Experimental Comparison of Emulated Lock-free vs. Fine-grain Locked Data Structures on the Cray XMT

Rob Farber
Pacific Northwest National Laboratory
David Mizell
Cray, Inc.
Dec. 15, 2009

Abstract

Three implementations of a concurrently-updateable linked list were compared, one that emulates a lock-free approach based on a compare-and-swap instruction, one that makes direct use of the Cray XMT's full-empty synchronization bits on every word of memory, and a third that uses the XMT's atomic int_fetch_add instruction. The relative performance of the three implementations was experimentally compared on a 512-processor XMT. The direct implementation approach performed up to twice as fast as the other two approaches under conditions of low contention, but the three implementations performed about the same when the amount of contention was high.

The Cray XMT

The Cray XMT is a heavily multithreaded architecture, with 128 hardware threads per processor. This degree of multithreading provides a significant amount of latency tolerance, to the point that a global shared memory is practical. The 512-processor XMT prototype now in Cray's testing laboratory provides 4 terabytes of globally-addressed, shared memory. Shared memory of this size together with the XMT's unique degree of latency tolerance gives the XMT outstanding performance on applications that lack any processor affinity or spatial locality for a parallel applications programmer to exploit. In particular, the XMT can provide better scaling and performance than its distributed memory/commodity processor competition for commonly used graph analysis kernels [9]. This makes it a promising parallel system for applications, such as social network analysis, that have to process huge, graph-oriented data structures.

For this class of applications, holding large, complex data structures in main memory appears to be inevitable. Performance requirements coupled with the presence of many thousands of hardware and software threads on the XMT dictate that these data structures be concurrently accessible and updateable, without becoming scaling bottlenecks. Accordingly, it is attractive to consider using lock-free implementations of shared data structures. Lock-free data structures have the potential for high performance because of their ability to be concurrently updated and accessed. Furthermore, there is a significant software productivity consideration: they are already designed and written, and in many cases tested, for a large number of commonly-used data structures [11, 12].

There has been a great deal of research over the past 20 years on lock-free data structures and approaches to their implementation [2, 6, 7, 10]. A few of these have focused on performance evaluation [1, 3, 4], as does this paper. The attractiveness of using lock-free data

structures stems from their support of fine-grain, concurrent updates and their characteristic of being non-blocking: the failure of one thread doesn't affect the ability of other threads to progress. This is contrasted with a blocking style of synchronization in just this way: suppose a thread fails while it is holding a lock. Any other threads waiting for that lock to be released will not make progress. At least on the XMT architecture, however, robustness in the face of a thread failure of this sort isn't a significant issue, because failures will occur, or at least will be detected, at the processor/node level. The user is certain to have larger problems than some threads being blocked; 128 threads and 8 gigabytes of memory will be gone.

The high parallel computational throughput potential of fine-grain accesses and updates is very attractive, however. The problem for the Cray XMT is that most lock-free data structure implementations employ the atomic compare-and-swap (CAS) operation, which the XMT instruction set does not have. Instead, the XMT provides ubiquitous, fine-grain, blocking synchronization, in the form of an additional bit, called the full-empty bit, on every 64-bit word of memory. The instruction set includes specialized load and store and instructions, such as readfe() ("read full-empty"), the semantics of which can be thought of as follows:

```
readfe(word):
{ block until word is marked full
  load contents and mark word empty
}
```

Symmetrically, the writeef() ("write empty-full") instruction blocks until a word is marked empty, then stores a value into the word and marks it full.

Additionally, the Cray XMT instruction set includes an atomic int_fetch_add() operation, the semantics of which are as follows:

int_fetch_add(word, value) = do these two steps atomically:

- Add "value" to the contents of "word."
- Return the previous contents of "word."

This suggests three possible approaches to implementing data structures with fine-grain, concurrent updates on the XMT:

- Emulate CAS using full-empty bits, and implement a "classical," CAS-based data structure from the lock-free research literature.
- Directly implement a concurrently-updateable data structure with blocking, but fine-grain synchronization, using the full-empty bits and the blocking read and write instructions.
- Implement fine-grain synchronization on data structure components by including a synchronization integer with each component. Perform an int_fetch_add on the synchronization integer, with an add value of 1. Initialize each sync integer to 0. If the returned value is 0, then the attempt to lock the component was successful. Otherwise, perform another int_fetch_add with a -1 value to return the sync integer to its previous state, and try again later.

In this experiment, we implemented a linked list using each of these approaches and compared their performance.

For the emulated CAS instruction, we implemented a software compare-and-swap, using full-empty bits to guarantee atomicity. The C++ implementation is shown below.

```
typedef uint64_t word;

bool CAS( word* x, word match,
word replace )
{
    word temp = readfe(x);
    //lock x (mark it empty)
    if( temp == match ) {
      writeef(x, replace);
      //unlock
      return true;
    } else {
```

```
      writeef(x, temp);
      return false;
      }
}
```

For the CAS-based linked list we implemented John Valois's linked list design from [8].Valois's approach uses two types of nodes: "normal" nodes that contain data, and an equal number of "auxiliary" nodes, one between every two normal nodes in the list. The auxiliary nodes are used to avoid the correctness problems CAS-based approaches can get into when two or more updates are occurring very near each other. Following the deletion of a normal node, there may be more than one auxiliary node in the linked list between two normal nodes, but these are cleaned up during subsequent searches of the list. Valois's design assumes that the data items in the linked list (we used non-negative integers) are maintained in increasing order.

To insert a new node into the list or delete one from it, Valois's design starts at the beginning of the list and searches it in order, until either the relevant value is found, or a value larger than it is found. To carry out the search, Valois uses a data structure he calls a "cursor," which consists of three pointers: one ("target") that points to a normal node being visited, one ("pre_aux") that points to an auxiliary node proceeding this normal node, and one ("pre_cell") that points to the previous normal node. As the cursor is stepped forward through the list, a procedure named "update" takes care of deleting any extra auxiliary nodes that appear between two normal nodes. Valois's "Insert" procedure repeatedly calls "TryInsert," a procedure containing a CAS operation, to perform the insert when conditions are right – that is, when the CAS determines that the relevant node pointer has not changed since this thread last accessed it. The "Delete" procedure similarly spins calling a "TryDelete" procedure that contains a CAS, until conditions are right.

We did not implement two parts of the Valois design. One consisted of two procedures, SafeRead and Release, which manages the memory occupied by deleted nodes, in such a way to avoid the version of the "ABA problem" that occurs when the same section of memory is re-cycled by the OS's memory manager and deceptively appears to the application program never to have been changed. We did not free deleted nodes to the memory manager in either implementation, so the performance comparisons are independent of that aspect. This simplified the code considerably, because we didn't have to worry about freeing a node that still had some cursor pointing to it. This also allowed us to skate around the critical race problem in Valois's original design, addressed by Michael and Scott in [5]. The other unimplemented component was a fairly complex mechanism within TryDelete which uses backpointers to search for any extra auxiliary nodes resulting from a normal node's deletion. Since the next cursor in this section of the list accomplishes this anyway, we left this part out of the implementation of TryDelete.

In the implementation that directly uses full-empty synchronization, we still followed the top-level structure of the Valois design fairly closely. An insert entails a search from the start of the list for the place to insert the new node, and a delete starts with a search for the node to be deleted. One difference is that the direct full-empty implementation did not need the auxiliary nodes. This saves about 33% of the memory consumed by the CAS-based linked list – but note that this fraction would be much smaller in a realistic application in which "normal" nodes contained nontrivial amounts of data. It also affected overall performance, as is explained below. In the full-empty implementation, a two-element cursor data structure points at the current node of interest and its immediate predecessor as the list is searched.

Rather than spinning on the CAS procedure, the direct implementation of delete uses a

procedure named "Reserve3," which uses readfe() to mark a word empty in three nodes: the node to be deleted, the previous node, and the successive node. Insert calls "Reserve2," which similarly locks each of the two nodes that are on either side of where the new node is going to be inserted. To avoid deadlock, the readfe() locks are done in increasing order in the list. The word in each node that the readfe() is performed on is a "valid" flag, which is set to 0 while the node is in use and to 1 when it is deleted. If either of the Reserve procedures finds an "invalid" node, it unlocks anything it has locked and returns information about which node was invalid. If it was the lowest node in the set being locked, the program starts the search over from the beginning of the list – it has no better information about an appropriate place to restart. If, however, the first node in the set is still valid, the program resumes the search from there.

The Test Harness

All three versions were implemented in C++ on the XMT, and compiled with Cray's specialized, parallelizing XMT compiler. We wrote a main program that creates a linked list of "LISTSIZE" nodes and then performs "NUMTRIES" updates on it. The data values in the nodes are initialized to be increasing even integers between 0 and LISTSIZE x 2. Building and initializing the list and the random number arrays is untimed; only the update phase is timed for the experiment. In the first phase of the update operations, the update loop randomly chooses, with 50% probability, either to delete a node containing an even integer or to insert a node containing an odd integer. This guaranteed that, for openers, a lot of inserts as well as deletes would happen. Preliminary experiments determined that after about LISTSIZE x 2 iterations of this loop, the list contained about 50% even and 50% odd values. So for subsequent updates, a 50-50 mix of inserts and deletes were performed using a random array of even and odd integers from 0 to LISTSIZE x 2.

When generating the arrays of random integers to be inserted or deleted from the linked list, we used a procedure called "focus" to artificially create update "hot spots." Focus took three integer parameters, each representing a percentage value between 0 and 100. The first parameter represented the fraction of all the random integers that were to be generated within the hot spot range. The second and third parameters were, respectively, the bottom and top percentiles of the hot spot range. So, for example, a call to focus() with the parameters of 80, 90, and 100 would generate, for a list of length 10,000, an array of random integers, 80% of which fell within the highest 1000 of the linked list nodes, while the other 20% were uniformly distributed among the values held in the other 9000 of the nodes. Giving focus() the parameters of 100, 0 and 100 would result in the random integers being distributed uniformly across the whole list – i.e., without a hot spot.

Results

All experiments were performed on a linked list initialized to hold 10,000 nodes (10,000 "normal" nodes in the emulated CAS case), and each experimental run attempted 100,000 updates on the list (since the update values were randomly generated, it was possible that an iteration could try to insert a node that was already there, or delete a node that wasn't there – these attempts just ended up as searches with no update). Table 1 shows the performance results for the focus(100, 0, 100) case, i.e., when updates and deletes were uniformly distributed across the whole list. Times are in seconds. Using the XMT's hardware instrumentation, we also collected the cumulative number of memory requests for each update loop, shown in the table in billions. A dittoed entry signifies that the number didn't change significantly from the previous run. The cumulative number of memory retries, also collected by built-in hardware instrumentation, is shown in the last two columns. We only recorded them for larger numbers of processors, because they were negligible for the

others. Memory retries happen when a thread tries to perform a readfe() and the word turns out to be empty, i.e. the calling thread is blocked. Invisibly to the application programmer, the hardware spins on the readfe(), retrying it 1000 times before trapping and notifying the XMT runtime software. The runtime removes this thread from the processor registers at this point, placing it into a queue from which it can be resumed once the word is set to full. We also measured these traps, but they were negligible for all but the highest-contention cases, and even then were only around 10,000.

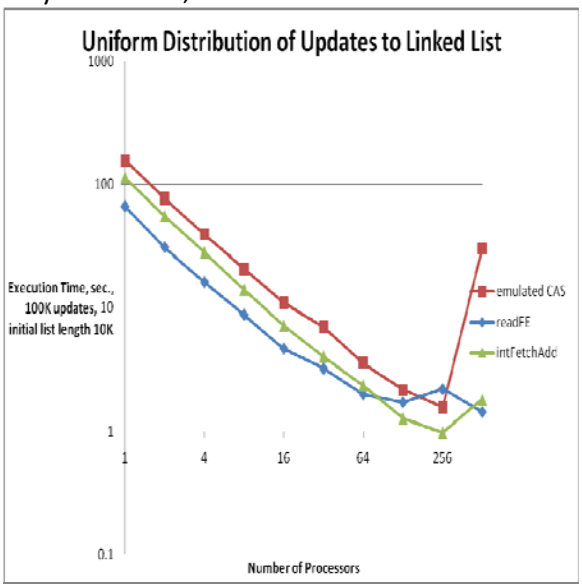

Figure 1: Graph of the performance results from Table 1

One can see that all three implementations appear to scale pretty close to linearly, with the exception of the 30.14-sec. execution time for the emulated CAS run on 512 processors. We believe this to be an anomaly caused by some exogenous factor in the system – the first XMT to be built with 512 processors, which is still undergoing intensive testing in our laboratory. We re-ran this configuration later and got an elapsed time of 62 seconds, which supports the supposition that something exogenous is happening.

Table 2 shows the results when 90% of the updates were focused on the middle of the list, from the 45th percentile up to the 55th percentile. Table 3 shows the results when 95% of the updates were focused on 5% of the list, and Table 4 shows the results when 95% of the updates were concentrated on 1% of the list.

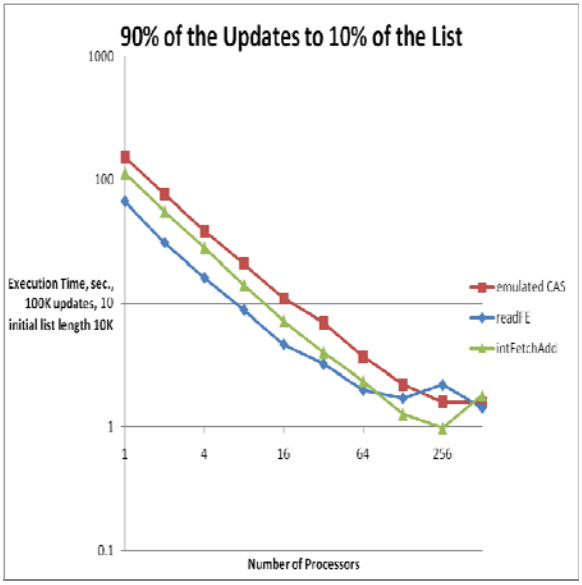

Figure 2: Graph of the performance results shown in Table 2

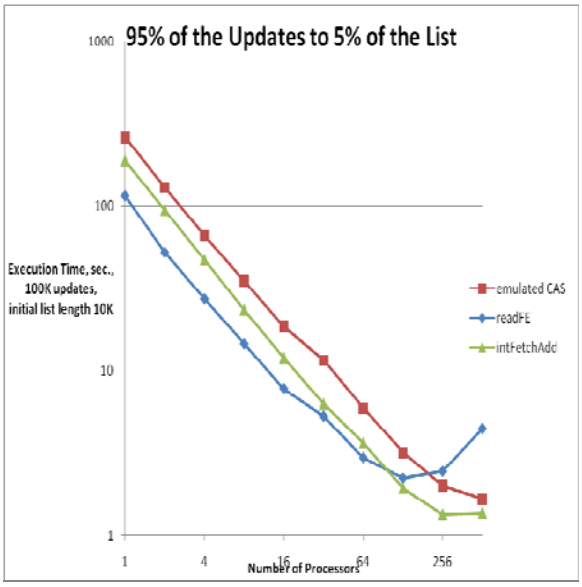

Figure 3: Graph of the performance results shown in Table 3

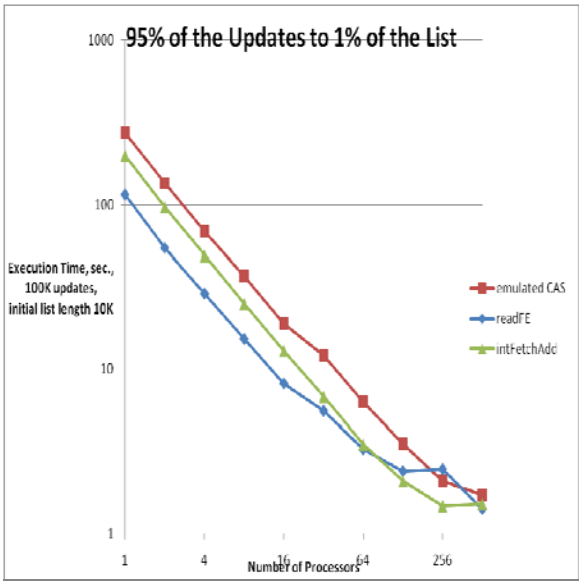

Figure 4: Graph of the performance results shown in Table 4

Analysis

All three implementations scale roughly linearly until the number of processors approaches 512. All three were also fairly robust under increasing levels of contention, until we reach the higher levels of contention, shown in Tables 3 and 4. It seems clear that when contention is low, the full-empty bit implementation outperforms the CAS implementation by a factor a little above two, while the int_fetch_add implementation lands between the other two. This is probably because searches in the CAS-oriented approach go across twice as many nodes, since this implementation employs auxiliary nodes. Additionally, the CAS approach updates three pointers every time the cursor moves, as opposed to two pointers in the full-empty approach. Furthermore, both the CAS emulation approach and the int_fetch_add approach have to "undo" what they have done if their attempt to gain control of a synchronized component has failed. On a latency-tolerant, highly multithreaded architecture like the XMT, memory bandwidth is often the limiting factor for performance, and one can see from the memory request columns in the tables that the CAS approach accesses memory roughly three times more often than the direct full-empty approach, and the int_fetch_add approach roughly twice as often.

As contention increases, however, either from increasing the number of processors (which increases the number of threads concurrently performing updates) or from tightening the focus, the performance difference between the three implementations shrinks significantly. The explanation appears to have to do with the number of retries, which approach parity as contention increases (except for the anomalous case, which should be ignored). The measured number of retry-induced traps was down in the noise for every run, but one can see that the number of retries increases significantly with the amount of contention.

Conclusion

Thus, it seems to be a matter of how much contention, or hot-spotting, the user expects, that would determine the choice of concurrent implementation, at least for this linked list data structure. If a true, atomic compare-and-swap instruction were to be implemented for some future version of the XMT, we would expect that the full-empty bit implementation would still dominate for the low-contention case, simply because it performs fewer memory accesses, but the CAS implementation would probably be distinctly superior under heavy contention.

Programming Effort

Programmer productivity has become a significant issue in the high-performance computing community, and papers that describe software implementations ought to say something about how long they took, however small the sample size and however unscientific the estimate may be.

Implementing the Valois CAS approach took the second author about two days to write in C++ and two more days to debug. All of the bugs were a function of lack of familiarity with the

esoteric profundities of C++ syntax. Once the sequential version ran correctly, so did the parallel version – for which the credit goes to John Valois.

The full-empty version took about a day to write, the first time, and two days, with a colleague's help, to debug its parallel version enough to realize that it contained a critical race, and that the synchronization approach had to be completely rewritten. The rewrite took a day, and ran correctly immediately.

Acknowledgments
This research was funded by the DOD via the CASS-MT program at Pacific Northwest National Laboratory.

A note of thanks goes to Kristi Maschhoff, for discussing this research with us and helping debug the full-empty-based implementation.

Thanks also go to Gail Alverson and the Cray XMT software development team, for allocating some experimentation time on "Nemo," the 512-processor XMT prototype now in Cray's testing laboratory.

References

[1] Guojing Cong and David Bader. Lock-free parallel algorithms: An experimental study. In *Proceedings of the 11th International Conference on High Performance Computing.* 2004.

[2] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15:745–770, 1993.

[3] Theodore Johnson. Characterizing the performance of algorithms for lock-free objects. *IEEE Transactions on Computers*, 23:200–2, 1995.

[4] Anthony Lamarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC*, pages 130–140. ACM Press, 1994.

[5] Maged Michael and Michael Scott. Correction of a memory management method for lock-free data structures. Technical Report, University of Rochester, 1995.

[6] Håkan Sundell, Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, page 84. IEEE press, 2003.

[7] John Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems.* Las Vegas, NV, 1994.

[8] John Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing.* 1995.

[9] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak, An Experimental Study of A Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances, *Workshop on Algorithm Engineering and Experiments (ALENEX)*, New Orleans, LA, January 6, 2007.

[10] Keir Fraser and Tim Harris , Concurrent programming without locks , *ACM Transactions on Computer Systems*, Vol. 25 (2), May 2007

[11] Cambridge Lock Free Library, http://www.cl.cam.ac.uk/research/srg/netos/lock-free/

[12] Parallel mode in GNU libstdc++, http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html

Appendix

				memory requests (in B)			retries (in B)		
#processors	emulated CAS	readFE	intfetchadd	em. CAS	readFE	intfetchadd	em. CAS	readFE	intfetchadd
1	156.77	65.47	110.23	9.3	3.1	4.1			
2	77.1	31.06	55.83	“	“	4.1			
4	39.42	16.18	28.11	“	“	“			
8	20.67	8.8	13.96	“	“	“			
16	10.97	4.64	7.17	“	“	“			
32	6.97	3.24	3.75	“	“	“	0.241	0.118	
64	3.56	1.63	2.26	“	“	“	0.473	0.38	
128	2.17	1.69	1.26	“	“	4.2	1.7	1.26	1.4
256	1.57	2.48	1.3	“	3.5	4.6	4.9	5.5	6.5
512	30.14	1.59	2	28.9	4.7	5.7	152.7	19.9	22.2

Table 1: Experimental results for updates distributed uniformly across the entire list

				memory requests (in B)			retries (in B)		
#processors	emulated CAS	readFE	intfetchadd	em. CAS	readFE	intfetchadd	em. CAS	readFE	intfetchadd
1	153.21	66.33	112.89	9.3	3.1	4.1			
2	76.21	30.91	55.78	“	“	“			
4	38.75	16.17	28.12	“	“	“			
8	20.87	8.81	14.05	“	“	“			
16	10.93	4.67	7.21	“	“	“			
32	7.01	3.22	4.02	“	“	“	0.241	0.118	
64	3.7	2	2.37	“	“	“	0.661	0.449	
128	2.21	1.72	1.27	“	“	“	1.8	1.3	1.3
256	1.59	2.2	0.98	“	“	4.5	4.9	3.8	3.9
512	1.58	1.43	1.81	10	4.4	5.6	13.2	12.2	15.7

Table 2: 90% of updates focused on the 45th to 55th percentile of the list

				memory requests (in B)			retries (in B)		
#processors	emulated CAS	readFE	intfetchadd	em. CAS	readFE	intfetchadd	em. CAS	readFE	intfetchadd
1	263.2	116.2	189.31	15.7	5.3	6.2			
2	129.89	52.53	95.21	“	“	7			
4	66.43	27.41	47.26	“	“	“			
8	35.28	14.64	23.55	“	“	“			
16	18.65	7.8	12.01	“	“	“			
32	11.57	5.3	6.29	“	“	“	0.247	0.156	
64	5.94	2.97	3.63	“	“	“	0.605	0.434	0.6
128	3.19	2.22	1.94	“	“	7.2	1.6	1.4	1.5
256	2.01	2.45	1.34	“	“	7.4	4.1	3.4	3.9
512	1.67	4.46	1.37	16.5	6.6	8.2	12.9	14.3	13.3

Table 3: 95% of the updates focused on the 85th to 90th percentile of the list

				memory requests (in B)			retries (in B)		
#processors	emulated CAS	readFE	intfetchadd	em. CAS	readFE	intfetchadd	em. CAS	readFE	intfetchadd
1	275.14	115.09	199.02	16.3	5.5	7.3			
2	135.55	54.47	97.3	“	“	“			
4	68.63	28.95	49.3	“	“	“			
8	36.91	15.31	25.07	“	“	“			
16	19.07	8.18	12.99	“	“	“			
32	12.15	5.58	6.81	“	“	“	0.385	0.181	
64	6.33	3.27	3.5	“	“	7.4	0.673	0.59	0.49
128	3.53	2.41	2.1	“	“	7.5	1.7	1.3	1.3
256	2.1	2.48	1.48	“	“	7.8	4.5	3.2	3.8
512	1.73	1.41	1.53	17	6.7	8.5	12.7	12.6	12.3

Table 4: 95% of the updates focused on the 90th to 91st percentile of the list