Clustered Communication for Efficient Pipelined Multithreading on Commodity MCPs

Yuanming Zhang, Kanemitsu Ootsu, Takashi Yokota, and Takanobu Baba

***Abstract*—Low inter-core communication overheads are critical for pipelined multithreading (PMT) to using multi-core processors (MCPs) to improve the performance of general sequential applications. However, conventional software queue based communication mechanism will bring significant communication overheads, which limit the potential performance and hinder the wide commercial use. While dedicated inter-core communication mechanism has been proposed, it demands chip redesign effort, costs so much and needs extensions to ISA.**

This paper addresses this problem and proposes a novel clustered communication mechanism to minimize the communication overheads from the average standpoint. We observe that the PMT performance is very sensitive to inter-core communication overheads, but is insensitive to amount of parallelisms. Based on the observation, we can achieve very low average communication overheads (ACOs) through sacrificing a certain amount of parallelisms. The principle of clustered communication mechanism and how to reduce the ACOs with this mechanism are presented in detail. A concurrent lock-free clustered software queue algorithm, which applies this mechanism, is given to support the pipelined communication. The algorithm is evaluated on the AMD Phenom four-core processor and experimental results show its communication performance is over 10x faster than that of conventional software queue, and significant PMT performance of real applications are, therefore, achieved.

***Keywords*-Pipelined multithreading; commodity multi-core processors; software queue; clustered communication; low inter-core communication overheads.**

I. Introduction

Multi-core processors (MCPs) have been widely accepted as predominant computing architecture. Machines with two to four cores have dominated current commodity computers, and future machines promise more cores [1], [2]. Although MCPs can improve the performance of multiple applications and multi-threaded applications, they can do nothing to improve the performance of sequential applications. Therefore, it is an important issue to parallelize sequential applications into multi-threaded ones executing on multiple contexts for higher performance.

Recent work (StreamIt[3], Coarse-grained pipeline[4], Decoupled Software Pipelining (DSWP)[5], and others[6]) shows that pipelined multithreading (PMT) techniques have great applicability to parallelizing general programs, such as uncounted loops, control flow and irregular pointer-based memory access[7]. These techniques parallelize sequential applications into multiple stages that are executed by multiple threads. These threads are bound to different cores and run concurrently.

While PMT techniques show great promise, current MCPs are without architectural support for pipelined communication. The significant inter-core communication overheads limit the potential performance and hinder the wide commercial use. The programmers or compilers have to exploit long running threads with minimal communication[4], [8]. Although dedicated communication mechanism for multi-core architecture, termed as synchronization array[9], has been proposed, it demands chip redesign efforts, costs so much and needs extensions to ISA. Software queues[10], [11] avoid these shortcomings. Based on the memory consistency or cache coherence implementation on general processors, the shared memory or shared cache can provide complete support for pipelined communication. However, the main drawback is the software queue will bring significant overheads, which tend to negate most benefit from pipeline or even lead to performance degeneration over the sequential applications.

This paper addresses this problem and proposes a novel *clustered communication mechanism* to minimize the communication overheads from the average standpoint. Our researches show that the PMT performance is very sensitive to the delay from frequent communication operations and inter-core transit delay, but is insensitive to the amount of parallelisms. Based on the observation, we can achieve very low average communication overheads (ACOs) by sacrificing a certain amount of parallelisms. A concurrent lock-free clustered software queue algorithm, which applies the clustered communication mechanism, is given. Experiments on the AMD Phenom four-core processor show that the clustered software queue is over 10x faster than the conventional software queue. Further evaluation on real applications also demonstrates that it is very effective to improve the PMT performance. Thus, the clustered communication mechanism makes it possible to construct efficient PMT on commodity MCPs without any specific hardware support.

The rest of this paper is organized as follows. Section II analyzes the PMT performance characteristics and motivates this research. Section III presents the principle of clustered

Manuscript received June 25, 2009.

Department of Information Science, Graduate School of Engineering, Utsunomiya University; 7-1-2 Yoto, Utsunomiya city, Tochigi, Japan. Tel/Fax:+81-28-689-6262; Emails: zym@virgo.is.utsunomiya-u.ac.jp,{kim, yokota, baba}@is.utsunomiya-u.ac.jp.

```
1. while(node=node->child){
2.    sum+= node->value;
      }
```
(a) Loop containing recursive data structure

```
while(node=node->child){
  enqueue(node);
}
```
(b) Producer thread

```
while(node=dequeue()){
   sum+= node->value;
}
```
(c) Consumer thread

Figure 1. A PMT example

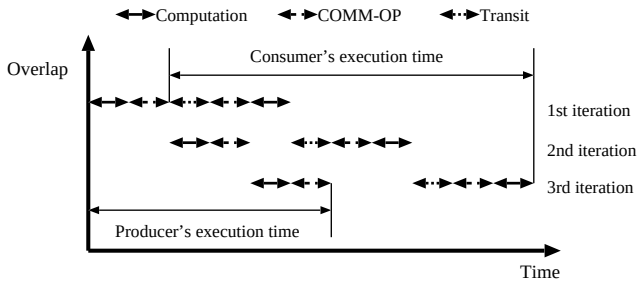


Figure 2. Effect of COMM-OP and transit delays on PMT performance

communication and analyzes how to reduce communication overheads with this mechanism. Section IV gives a clustered software queue algorithm, which applies the clustered communication mechanism, to support single-producer and single-consumer pipelined communication. Section V evaluates the clustered software queue on commodity MCPs and also demonstrates its effectiveness on some real applications. The related work is discussed in Section VI, and Section VII concludes the paper.

II. PMT Performance Characteristics

This section analyzes the PMT performance characteristics from two major aspects: communication overheads and amount of parallelisms, to show their effects on the performance. This theoretically illustrates why clustered communication is appropriate for PMT.

A. Sensitive to Communication Overheads

To illustrate the basic concepts of PMT, we give an example, as shown in Figure 1, in which a loop containing recursive data structure is decomposed into two threads: producer thread and consumer thread. These two threads execute different parts of the loop body and run concurrently. One important feature of PMT is the dependent data between stages is acyclic. The data only flows in the forward direction. The inter-thread communication is supported by concurrently enqueuing (dequeuing) data via a shared queue. Due to the enqueue and dequeue operations, significant communication overheads occur, which become the PMT performance bottleneck.

The inter-core communication overheads can be further divided into communication operation (COMM-OP) delay and transit delay[10]. The COMM-OP delay refers to the time taken to execute communication instructions to enqueue (dequeue) data into (from) queue. This delay depends on the amount of instructions, synchronization mechanism and specific hardware implementation. Fewer instructions and responsive synchronization mechanism tend to lower COMM-OP delay. The transit delay refers to the time taken to transfer a data value from one core to another. This delay will increase with the physical separation among cores. Compared with shared memory, shared cache due to closer connection among cores tends to lower transit delay.

Figure 2 shows the effect of COMM-OP and transit delays on the PMT performance. Assume the queue has three slots. The executions of producer and consumer overlap at least three times. The producer first does the useful computation, labeled as computation time. Then, it writes the data into the queue if there is empty slot. The time taken to execute enqueue operation is labeled as COMM-OP delay. Since there are three slots, it can execute next two iterations without waiting for the consumer. The time taken to execute one producer's iteration is the sum of *computation time* and *COMM-OP delay*. Since the producer does not request data from other cores and only write the data into the local cache, there is no inter-core transit delay.

The consumer thread, on the other hand, first executes the dequeue operation and reads the data. Since the requested data exists in the remote (producer's) cache, it has to fetch the data to the local (consumer's) cache. The time taken to fetch data is labeled as transit delay, and the time taken to execute dequeue operation is also labeled as COMM-OP delay. After getting the data, the consumer continues to do the useful computation, also labeled as computation time. As a result, the time taken to execute one consumer's iteration is the sum of *transit delay, COMM-OP delay* and *computation time*.

Besides the two kinds of overhead, there also are other overheads. Since some queue slots may be located in a single cache line, the concurrent access to a single cache line will incur cache line ping-ponging between producer and consumer, and result in frequent cache misses. This overhead, referred to as false sharing overhead, will increase the COMM-OP delay. Furthermore, the actual data bandwidth on specific machine also affects the COMM-OP and transit delays.

The above analysis shows that the COMM-OP and transit delays are both on the *critical execution path* of PMT, and they directly prolong the execution time of PMT, so the PMT performance is very sensitive to these delays, especially for fine-grained pipeline, and reducing them can directly improve the PMT performance.

B. Insensitive to Amount of Parallelisms

The basic parallelization feature of PMT is the threads' execution overlaps each other. The overlap times will affect

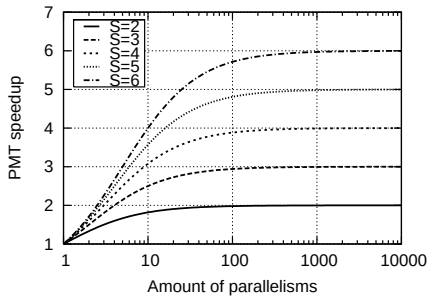


Figure 3. Effect of amount of parallelisms on PMT performance

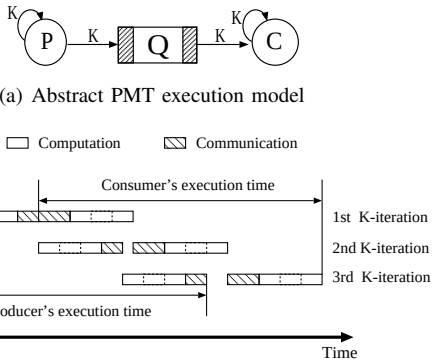
Table I
RATIO OF LOST SPEEDUP TO MAXIMUM SPEEDUP

Stage number	Amount of parallelisms (N)				
	10	50	100	150	200
2	0.091	0.020	0.010	0.007	0.005
3	0.167	0.038	0.020	0.013	0.010
4	0.231	0.057	0.029	0.020	0.015
5	0.286	0.074	0.038	0.026	0.020
6	0.333	0.091	0.048	0.032	0.024

how much benefit can be obtained from the pipelined execution. We use the *amount of parallelisms* to denote the overlap times. This subsection tries to show the effect of amount of parallelisms on performance, and concludes that the performance becomes insensitive to the amount of parallelisms, once it is large enough. When the dependent data is communicated one by one between stages, the amount of parallelisms equals to the iteration number.

To illustrate the effect of amount of parallelisms on PMT performance, we assume an ideal pipeline, in which the communication overheads are neglected, and the pipeline stage sizes are well balanced. An general speedup formula, $\frac{\sum_1^S T_i}{T_L + \frac{\sum_{i \neq L} T_i}{N}}$, can be concluded to calculate the PMT speedup, where N is the amount of parallelisms, S is the stage number, T_i is the execution time of stage i , and stage L is the longest stage. Because the stages are well balanced (T_1=T_2=...=T_S), the formula can be further simplified to $\frac{N \times S}{N+S-1}$. This means for a specific pipeline the PMT performance only depends on the stage number and the amount of parallelisms. According to this speedup formula, we plot the PMT speedup, shown in Figure 3, with the increase of amount of parallelisms, when the stage number is 2, 3, 4, 5 and 6 respectively.

This figure shows that the performance improvement is nonlinear with the increase of amount of parallelisms. When it is large enough, the PMT performance is almost close to the theoretical maximum performance. Table I lists the ratio of lost speedup to theoretical maximum speedup, $\frac{SP_\infty - SP_N}{SP_\infty}$, when the amount of parallelisms is 10, 50, 100, 150 and 200. For example, the ratio of lost speedup to theoretical maximum speedup for a four-stage pipeline is 5.7%, when N is 50. This fact shows that the PMT performance becomes *insensitive* to the amount of parallelisms, only if it is large enough. Although we utilize the above ideal pipelines to illustrate the effect of amount of parallelisms on the performance, this fact also applies to general pipelines.



(a) Abstract PMT execution model

(b) Execution procedure of producer and consumer

Figure 4. Pipeline execution model with clustered communication

Considering the PMT performance characteristics, we propose a novel clustered communication mechanism below for pipelined communication, which can minimize the communication overheads from the *average standpoint* through sacrificing a certain amount of parallelisms. Since the communication overheads are reduced, the PMT performance can be improved.

III. CLUSTERED COMMUNICATION MECHANISM

A. Principle of Clustered Communication

In conventional pipeline, the data between stages is communicated one by one in the forward direction, and the execution is driven by the first stage. The producer (P) produces a single data item and then enqueues this item into the queue (Q). When the consumer (C) finds there is available data item in Q, it will dequeue the item and do the useful work.

If the producer delays the enqueue operation until it produces multiple data items, and the consumer delays the useful computation until it dequeues multiple data items, the pipeline will execute in a different model[15], as shown in Figure 4 (a). In this model, the communication unit between stages is not a single data item, but multiple data items. We call this kind of communication mechanism *clustered communication*. The meaning of "clustered" is the multiple data items are clustered together as a communication unit.

Figure 4 (b) further illustrates the execution process of producer and consumer threads. In each pipeline frame, the producer will execute multiple (K) iterations and then

enqueue the multiple data items. The consumer will dequeue multiple data items and then execute multiple iterations. The labeled computation time refers the time taken to execute multiple iterations, and the labeled communication time refers the time taken to enqueue (dequeue) multiple data items. When K is one, the PMT will execute like conventional pipeline.

The clustered communication mechanism has the following five features:

1) Delayed threads' execution. The producer delays the enqueue operation until it completes specified multiple iterations, and the consumer delays the useful computation until it dequeues specified multiple data items.
2) Fully utilize high level caches. The produced data items can be temporarily stored in the producer's cache before being communicated. The total useful computation time of producer and consumer is not changed, but the inter-core communication overheads can be greatly reduced.
3) Low average communication overheads (ACOs). The key advantage of this mechanism is the ACOs per each data item are greatly reduced. Next subsection illustrates the reasons in detail. It is by reducing the ACOs that the PMT performance is improved.
4) The number of clustered data items, called clustered data number (CDN), is an important parameter. It greatly affects the actual communication performance, and it is critical to select an appropriate CDN for ideal PMT performance.
5) As a penalty, a certain amount of parallelisms is sacrificed. The left amount of parallelisms is $\lceil \frac{N}{K} \rceil$, where K is the CDN value. This penalty is acceptable, since the PMT performance is insensitive to the amount of parallelisms (only if $\lceil \frac{N}{K} \rceil$ is large enough).

B. Communication Overheads Reduction

Next, we analyze how the communication overheads are reduced by this mechanism. As illustrated above, the communication overheads mainly include COMM-OP delay and transit delay. Besides that, the false sharing also should be eliminated to avoid the extra overhead due to frequent cache misses.

False Sharing Elimination: False sharing occurs when the producer and consumer will be accessing the queue slots located in a single cache line. It will lead to significant coherence traffic and bring significant extra overhead. Thus, it is very important for software queue to eliminate false sharing.

In clustered communication mechanism, the communication unit is composed of multiple data items and is like a *chunk* that can only be accessed by one thread (producer or consumer) at a time. During the time when the producer (consumer) is enqueuing (dequeuing) the chunk, the consumer (producer) is not permitted to access any data item of the chunk. This chunk may occupy one or several cache lines, and forces the producer and consumer can only access data items located in different cache lines. The CDN determines the chunk size. There exists a theoretical minimum CDN to make the chunk size equal to the cache line size. For example, assume the data item size is 8 bytes, and the cache line size is 64 bytes. The theoretical minimum CDN will be 8 to make the chunk occupy a single cache line. A larger CDN value is also effective to eliminate false sharing, since the chunk will occupy several cache lines. Compared with previous approach of eliminating false sharing by inserting empty pads[10], the clustered communication can avoid wasting large portions of cache space and make the data transfer more efficient.

Low Average COMM-OP Delay: In conventional single datum queues, each datum requires one communication operation and results in one COMM-OP delay. The total delays are the COMM-OP delay multiplied by iteration number (N). However, in clustered communication, multiple data items require only one communication operation, and the actual total delays are COMM-OP delay multiplied by $\lceil \frac{N}{K} \rceil$. Furthermore, the synchronization delay can also be reduced, since the condition variable (or control data) will not be frequently modified by the producer and consumer. This reduces the significant contention on the same location. On the average standpoint, the COMM-OP delay will be reduced for each data item.

Low Average Transit Delay: Generally, the data transfer unit between cache to cache or cache to memory is a single cache line. Although the requested datum only occupies several bytes of one cache line, the whole cache line has to be transferred. Because of this, conventional single datum queues not only make the inter-core data transfer inefficient, but also will incur conflict access to the same cache line. However, the clustered communication mechanism can effectively avoid these shortcomings. As illustrated early, the communication unit can occupy one or several cache lines. The data transfer unit is one or several cache lines data items, which make the data transfer much more efficient. Furthermore, clustered communication also provides a chance for a hardware prefetch unit to reduce the penalties of compulsory cache line misses by transparently prefetch cache lines into the consumer's high level cache. Prefetching provides additional potential for performance improvement. Thus, the average transit delay can be reduced overall.

The above analysis shows the clustered communication mechanism can reduce the communication overheads from the average standpoint. The actual ACOs can be estimated by O/K, where O is the time taken to enqueue/dequeue K data items within one operation. Since the COMM-OP and transit delays are both on the critical execution path, reducing them can result in higher PMT performance.

IV. Clustered Software Queue Algorithm

This section gives a concurrent lock-free (CLF) clustered software queue algorithm, which applies the clustered communication mechanism. This algorithm is suitable for the single-producer/single-consumer (SP/SC) pipelined communication. This section also discusses the policy to select an appropriate CDN parameter to achieve ideal PMT performance.

A. Queue Structure and Algorithm

To implement the clustered communication, we should make sure that the multiple data items can only be accessed by one thread at a time. That is, before the producer completely enqueues the multiple data items, the consumer is not permitted to dequeue any data item. And, before the consumer completely releases the queue slots, the producer is not permitted to enqueue any data item.

Figure 5 shows a two-level array based queue structure, which is composed of a tail index, a head index, and an array based slots in the first level. And in the second level, each slot is further composed of a pair of nodes: a flag condition variable and a chunk of array based sub slots. Figure 6 shows the CLF clustered software queue algorithm based on the two-level queue structure. Since only a single producer enqueues the data and a single consumer dequeues the data, no explicit mutex is required to protect the queue[12]. The tail and head indices are updated exclusively by the producer and consumer, and they as thread-local variables can be stored locally in the producer's and consumer's cores respectively. This avoids any penalty for data coherent. The queue utilizes conditional variable as fine-grained signal to make sure the correct sequential access of producer and consumer to the queue slots (first-level). The chunk nodes containing multiple sub slots are used to store the multiple data items.

To enqueue data items, the producer (consumer) will spin[13] until the flag condition variable shows the queue slot is empty (full). Then it will acquire the right to access the sub slots to enqueue (dequeue) all data items. The data items are enqueued (dequeued) between the chunk node and the private data array. This kind of *memcpy* block data transfer achieves much higher bandwidth. After that, the flag condition variable is signaled, and the tail (head) index is updated to point to next queue slot.

Two parameters, SLOTS and SUB_SLOTS, are defined in the queue. The SLOTS represents the available queue slot size, and the SUB_SLOTS corresponding to the CDN value represents the available sub slot size. The actual total queue size is SLOTS$\times$SUB_SLOTS. These two parameters affect the actual communication performance.

It is easy to prove the correctness of above algorithm. Like conventional single datum software queues, the clustered software queue treats the multiple data items as a chunk. The flag condition variable, head and tail indices ensure the producer and consumer can only access one queue slot each time. Since each queue slot contains multiple sub slots which are closely coupled, it makes the multiple data items can only be accessed by one thread at a time. The sub slot is implemented based on array structure, which makes the data items located in the contiguous spaces and improves the efficiency of data transfer. As a result, accessing multiple data items will be accessing one or several cache lines.

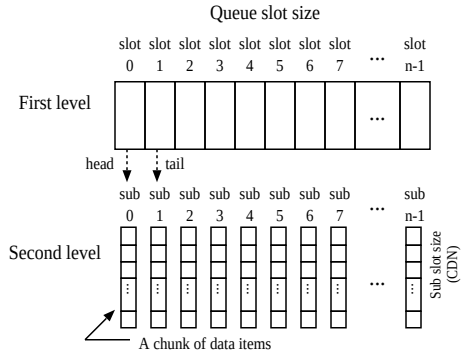


Figure 5. Two-level array based queue structure

```
void enqueue_clustered (int *data) {
   //if all slots are full, spin
   while (queue[tail].flag) ;
   //enqueue a chunk of data items
   memcpy(queue[tail].chunk,data,
          SUB_SLOTS*sizeof(int));
   queue[tail].flag=1;
   tail=(tail+1)%SLOTS;
}
```

```
void dequeue_clustered(int *data) {
   //if all slots are empty, spin
   while (!queue[head].flag) ;
   //dequeue a chunk of data items
   memcpy(data,queue[head].chunk,
          SUB_SLOTS*sizeof(int));
   queue[head].flag=0;
   head=(head+1)%SLOTS;
}
```

Figure 6. Clustered software queue algorithm

B. Policy of Clustered Data Number Selection

The clustered data number (CDN) is an important parameter. It greatly affects the communication performance of clustered software queue. Experiment results (given in evaluation section) show that there is a regular reduction pattern of ACOs with the increase of CDN, and the ACOs become stable when the CDN is large enough ($\geq$64) in spite of a little reduction.

Since the amount of parallelisms will be reduced due to the clustered communication, we can select an appropriate CDN for applications according to the iteration number and the regular reduction pattern. For large iteration number, it is natural to select a large CDN (such as 64) for lower ACOs. This can result in higher PMT performance. For small iteration number, it is a trade-off on the left amount of parallelisms and the ACOs. For actual applications, the loop should have at least 10^3 iterations for ideal PMT performance.

Table II
EXPERIMENTAL PLATFORM

Cores	AMD Phenom 9600, four cores, 2.3GHz
	L1: 128KB*4, 64B line size, 3-cycle latency
	L2: 512KB*4, 64B line size, 13-cycle latency
Shared cache	L3: 2MB*1, 64B line size, 32-cycle latency
Memory	4.0GB, 138-cycle latency

V. EVALUATION

This section evaluates the above clustered software queue algorithm on commodity MCPs. Experimental results show that the clustered software queue can achieve much higher communication performance, and is also effective on improving the PMT performances of real applications.

The AMD Phenom[14] was used as the experimental platform. This platform has four cores. One important feature of the multi-core processor is that it has *a L3 cache shared among the four cores*. It provides low inter-core communication latency. Detailed parameters, such as cache line size, cache and memory latencies, are listed in Table II.

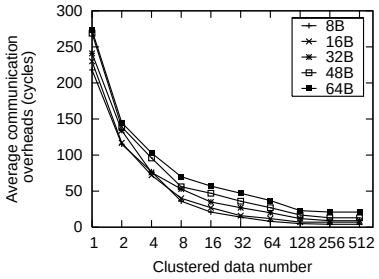
We first used the sample loop of Figure 1, which is isolated from the benchmarks, to evaluate the clustered software queue. This two-stage pipeline was rewritten to support the clustered communication with the approach in[15]. This sample loop provides conveniences to control the queue parameters, such as queue slot size, sub slot size and data size. The iteration number was set 10,000.

To demonstrate the effectiveness of clustered software queue, we also applied the clustered software queue to some real applications extracted from SPEC CPU 2000 benchmarks. The results show that the PMT performances were generally improved so much compared with the performances without using this mechanism.

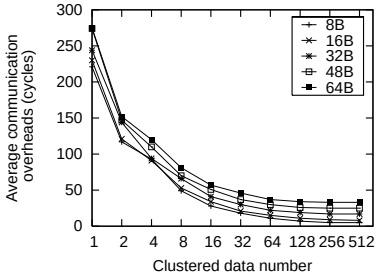
A. Communication Performance

As the clustered software queue utilizes a two-level array structure, we first kept queue slot size (first level) constant and set its size 64, and varied the sub slot (second level) different sizes, which correspond to CDN values. The average communication overheads (ACOs) were calculated through dividing the execution time of single enqueue or dequeue operation by CDN value.

In the evaluation, we also considered the impact of data size. Two factors affect the data size. One is the data type. Each data type has its own data size. Another is the data count. In practice, it is usual that there is more than one data dependence between stages. To avoid inserting multiple communication operations, compound structure may be used to pack the multiple data dependences into a compound data, which is forwarded as one data item. The actual compound data size varies with the contained data count. We set the data size 8B, 16B, 32B, 48B and 64B.



(a) Enqueue operation



(b) Dequeue operation

Figure 7. Communication performances of enqueue and dequeue operations with the increasing of CDN value

Figure 7 shows the communication performances of enqueue and dequeue operations with the increasing of CDN value. The x axis represents the CDN values, and the y axis represents the ACOs (cycles) for a specific CDN. For each data size, there is one curve showing how the ACOs are reduced with the increase of CDN.

A common characteristic of the two operations is that their ACOs are both reduced along with the increase of CDN. The curves present a regular reduction pattern of ACOs, which may be divided into two reduction stages. In the first stage ($<$64), the reduction of ACOs is very soon and great. The reasons of that are the false sharing is eliminated and the average COMM-OP and transit delays are reduced at the same time. In the second stage ($\geq$64), the reduction becomes slow and stable. This shows that the false sharing has been eliminated and the COMM-OP delay and transit delay also have been reduced almost to their lower limits. The regular reduction pattern shows that the ACOs have a lower limit and can't be continually reduced through increasing CDN. We use the value 64 to divide the two stages, since the ACOs become almost stable after this value in spite of a little reduction.

Comparing the curves for different data sizes, we can find that larger data size tends to larger ACOs for the same CDN. The key reason is the larger data size will take more data transfer time. For producer, the data is transferred between high level cache to the shared cache (to memory in first access), and for consumer the data is transferred among the local cache to the remote cache. When CDN is 64, the ACOs

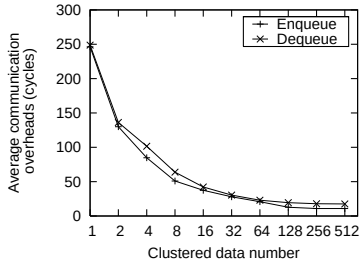


Figure 8. The average communication performances of enqueue and dequeue operations across all data sizes

of enqueue operation are 8 cycles, 12 cycles, 20 cycles, 27 cycles and 37 cycles corresponding to the data size 8B, 16B, 32B, 48B and 64B.

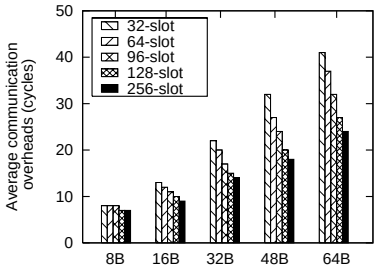
Figure 8 compares the average performances of enqueue and dequeue operations across all the five data sizes. This figure shows that although the performance of enqueue operation is generally higher than that of dequeue operation, the actual performance gap is not so large. This shows that the shared cache is effective to lower the inter-core transit delay.

To calculate the actual communication performance improvement, we compared the performances when CDNs are 1 and 64. When CDN is 1, the clustered software queue is same to the conventional single datum queue. We selected 64 as the CDN value, because this value is not so large, but can result in much low ACOs. The ACOs of enqueue operation are 246.2 cycles and 20.8 cycles, and the ACOs of dequeue operation are 248.8 cycles and 23 cycles, corresponding to the CDN values 1 and 64. As a result, the performances of enqueue and dequeue operations are 11.8x and 10.8x faster than those of single datum software queue.

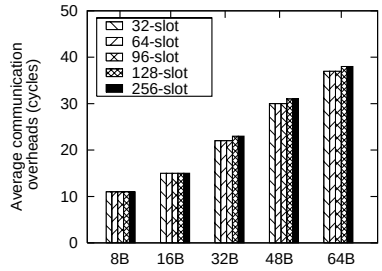
B. Impact of Slot Size on Communication Performance

In above experiments, the queue slot size was constant and was set 64. We further evaluated its impact on the communication performance. We kept the sub slot size (CDN) constant and set the value 64, but varied the queue slot size from 32 to 256. The five data sizes were same to the above experiments.

Figure 9 shows the impact of queue slot size on the performances of enqueue and dequeue operations. For each data size, there is a group of bars that show the ACOs for five queue slot sizes (32-slot, 64-slot, 96-slot, 128-slot and 256-slot). According to the figures, we found that the performance of enqueue operation is *sensitive* to the queue slot size. The performance is improved for large queue slot size, especially when forwarding large data size, such as the 64-byte data size. While, the performance of dequeue operation is almost *insensitive* to the queue slot size for all data sizes. These two figures also clearly show the actual communication performances of enqueue operation



(a) Enqueue operation



(b) Dequeue operation

Figure 9. Impact of queue slot size on communication performances of enqueue and dequeue operations

and dequeue operation. The average ACOs of enqueue (dequeue) operation across the five queue slot sizes are 7.6 (11) cycles, 11 (15) cycles, 17.6 (22.4) cycles, 24.2 (30.4) cycles and 32.2 (37.4) cycles, corresponding to the data size 8B, 16B, 32B, 48B and 64B.

C. Effectiveness on Improving PMT Performance

We have shown the clustered software queue can achieve higher communication performance with an appropriate CDN. We further evaluate its effectiveness on improving PMT performance.

Five loops were extracted from SPEC CPU 2000 benchmarks, as shown in Table III. The 181.mcf and 188.ammp benchmark loops contain recursive data structure, and the 179.art, 183.equake and 300.twolf benchmark loops contain cross-iteration data dependences, so they all could not be parallelized with DOALL parallelism[7]. The queue slot size and the sub slot size were both set 64 (so the total queue size is 4096). The loops whose iteration number is under 10^3 were excluded to keep enough parallelisms (at least 10 for two-stage pipeline). Compound structure was utilized to pack multiple data dependences.

Because the experimental platform has only *four* cores, we decomposed the benchmark loops into *four* threads at most. Taking into account the relative sizes of computation and communication, we decomposed the 181.mcf and 188.ammp loops into two-threaded pipeline and decomposed 179.art, 183.equake and 300.twolf loops into four-threaded pipelines[15], which contain parallel sub stages, as shown in the last column of Table III, to balance the stage sizes.

Table III
STATISTIC INFORMATION OF SPEC CPU 2000 BENCHMARK LOOPS

Benchmark	Function	Dependence count	Data size	Iteration number	$\lceil \frac{N}{K} \rceil$	Decomposed pipeline
181.mcf	refresh_potential()	1	8B	18101	283	one producer/one consumer
188.ammp	a_tether()	1	8B	9582	150	one producer/one consumer
179.art	simtest2()	1	8B	10000	157	one producer/three consumers
183.equake	smvp()	2	16B	30169	30	one producer/three consumers
300.twolf	initialize_cost()	1	8B	1920	30	three producers/one consumer

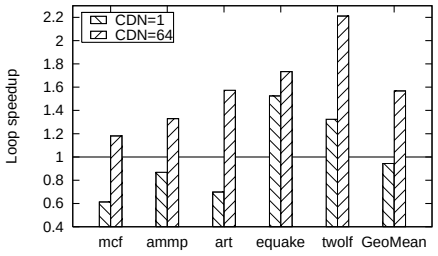


Figure 10. PMT performance over single threaded version when CDN=1 and CDN=64

The single threaded (baseline) and the multi-threaded applications were compiled with the GNU C compiler.

Figure 10 shows the PMT performances of the five selected loops over the single threaded version. For each loop, the graphs compare the loop speedups obtained by setting CDN 1 and 64 (this is the *only* difference). When CDN is 1, the speedup ranges from 61.3% to 152.4%, and the geometric mean speedup is 94.4% across these loops. In comparison, when CDN is 64, the speedups range from 118% to 221.1%, and the geometric mean speedup is 156.7%. The 183.equake benchmark loop is a long running loop body, which makes the speedup not so sensitive to the communication overheads, and the loop speedup is improved from 152.4% to 173.3%. The 300.twolf benchmark loop is decomposed into a two-stage pipeline, in which there are three producer threads and one consumer thread. The parallel sub stages greatly improve the PMT performance. The experiment results indicate that the clustered software queue with an appropriate CDN value is effective on improving the PMT performance.

VI. RELATED WORK

Concurrent software queues have been widely researched in multiprocessor architecture to ensure the consistency of concurrent accesses. Generally, algorithms for concurrent software queues have two principal strategies: blocking and non-blocking (lock-free). Blocking algorithms mostly use critical sections protected by mutual exclusion locks. Since the performance of mutual exclusion locks is degraded significantly in parallel applications due to busy-waiting[16], lock-free algorithms were widely researched and used in parallel applications and operating systems[17], [18], [19], [20], [21], [22].

Most of the previous work on CLF software queues focused on providing general implementation for multi-producer/multi-consumer (MP/MC). Based on some hardware atomic primitives, such as compare-and-swap (CAS) or load-linked/store-conditional (LL/SC), provided on modern processors, these algorithms may guarantee at least one process of those trying to concurrently update queue will succeed in completing its operation within finite time. However, the ABA problem[16] introduced by the synchronization primitives should be avoided to keep the correctness of concurrent accesses. Michael[16], Ladan-Mozes [17], Prakash[19], Scherer III[20] implemented efficient CLF software queues with hardware atomic primitives and presented their methods to avoid the ABA problem.

As a special case of MP/MC queues, the SP/SC software queues are suitable for the pipelined communication. Some extra operations in MP/MC queues are not necessary any more. For example, the ABA problem does not exist in SP/SC queues. Lamport[12] proved that the locks could be removed in SP/SC case under sequential consistency, and presented the well known Lamport's queue that removed the explicit synchronization at the algorithmic level. However, in the Lamport's queue, there are still coupled control data (head and tail indices) between producer and consumer. The producer and consumer have to frequently modify the same control data and result in poor performance. Giacomoni[11] further eliminated the implicit synchronization on the control data by replacing the head and tail indices with NULL, and presented the FastForward queue for efficient pipelined communication. To eliminate false sharing, it introduced a temporal slip technique. In comparison with the clustered communication, the temporal slip couldn't reduce the COMM-OP delay and transit delay, so the communication performance was limited.

Compared with the previous SP/SC software queues, the clustered software queue technique applies a clustered communication mechanism that can minimize the inter-core

communication overheads from the average standpoint. It is not only can avoid the penalty from false sharing, but also can greatly reduce the average COMM-OP delay and average transit delay. Actual experiments showed this clustered software queue can achieve much higher communication performance, and therefore can significantly improve the PMT performance.

VII. Conclusions

Although pipelined multithreading (PMT) techniques have shown great promise to parallelizing general programs for higher performance, significant inter-core communication overheads limit the potential performance and hinder the wide commercial use. To handle this problem, this paper proposed a clustered communication mechanism, which can achieve much higher average communication performance by eliminating false sharing and reducing COMM-OP and transit delays. A CLF clustered software queue algorithm, which applied this mechanism, was given. Actual experimental results showed its communication performance was over 10x faster than that of conventional software queue, and higher PMT performance improvement, therefore, was achieved. As a result, the clustered communication mechanism makes it possible to construct efficient PMT on commodity multi-core processors without relying on specific hardware.

Acknowledgement

This research was supported in part by Grant-in-Aid for Scientific Research ((C)21500050 and (C)21500049, (C)2050047) of Japan Society for the Promotion of Science (JSPS), and by Eminent Research Selected at Utsunomiya University.

References

[1] AMD Corporation, *Multi-core Processors: The Next Evolution in Computing*, http://multicore.amd.com/Resources/33211A_Multi-Core_WP_en.pdf

[2] Intel Corporation, *Intel Multi-Core Processors: Making the Move to Quad-core and Beyond*, http://www.cse.ohio-state.edu/ panda/775/slides/intel_quad_core_06.pdf

[3] W. Thies, M. Karczmarek, and S. Amarasinghe, *StreamIt: A Language for Streaming Applications*, In Proceedings of the 11th International Conference on Compiler Construction, April, 2002, 179-196.

[4] W. Thies, V. Chandrasekhar, and S. Amarasinghe, *A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs*, In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, Dec.,2007, 356-369.

[5] R. Rangan, N. Vachharajani, G. Ottoni, and D. I. August, *Performance Scalability of Decoupled Software Pipelining*. ACM Transaction on Architecture and Code Optimization. Vol 5, NO.2, 2008, pp.1-25.

[6] 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, Jun. 2005, pp.237-248.

[7] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, Publishers Inc. 2002.

[8] G. Ottoni, R. Rangan, A. Stoler, M. J. Bridges, D. I. August, *From Sequential Programs to Concurrent Threads*. IEEE Computer Architecture Letters, Vol 5, 2006, pp.6-9.

[9] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, *Decoupled Software Pipelining with the Synchronization Array*. In Proceedings of 13th International Conference on Parallel Architecture and Compilation Techniques, Sep., 2004, pp.177-188.

[10] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. Cai, *Support for High-Frequency Streaming in CMPs*. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, Dec., 2006, pp.259-272.

[11] J. Giacomoni, T. Moseley, and M. Vachharajani, *FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-free Queue*. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Feb., 2008, pp.43-52.

[12] L. Lamport, *Specifying Concurrent Program Modules*. ACM Transactions on Programming Languages and Systems, 5(2), 1983, pp. 190-222.

[13] T. E. Anderson, *The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors*. IEEE Transactions on Parallel and Distributed System, 1(1),1990, pp.6-16.

[14] AMD Phenom 9600. http://products.amd.com/en-us/DesktopCPUResult.aspx

[15] Y.M. Zhang, K. Ootsu, T. Yokota, and T. Baba, *Clustered Decoupled Software Pipelining on Commodity CMP*. In Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems, Dec. 2008, pp.681-688.

[16] M. M. Michael and M. L. Scott, *Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared Memory Multiprocessors*. Journal of Parallel and Distributed Computing, 51(1), 1998, pp.1-26.

[17] E. Ladan-Mozes and N. Shavit, *An Optimistic Approach to Lock-free FIFO Queues*. In Proceedings of the 18th International Conference on Distributed Computing, volume 3274, 2004, pp.117-131.

[18] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit,*Using Elimination to Implement Scalable and Lock-free FIFO Queues*. In Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, 2005,pp. 253-262.

[19] S. Prakash, Y. H. Lee, and T. Johnson, *A Nonblocking Algorithm for Shared Queues Using Compare-And-Swap*. IEEE Transactions on Computers, 43(5), 1994, pp.548-559.

[20] W. N. Scherer III, D. Lea, and M. L. Scott, *Scalable Synchronous Queues*. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2006, pp. 147-156.

[21] 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, pp. 134-143.

[22] M. P. Herlihy, *A Methodology for Implementing Highly Concurrent Data Objects*. ACM Trans. Progrmg. Lang. Syst. 15, 5, Nov., 1993, pp.745-770.