BatchQueue: Fast and Memory-thrifty Core to Core Communication

Thomas Preud'homme Julien Sopena Gaël Thomas Bertil Folliot

LIP6 - UPMC/CNRS/INRIA
4 place Jussieu, Paris
first.last@lip6.fr

Abstract

Sequential applications can take advantage of multi-core systems by way of pipeline parallelism to improve their performance. In such parallelism, core to core communication overhead is the main limit of speedup. This paper presents BatchQueue, a fast and memory-thrifty core to core communication system based on batch processing of whole cache line. BatchQueue is able to send a 32bit word of data in just 12.5 ns on a Xeon X5472 and only needs 2 full cache lines plus 3 byte-sized variables — each on a different cache line for optimal performance — to work. The characteristics of BatchQueue — high throughput and increased latency resulting from its batch processing — makes it well suited for highly communicative tasks with no real time requirements such as monitoring.

1. Introduction

Monitoring program execution provides useful information to optimize and debug programs, and more generally to study the behaviour of a given system. The downside of monitoring is that it slows down the monitored program proportionally to the quantity of information gathered and processed. However, the increasing number of cores on current processors gives an opportunity to monitor an application at a very fine grain without slowing down the application too much by way of pipeline parallelism.

Pipeline parallelism consists in splitting a sequential task into several smaller successive sequential tasks called stages. Each stage takes as input the output of the previous stage which allows them to all run in parallel by working on different set of data. Pipeline parallelism suffers from two limitations. First, all stages must complete in approximately the same amount of time. The bigger is the difference between the slowest and the fastest stage, the less is the efficiency. Second, the communication overhead must be small compared to the stage duration which implies communication must be very fast to be able to cut a sequential task into many small stages. Although the first limitation is inherent to the task being cut, the second one highly depends on the efficiency of the communication algorithm used.

This article presents BatchQueue, a single producer single consumer queue (SPSC queue) that allows fast core to core communication with small memory needs. BatchQueue uses batch processing of whole cache line to reduce synchronisation between producer and consumer and thus achieve high throughput. BatchQueue is well suited for communicative tasks with no real-time requirement as batch processing incurs increased overhead and needs high level of communication to work. In this regard, monitoring is a particularly good candidate for pipeline parallelism using BatchQueue.

Evaluation of the inter-core communication algorithm shows it fulfills its goal of being fast and memory-thrifty. BatchQueue is able to send a data in just 12.5 ns and only needs 2 full cache lines plus 3 byte-sized variables — each on a different cache line for optimal performance — to work. This represents an improvement of a factor of 10 over the traditionnal Lamport CLF's queue.

The remainder of the paper is organized as follows. First, section 2 presents BatchQueue, the inter-core communication algorithm. The results of the evaluation of BatchQueue are shown in section 3. Then, section 4 presents related work. Section 5 gives a few examples of systems which could benefit from using BatchQueue. Finally, section 6 concludes this paper.

2. Inter-core communication

Efficient core to core communication is a key condition for high speedup with pipeline parallelism: the more efficient the communication is, the more stages a task can be splitted up in. Building an efficient core to core communication requires to avoid synchronisation, not only software synchronisation but also hardware synchronisation which is linked to the cache consistency system. Thus, before presenting the inter-core communication algorithm in section 2.2, section 2.1 gives some details about processor caches behavior on multi-core systems.

2.1. Processor caches behavior

To be efficient, an algorithm not only needs to be the simplest possible it also needs to handle memory with care.

Although Lamport's CLF queue manages to avoid any software synchronization and is very simple [6], it is slower than other more complex approaches which handle memory with more care [16, 7, 2, 17]. Using memory appropriately is a very difficult task [1]. Achieving maximum bandwidth and minimal latency in memory access depends on many parameters, most of them being related to caches. This section focuses on the cache consistency system since it is the less known source of influence on memory access [1]. The section explains the memory coherency protocol with highlight on the limitations it implies and then gives some guidelines to maximize memory access performance.

Cache sharing Multi-core systems obey to the shared memory model. Every core sees the same memory. It allows multi-threaded applications or applications sharing a memory segment to execute on several cores without code modification. For the hardware it means the memory view exposed by the caches must be coherent. In particular, if a thread writes a value at a given location and then a thread reads at that address, it must read what the first thread wrote. And, if it reads the value before it is written and reads it again after the value is written, it must read two different values, whether the old value is still in the cache or not. The cache coherency is ensured by the MESI protocol which runs on each core.

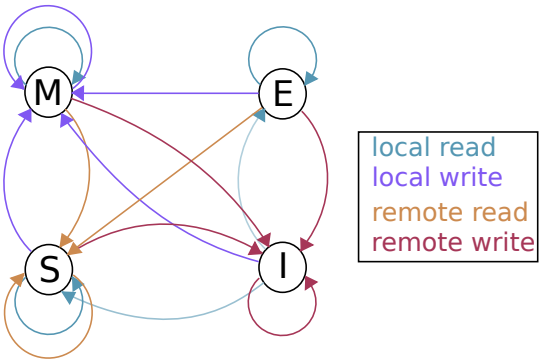


Figure 1. MESI coherency protocol

Cache coherency protocol MESI stands for Modified, Exclusive, Shared, Invalid, the four states in the MESI protocol. All transitions between states are shown in figure 1. Transitions to invalid state when a remote write is performed means the remote core sends an invalidation message to all cores and executes the next instruction only when this is done. It is an expensive operation whose cost depends on the number of cores to notify. But this operation is necessary because it prevents other cores to read a data they have in their cache which is now invalid since a thread has written a new value. The transition from invalid state to another state is also costly since it means getting the cache line from another core, the one which writes last in the cache line.

[1]For an exhaustive study of all sources of influence upon memory access, see [1].

Guidelines for efficient use of cache To achieve optimal latency and bandwidth in memory access, several rules must be followed when laying out data and objects in memory. First, data shall be compacted and cache line aligned. Compaction makes data hold less space and cache line alignment reduces the number of lines data spread on. This has two effects: it reduces pressure on the eviction policy and helps data to stay longer and in a much bigger part in the first level of cache. Then, the working set size should be as tiny as possible. Smaller working set size means lower levels of cache involved so the bandwidth is higher. At last, sharing data shall be avoided as much as possible. Indeed, whether sharing is done between cores with a shared cache or not, at least the first level of cache is not shared which implies the cache coherency protocol is solicited. If sharing is necessary, then frequent interleaving of read and write should be avoided as much as possible. The reason is that a read after a write incurs a remote read of the latest value of the data, while a write after a read by a remote core means an invalidation of the value which has been read is necessary.

2.2. Inter-core communication algorithm

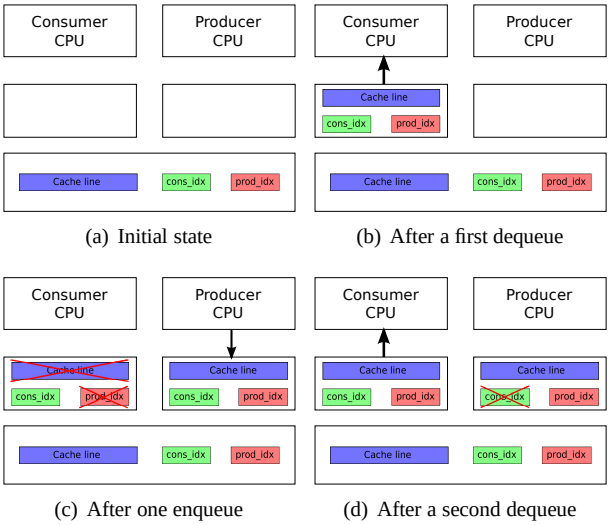
While the previous section explains theory about cache behavior, focusing on effect of the cache consistency system on latency and bandwidth, the current section looks at what happens in the context of core to core communication. First, execution of Lamport's CLF queue is studied from the cache consistency system's point of view, then BatchQueue is compared against it in the same context.

```
1  void enqueue(int data) {
2    while ((prod_idx + 1) % N == cons_idx);
3    tab[prod_idx] = data;
4    prod_idx = (prod_idx + 1) % N;
5  }
6
7  int dequeue() {
8    while (cons_idx == prod_idx);
9    int result = tab[cons_idx];
10   cons_idx = (cons_idx + 1) % N;
11   return result;
12 }
```

Figure 2. Lamport's CLF queue algorithm

Lamport's CLF queue Consider the Lamport's CLF queue code in figure 2, whose workflow is presented in figure 3, where tab is an array of size N shared between the consumer and the producer. The consumer and the producer execute on separate cores.

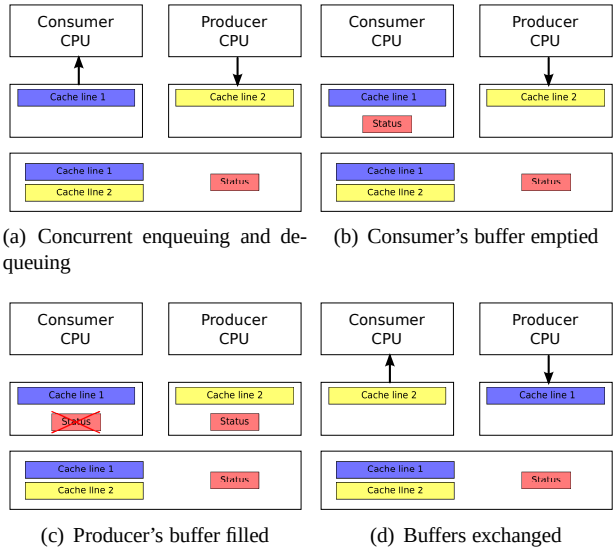
Initially caches do not contain any data related to the algorithm (see figure 3(b)). When the consumer reads a data from the tab array, the consumer's cache gets the data from the cache line which contains that particular cell but also surrounding cells (see figure 3(b)). If that cache line is not in the cache, it has to retrieve it from the cache of the core executing the producer. The line then enters in shared state on caches of both cores according to figure 1. When the producer writes a new data in the array it has to invalidate the cache line of the consumer (see figure 3(c)). If



(a) Initial state (b) After a first dequeue

(c) After one enqueue (d) After a second dequeue

Figure 3. Lamport's CLF queue workflow



(a) Concurrent enqueuing and dequeuing (b) Consumer's buffer emptied

(c) Producer's buffer filled (d) Buffers exchanged

Figure 4. BatchQueue workflow

the consumer is fast, each write occurs after a read from the consumer and has to invalidate the cache line for the consumer's cache. The same effect applies for index prod_idx but also cons_idx as the consumer reads prod_idx before reading data to know if new data is available and the producer reads cons_idx before producing data to know if a new slot is available for writing in the array (see figure 3(c) and 3(d)).

The previous algorithm is the perfect example of what not to do in terms of cache consistency system handling. To avoid invalidations for the producer, the consumer must avoid reading a data if the cache line holding it might change. For the same reason prod_idx and cons_idx must not be read too often because their value change a lot. These two indices should also be in dedicated cache lines to avoid invalidation of their value when some data stored nearby in memory are modified.

BatchQueue To fulfill these requirements the synchronization between producer and consumer must be decoupled from their own progression and the buffer must not be accessed by both participant at the same time. This can be achieved with a sequential access to the shared buffer, with an additional flag *status* to make the synchronization. A value change of this flag indicates that one of the protagonists, the producer or the consumer, has just finished accessing the shared buffer. Setting the flag means the buffer is full, unsetting the flag means it is empty.

The algorithm presented above respects all the rules to avoid cache invalidation but is yet not adapted to our needs. Indeed, the producer and the consumer work in turn which can lead to substantial pauses for the producer. The solution is to divide the shared buffer in two parts, each of them managed by the algorithm above in opposition. At any time the producer produces in one part while the consumer consumes from the other part. The producer and consumer exchange their part when both the consumer has finished reading its part and the producer has finished producing in its part of the shared buffer (see figure 4).

For the *status* flag it means it is set (figure 4(b)) and unset (figure 4(c)) immediately after for each buffer exchange. Indeed, both the producer and the consumer have to notify one event and wait for another one. The producer notifies the consumer when its buffer is full and waits until the buffer of the consumer is empty. The consumer does just the reverse. It gives the following algorithm (figure 5 for the code and figure 4 for the workflow):

```
1  void enqueue(int data) {
2    tab[prod_idx++] = data;
3    prod_idx %= 2*N;
4    if (prod_idx % N == 0) {
5      while (status);
6      status = true;
7    }
8  }
9
10 void batch_dequeue(int copy_buf[]) {
11   int i;
12   while (!status);
13   for (i = 0; i < cons_idx + N; i++)
14     copy_buf[i] = tab[i];
15   cons_idx = (cons_idx + N) % (2*N);
16   status = false;
17 }
```

Figure 5. BatchQueue algorithm

Informal proof The algorithm does not deadlock because the producer and the consumer can't be blocked at the same time. Only lines 5 and 12 are blocking. If the producer is blocked on line 5, it means status is false and thus consumer can't be blocked on line 12 which leads eventually the consumer to execute line 12, set status to false and unblock the producer. The same way, if the consumer is blocked on line 12 it means the producer can pass through line 5 and execute line 6 which unblock the consumer.

Safety, on the contrary, is quite difficult to figure out. Safety is ensured with the busy-wait at lines 5 and 12. Each time the producer finishes producing in its buffer, it waits for *status* to be false and only the consumer can set it to false when it finishes consuming its buffer. Symmetrically, the consumer can't start reading in the buffer as long as *status* is false and only the producer can switch *status* from false to true. As the initial configuration is safe — the consumer is waiting for the producer to fill up its buffer to consume a new one — the safety is verified.

3. Performances

The previous section shows what are the drawbacks of Lamport's CLF queue and how BatchQueue addresses them. This section study what are the effects of the design of BatchQueue on performance. In other words, this section tells whether the choices made in the design of BatchQueue are relevant by quantifying the speedup against several other works.

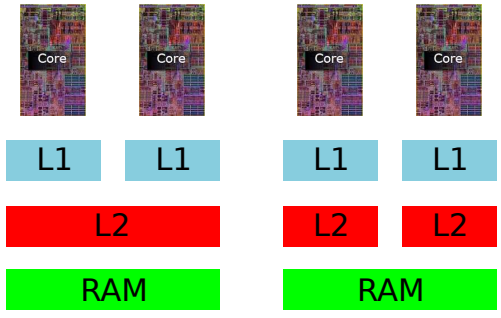
Communication techniques To quantify the efficiency of our inter-core communication algorithm, three other legacy techniques are compared against it. The first one is the pipe system call provided on POSIX systems. It is by far the slowest one because it uses read and write on a pipe to communicate, and read and write are system calls taking million of cycles to communicate a few bytes. POSIX pipes are also the simplest to use: they just need one system call available natively on all POSIX systems. This technique is used as a measurement basis for the other techniques. The goal is to show the interest of using a separate library — whether Lamport's CLF queue, BatchQueue or any other queue — over the natively available POSIX pipe technique. It shows that despite what is written in section 2.1, shared memory communication is still much faster than any other technique. Only a direct communication between cores could be faster[2] but current stock hardware does not provide such communication.

The second technique is the basic Lamport's CLF queue presented in figure 2. The consumer and the producer first verify they can consume or produce data by reading a shared index and then consume or read the data and increment their index. If the consumer and the producer run at the same speed, it generates a cache invalidation at each write of the producer and a remote read at each read of the consumer.

The last technique (Lamport DB) is a variation using some delayed buffering of the Lamport's CLF queue where the consumer and producer both have a local copy of the index of the other participant. The consumer test whether it can read data or not via a local copy of *prod_idx* and the producer does the same with a local copy of *cons_idx*. When a participant's index reaches the local copy of the other participant, the local copy is actualized with the shared index of the other participant. It leads to one more instruction but should perform better in case the consumer and producer works with burst. Indeed, suppose the producer has a write burst and is now 5 writes forward compared to the consumer. Then, the consumer copy the the new *prod_idx* in a local variable and read 5 pieces of data without reading *prod_idx*. During that time, *prod_idx* can evolve without needing any invalidation.

[2]Like what is available on Tera-scale [13]

Experimental platform The test is run on a bi-quad core with Intel X5472 processors running on a Linux 2.6.31 kernel with perfctr patch. Each test is run 10 times, and only the computed average is shown on the histographs. The metrics measured are time, cycles, cache hits and cache misses. Each test is run on two hardware configurations (see figure 6): with (a) or without (b) a shared cache between producer and consumer. X5472 processors provide both configurations as it's composed of two pairs of cores sharing a L2 cache, where pairs does not share any cache between them. Thus both configurations can be tested by affecting producer and consumer on the right cores. Bars in red are the tests run on a configuration with a L2 shared cache while bars in green only share the central memory.

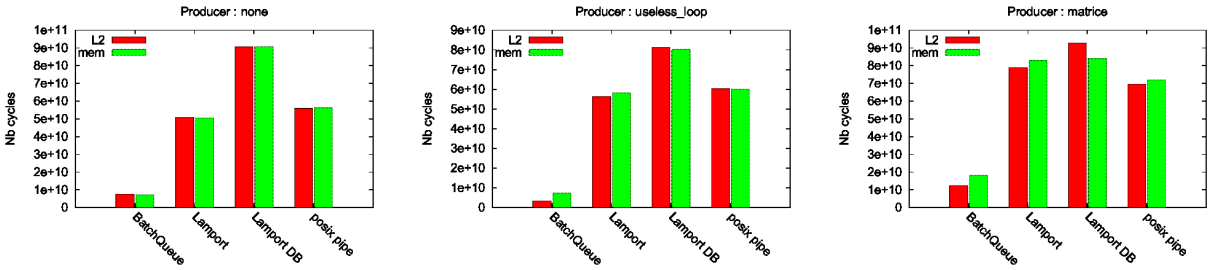


(a) Shared cache (b) No shared cache

Figure 6. Hardware configurations

Three kinds of producer behavior are considered: a producer that does nothing but sending always the same data to the consumer, a producer which does a useless loop before each write and a producer that makes a matrix multiplication before each write. These three different producer behaviors represent three different use cases. The first one corresponds to applications which mostly write, the second one to applications which perform only a few writes but does not use cache very much and the last one represents applications which make a big use of cache and perform a few writes from time to time. The test consists in sending 160 million word-sized data in the smallest time. The results in terms of cycles and time are shown in figure 7.

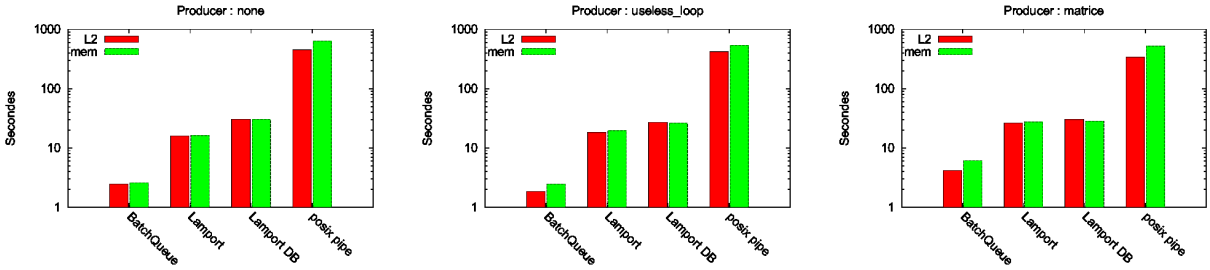
Results What first appears on the results is that our inter-core communication algorithm is much faster than the Lamport based techniques, 6 to 10 times faster. The Lamport's CLF technique is second, which suggest that its variant is only interesting in border cases, when production throughput vary quickly. It means Lamport's CLF queue is preferable over the simple delayed buffering optimization in most cases, as it provides better performance in all tested cases. It is important to notice that the efficiency of the pipe method can't be measured from the number of cycles. Indeed, as



(a) Total number of cycles needed for a producer mostly sending data

(b) Total number of cycles needed for a producer both computing and sending data

(c) Total number of cycles needed for a producer mostly computing



(d) Total time needed for a producer mostly sending data

(e) Total time needed for a producer both computing and sending data

(f) Total time needed for a producer mostly computing

Figure 7. Results for 160 millions writes in terms of cycles and time

it uses a system call, most cycles are not accounted for the producer but for some kernel thread because cycles measurements are done on a per-threads basis. Total time in this case is more accurate and it shows the pipe method is around 20 times slower than Lamport based techniques and up to 200 times slower than BatchQueue.

When reasoning in terms of time and cycle per sending, it gives the numbers from table 1. It must be reminded that the number of cycles accounted for posix pipe technique does not include the cycles spent in kernel space, hence the small number of cycles.

Techniques	Time (ns)	Cycles
BatchQueue	12.5	19
Lamport's CLF queue	93.8	313
Lamport DB's queue	188	500
Posix pipe	2810	344

Table 1. Results per sending for each technique in terms of cycles and time

With 18.8 cycles per sending, BatchQueue performs exceptionally well. Indeed, before each sending a function is called to do some computation related to the type of producer behavior chosen. As one function call is at least 5 instructions including call and ret which needs several cycles to complete, it means BatchQueue needs nearly 10 cycles per sending. It means BatchQueue can operate in pipeline parallelism at very fine grain, that is with stage as short as a few dozens of cycles.

Finally, the last result is about impact of cache sharing on communication performance. While others algorithms tend to be agnostic of the hardware configuration, at least in a relative comparison between two configurations, BatchQueue suffers of the situation: the extra time needed without a shared cache can reach one third of the time with a shared cache. However, the slowdown is not as severe as it seems since the absolute performance remains very low: in the worst case only 40.6ns and 125 cycles are needed to send one data.

4. Related work

The producer / consumer scheme is a very old and known problem so lots of work has been done on the subject, even in the more specific context of core to core communication. This section presents related work, from the simple single producer single consumer queue to more complex ones, also including multiple producer multiple consumer queues.

IPC Most of the time, communication between producer and consumer is done using inter-process communication (IPC). Among other, IPC provides anonymous FIFO, named fifo, message queue. Although convenient because they are high level and natively present, all these methods are very slow because they are provided by the kernel and need context switches. In consequences, these methods are mainly used for small communication and/or synchronization between processes.

Shared memory However, IPC also provides a way to have shared memory between two processes, like two threads natively have. Shared memory proves to be a highly

efficient way of communication between processes. It gives producers the ability to send data to consumers by just writing them in memory. The counterpart is that it does not provide any synchronization. Most implementation thus use software locks to ensure consistency of data.

Lamport's CLF queue Although the combination of shared memory and software locks is much better than other IPC alternatives, the solution remains costly. Yet, synchronization is not necessary in the single producer single consumer scheme. Lamport first proposed an algorithm to handle this case requiring no synchronization [6]. The idea is that conservatism upon indices of producer and consumer only lead to unused entry in the buffer. By comparing both indices, the consumer can know if the producer is ready to write in the next entry and the producer can know if the consumer has read the next entry. This algorithm is much simpler than BatchQueue but take no care of the underlying cache consistency system. Thus, when producer and consumer are highly concurrent, many cache invalidation and remote read are needed. On the other hand, with poor concurrency producer and consumer have to wait each other much more.

MPMC queues A number of works extends the lock-free ability to queues with multiple producers and multiple consumers, either in static configurations [14, 15] or dynamic configurations [15, 3, 5, 4, 8, 11, 12, 9, 10]. These works improves over Lamport SPSC queue by allowing more complicated communication to benefit from the speedup of lock-free algorithms on architectures with many cores or processors. Static configurations allow better locality of data and less memory overhead, while dynamic configurations, based on linked list, provides a more flexible environment. The counterpart of these solutions is more complexity: algorithms are more complex, rely on some atomic operations such as compare-and-swap and thus are slower. Furthermore, as Lamport SPSC queue these algorithms does not take special care of cache consistency system.

FastForward [16, 7, 2, 17] address the problem by focusing on the cache consistency system. Among all these algorithms, FastForward [2] is the simplest. The solution adopted to prevent cache line trashing is to reserve one of the value the data can hold to indicate when an entry is empty. That way, consumer and producer only need to manipulate their own algorithm and read the value inside the next entry to consume from or produce into.

This elegant approach is still vulnerable to cache line trashing if producer and consumer work on entries fitting in the same cache line. Authors suggest to address this concern by using temporal slipping so that there is always one cache line between entries manipulated by the producer and the consumer. Temporal slipping can be enforce by spinning until the distance between producer and consumer indices is sufficient. Thus, FastForward works only with relatively bounded stage duration since big variation in stage duration means to watch the distance between producer and consumer more often which causes cache line trashing because of changes in indices, which the algorithm precisely try to avoid. BatchQueue solve this problem by relaxing temporal slipping: delayed buffering with an extra variable is sufficient to ensure producer and consumer are working on separate cache line only once per cache line produced (respectively consumed).

DBLS and MCRingBuffer DBLS [16] and MCRingBuffer [7] takes the approach to delay sending data, until N cache lines are filled, with $N \geq 1$. Delayed buffering allows to reach the optimal ratio of one cache line trashing per cache line sent. Cache line trashing due to index changes are also avoided by shadowing them. Producer and consumer have a local copy of both shared indices: a "working copy" for their own progression — the read index in the case of the consumer — and a "shadow copy" to mirror the progression of their counterpart. Working copies are updated each time a data is produced (respectively consumed) and after N lines filled (respectively consumed), shared variables are updated from the working copies.

The difference between DBLS and MCRingBuffer resides in the update frequency of the shadow copies: DBLS update shadow copies each time N cache lines have been filled (respectively consumed) while MCRingBuffer only update them when no progress can be done without updating them. Furthermore, authors of MCRingBuffer makes clear that producer and consumer variables should be in separate cache lines. BatchQueue differentiate from DBLS and MCRingBuffer by its memory overhead: BatchQueue only need one extra bit for synchronization when DBLS and MCRingBuffer require 4 more variables.

Clustered software queue Clustered software queue [17] (CSQ) is the closest algorithm to BatchQueue and the most elegant with FastForward. As BatchQueue, CSQ has a low memory overhead, although it depends on one of the algorithm parameters. As BatchQueue, CSQ also uses delayed buffering to lower cache line trashing and uses a bit flip to indicate when a buffer has been filled or emptied. Both algorithms share the same principle but yet they differ in two points. First, CSQ employs two level of buffers: data is stored in several second level buffers pointed to by a single first level buffer. Second, CSQ uses one bit per second level buffer when BatchQueue only needs one for the two part of its buffer.

These two differences has two effects on performance. First, indirection caused by the two levels of buffer causes one or few more cycles to be spent depending on the ability for the first level buffer to stay in the L1 cache or not. Second, each second level buffer extra bit slightly increases the pressure on the cache since for better performance each of these bits should be placed in separate cache lines. So the bigger the first level buffer is, the slower the algorithm will be: bigger first level buffer meaning more probability that it does not stay long enough in the L1 cache and the more pressure applied to the cache by extra bits.

From the comparison with alternatives queues it appears BatchQueue matches all its goals. It successfully achieves

to prevent cache line trashing with a small memory usage overhead. As this is achieved by loosing some flexibility between the consumer and the producer, BatchQueue is targeted for very communicative systems with high usage of L1 cache and relaxed latency requirements.

5. Use case

The previous section shows that BatchQueue is very fast even when memory is very solicited. Thus, BatchQueue is suitable for many kind of intensively communicative applications. Complex monitoring, for instance, is a system which could benefit from using BatchQueue. Indeed, monitoring is usually realized by the application itself which slows down the application depending on the complexity of the task it performs. This section presents a monitoring system using BatchQueue to perform complex monitoring without a big loss in performance. Then, the section presents one of the possible usage of this monitoring system which motivated the conception of BatchQueue: object graph visualization.

5.1. Monitoring system

Monitoring an event requires an action to be performed each time the event occurs. This action can be constituted by very few instructions if the goal is to count the number of occurences of an event. But quite a lot instructions are needed if the goal is to log on disk the events or trigger some tasks as in an intrusion detection system. For some combination of events and corresponding actions it is possible to let the hardware do all the work. This is the case for memory access event and counting action where special registers on processors accumulate the number of a given memory event automatically, each time these event occurs. When the event cannot be tracked by hardware or the action is too complex to be done in hardware, then the code must be instrumented.

Instrumenting a code means modifying the compiler to add the execution of an action each time the tracked event occurs. If the action is complex and requires a lot of instructions and/or if the event is frequent, then the overhead become huge. The overhead is given by the formula $freq * nb\_inst$ where:

- $freq$ is the frequency of the tracked event in instruction percentage with 100 % meaning the event tracked is the execution of an instruction ;
- nb_inst is the number of instructions required to perform the action.

With the constantly growing number of cores in modern computers, power computation becomes available as idle cores. In consequences, it becomes possible and interesting to perform the monitoring asynchronously on idle cores. Hence the utility of a fast core to core communication algorithm. If the monitoring is complex enough to be slower than sending data with BatchQueue, it becomes faster to send the data to one or more dedicated monitoring cores doing the monitoring computation concurrently with the execution of the application. Such an organization would form a generic monitoring platform whose architecture is detailed below.

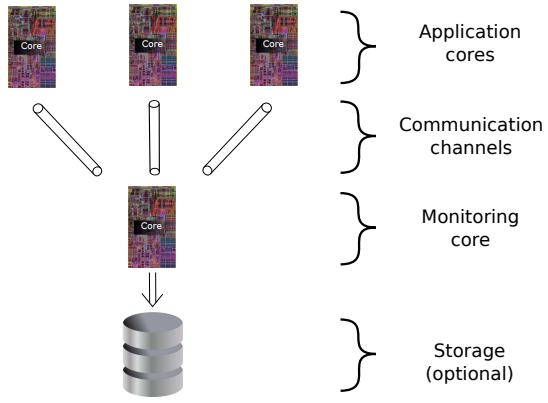


Figure 8. Monitoring platform architecture

The architecture adopted is to divide the monitoring platform into 2 components: communication channels provided by BatchQueue and a monitoring core. Their interactions are shown in figure 8. Monitored events are sent to the communication channels by instrumenting the code to call the enqueue function each time they happen. The communication channels send these informations from the application to the monitoring core which allows the monitoring system to be on a different core or processor than the application. The monitoring system then saves the result of processing the events on disk or can send it back to the application if it uses these informations to self-adapt.

5.2. Visualization tool

Programs behavior with regard to memory layout is a long studied field of research, in particular in the scope of NUMA systems. However, the knowledge gathered until now is expressed in terms of numbers: how much read and write accesses, which zones are accessed, and so on. Now, the monitoring system gives the opportunity to log precisely all memory events without too much overhead since recording the events on disk is done concurrently with the application studied. By recording all memory events related to pointers, the graph constituted by all pointers between objects, called the object graph, can be observed along with all the historic of its mutations. Then, offline analysis of the graph mutations record would allow to browse the object graph as it was at any moment of the application execution.

The goal of such work is to provide a tool to help debugging and optimizing programs thanks to dynamic analysis and the browsing of the object graph and its evolution over time. An analysis of the monitoring record could identify patterns in the object graph which may lead to suboptimal

memory access. In particular, on NUMA systems some layout patterns could identify all inter-node references and suggest a better layout for the data. The graph mutations record also provides the ability to create a complete back-in-time debugger.

The steps to build such tools would be the following. First, a visualization tool using object graph mutations record as input must be built. The visualization tool must provide basic video player features. The object graph must be graphically displayed and the tool must give the ability to see an animation of the graph mutation over time. We expect the animation to gives us the general aspect of the graph and trends in its evolution. A zoom function and a speed parameter may also be useful to control what the animation displays.

After some trends are identified with the visualization tool, they must be verified and measured precisely. Both these actions require to build an extraction tool to extract the desired information from the record. Then, a calibration phase of the tool to determine the threshold above which an optimization of the user program could be done. The threshold known, the extraction tool can be transformed in a fully automatic diagnostic tool which is the last step. At this stage, the extraction tool is able to automatically detect programs which can be improved and tells what must be improved.

6. Conclusion

This paper presents a new single producer single consumer (SPSC) queue with no synchronization called BatchQueue. Unlike Lamport's concurrent lock-free (CLF) queue, BatchQueue takes care of the underlying cache consistency system in its algorithm. BatchQueue is designed to avoid cache line invalidations and remote reads as much as possible. BatchQueue is also designed to have the smallest overhead in terms of memory usage compared to Lamport's CLF queue: only one extra bit is needed.

All these characteristics make BatchQueue a fast and memory-thrifty SPSC queue. On a 32-bit architecture, BatchQueue is able to send word-sized data in 12.5 to 40.6 nanoseconds and 19 to 125 cycles depending on the pressure on L1 cache and presence of a shared cache between producer and consumer cores. BatchQueue is therefore very well suited for very communicative tasks, monitoring for instance, as long as they can tolerate the small increased latency due to its batch processing principle.

References

[1] U. Drepper. What every programmer should know about memory, 2007.

[2] J. Giacomoni, T. Mosely, and M. Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *In PPoPP '08: Proceedings of the The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 2008.

[3] A. Gottlieb, B. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):189, 1983.

[4] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. *Principles of Distributed Systems*, pages 401–414, 2007.

[5] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. *Proceedings of Distributed Computing*, pages 117–131, 2004.

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

[7] P. Lee, T. Bu, and G. Chandranmenon. A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring. In *IPDPS '10: Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[8] J. Mellor-Crummey. Concurrent queues: Practical fetch-and-ϕ algorithms. Technical report, Technical Report 229, Computer Science Department, University of Rochester, 1987.

[9] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[10] 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*, page 262. ACM, 2005.

[11] S. Prakash, Y. Lee, and T. Johnson. Non-blocking algorithms for concurrent data structures. Technical report, University of Florida, 1991.

[12] S. Prakash, Y. Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, pages 548–559, 1994.

[13] Single-chip Cloud Computer. http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[14] 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*, page 143. ACM, 2001.

[15] J. Valois. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pages 64–69, 1994.

[16] C. Wang, H.-s. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Y. Zhang, K. Ootsu, T. Yokota, and T. Baba. Clustered Communication for Efficient Pipelined Multithreading on Commodity MCPs. *IAENG International Journal of Computer Science*, 36, 2009.