consume` instructions can be converted into `store` and `load` instructions to memory locations.

In our implementation, we used condition variable based synchronization. The data layout of these per-entry condition variables were similar to queue data layouts (i.e. each condition variable also took up 8 bytes and had 64 entries corresponding to each queue entry). This layout enabled the code generator to coalesce queue pointer updates for all $SEG$s, by doing queue pointer updates only for the condition variable queue corresponding to each syncno, and using that offset as the offset for operand queue accesses as well.

# 5. EVALUATION

The analysis and the code generation was implemented in the

```
        Producer Thread                 Consumer Thread

Loop:   ACQUIRE  0              Loop:   ACQUIRE 0
        br r4 == r5, If                 br r4 == r5, If
        produce [4] = r5                consume r5 = [4]
        RELEASE  0                      RELEASE  0
        br Loop                         br Loop
If:     br r5 > 0, Loop         If:     br r5 > 0, Loop
```

**Figure 6: Control inequivalent ACQUIRE and RELEASE.**

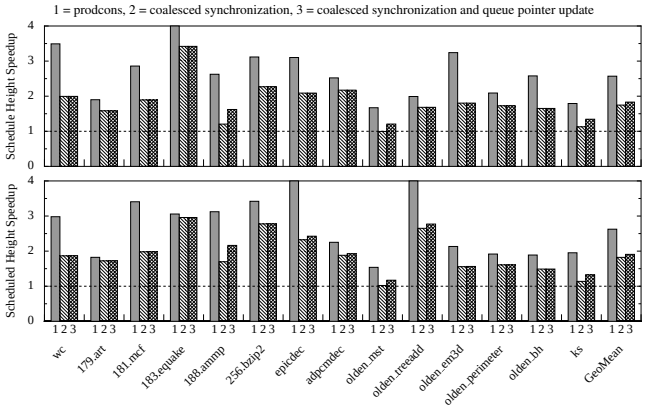



**Figure 7: Schedule height speedup in the producer (above) and consumer (below).**

Velocity compiler [10] framework. As a preliminary measure of the effectiveness of the propose optimization, we measured the speedup in profile-weighted schedule heights of 2-thread DSWP codes (Figure 7) relative to naïvely implemented software queues. The codes used three different types of communication support - `produce` and `consume` instructions ('prodcons' bar) (used in [4, 5]), software queues with synchronization coalescing ('coalesced synchronization' bar) and software queues with joint synchronization and queue pointer update coalescing ('coalesced synchronization and queue pointer update' bar). Our benchmark suite consisted of the Unix utility `wc`, and benchmarks drawn from SPEC 2000, MediaBench and Olden suites. We optimized one key loop in each of the benchmarks. While schedule heights are not a true reflection of the performance of a piece of code (since run-time effects like cache behavior and branch prediction behavior often tend to mask performance improvements predicted by pure schedule height measurements), such comparisons definitely enable us set rough expectations for the code performance. Overall, as seen from the geometric mean bars, synchronization coalescing alone can provide a speedup of 1.75x and joint synchronization and queue pointer update coalescing can provide a 1.8x speedup.

# 6. CONCLUSION

Though DSWP has emerged as a promising technique for general-purpose program parallelization, its reliance on special hardware support may impact its widespread commercial use. However, simple optimizations such as the one proposed may facilitate the use of software queues to support inter-thread communication in DSWP. As future work, we plan to study the dynamic behavior of codes with coalesced synchronization and queue pointer updates on both in-order and out-of-order multi-core processors to understand the performance bottlenecks with software queue implementations better. The insights gained from such a study will ultimately enable us to design efficient communication support without relying on dedicated hardware and will hopefully facilitate popular use of DSWP on current and future multi-core processors.

# Acknowledgments

We thank Ram Rajamony for suggesting the synchronization coalescing idea. We thank Neil Vachharajani, Matt Bridges and the entire Liberty Research Group for their help in refining the algorithm and providing support for the Velocity compiler framework.

# 7. REFERENCES


[1] E. Caspi, A. DeHon, and J. Wawrzynek. A streaming multi-threaded model. In *Proceedings of the Third Workshop on Media and Stream Processors*, December 2001.

[2] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–248, 2005.

[3] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

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

[5] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[6] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 112–119, April 1982.

[7] M. Takesue. Software queue-based algorithms for pipelined synchronization on multiprocessors. In *Proceedings of the 2003 International Conference on Parallel Processing Workshops*, October 2003.

[8] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, February 2005.

[9] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 12th International Conference on Compiler Construction*, 2002.

[10] S. Triantafyllis, M. Bridges, E. Raman, G. Ottoni, and D. I. August. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 61–71, June 2006.