Fast Distributed Mutual Exclusion

Sean Franey

Abstract

A technique is proposed for quickly distributing mutexes in a system of multiple nodes. Evaluated in the context of a physical network, it is relatively agnostic with respect to the underlying topology and can be modified to work with more abstract arrangements of mutex repositories (e.g. distributed databases). To prove its usefulness, it is compared to a variety of other proposals for distributing mutual exclusion from the operating systems and database realms and found to provide significant performance and flexibility benefit. Additionally, a method is presented for applying the proposal to atomic operations in GPGPU applications to allow the GPGPU space to efficiently support a broader range of data parallel applications.

1. Introduction

Mutual exclusion is a desirable property utilized to varying extents in many computing applications. While specific goals vary, at its core mutual exclusion ensures atomic (i.e. indivisible) update of a shared resource. Many uses exist, but one that motivated our desire to make it fast is Atomic Coherence [3]. Atomic Coherence utilizes mutual exclusion to simplify cache coherence protocols by requiring a processor to acquire a mutex (i.e. token) prior to initiating a coherence request. By making requests mutually exclusive, multiple processors are prevented from initiating possibly conflicting requests simultaneously and the cache coherence

controller can be greatly simplified. If simplification in itself is not enough, the reduced complexity allows optimizations to be incorporated into the protocol much more easily (adding an 'F' state, for example). As shown in [3], these optimizations can then potentially overcome the latency associated with acquiring the mutex in the first place, leading to a solution that is both more simple and faster.

Unfortunately for anyone wishing to apply Atomic Coherence to a near-future processor, the current proposal leverages an optical interconnect to allow mutexes to be quickly acquired and released. While latency-sensitivity evaluations were performed on Atomic Coherence, it becomes understandably much less desirable as the latency cost of acquiring a mutex increases to electrical interconnect levels because any proposed optimizations must then have a greater impact to make up for the cost.

If Atomic Coherence weren't enough however, many other situations exist where mutual exclusion is desirable or even necessary, such as the critical section of multi-threaded code. Unfortunately, the traditionally high cost of acquiring a mutex has led to a severe degree of avoidance and engineering of applications to not use mutual exclusion where it would otherwise be beneficial. In fact, the high cost of mutex acquisition has even led to proposals to speculatively ignore the mutex requirement and only recover when a race is detected (e.g. Lock Elision [11] and Token Coherence [12]). With this historical avoidance of mutual exclusion, there certainly exist many applications that in the past have gone to great lengths to avoid or speculate past locks and mutexes that could be greatly simplified and reasoned about if a low-cost method of acquiring a mutex was available. As such, we propose an implementation that can be logically laid on top of existing interconnection networks that can greatly improve the speed at which a system can distribute mutexes for any purpose.

One such area where mutual exclusion has been fiercely avoided that we believe has much to gain is in the realm of atomic operations in GPGPU. We will outline a study of application of our mutual exclusion scheme related to GPGPU atomics in a later section, but broadly, atomics are vitally important operations in many parallel applications that are currently a very costly operation – in terms of latency – in GPGPU. As such, this greatly reduces the appeal of GPGPU in certain realms (generally the increasingly prevalent realm of parallel applications with a high degree of sharing [8]). By providing a proposal to support fast enforcement of mutual exclusion, we believe many more applications can be operated in the exciting realm of GPGPU computing and we will show how such a task may be accomplished.

While more details will be provided in later sections, our proposal attempts to improve the latency of acquiring a mutex by distributing management across all nodes. If not done intelligently however, this method alone can lead to a configuration that's no better than having a centrally-located directory – albeit with improved load-balancing characteristics. In order to gain benefit beyond load-balancing, we improve the average latency (evaluated in terms of hop count) that a random request would experience by allowing multiple nodes to be grouped into sets that are all authorized to respond on behalf of mutexes in the same region. This leads to at least one of the set being closer on average to a randomly selected single node. Of course this introduces a challenge of unambiguously ordering racing requests that land on different nodes in the set. In order to resolve this, we utilize the ordering properties of an inherently ordered topology (ring, tree, etc.) to logically connect each of the nodes that are responsible for the same portion of the mutex space. As we will show, this can be done with negligible cost on common interconnect topologies.

The type of system that can make use of our proposal varies widely, but in general consists of nodes in any topological configuration (mesh, torus, butterfly, ring, etc.) and our context will generally be that of a physical implementation of an on-chip interconnect, though it is by no means necessary. The only requirements are that the nodes be logically divisible into multiple rings (the ordered interconnect we will work with in the remainder of the paper) – where all members of a ring are able to communicate with one another – and that all nodes are able to communicate with at least one member of all logical rings. One common configuration for our evaluation is that of nodes connected in a mesh topology where certain subsets – of varying size from 1 node to all nodes – constitute a ring. Since all nodes in a mesh can communicate with all other nodes, the simple constraints are easily satisfied and evaluated in this setup. This configuration has the added benefit of being nicely scalable and relatively easy to implement – making it quite likely to be a topology of choice for near-future real-world implementations.

2. Background

Given the context of an electrically-connected, physical processor network, there is no direct work to call upon that implements distribution of locks or mutexes. Previous work in distributed mutual exclusion has primarily dealt with software approaches to enforcing mutual exclusion and as such, requires mapping of these proposals to a physical network in order to properly evaluate them. In particular, the areas of operating systems and databases have proven fruitful for uncovering methods for enforcing mutual exclusion in a distributed manner. In operating systems, the proposals understandably center on mutual exclusion in critical sections, while databases revolve around replicated data – a more natural fit to our constraints. In these

distributed systems, data is replicated across a number of nodes to improve the fault-tolerance of the system and to a lesser extent, improve latency by having data more evenly distributed and thus likely closer to any given node (the same technique we employ to improve performance). The primary metric of interest in these systems is message count with the performance improvement being inversely proportional (i.e. a lower message count means higher performance since there are fewer replicas to communicate with). Due to its status as a second-order constraint however, latency is not often directly quantified. To further complicate the matter, the method for counting messages in these proposals can sometimes be equated with hop count – thus giving us a proxy for latency – but many times cannot because often only 1 message is counted to traverse from source to destination regardless of the number of intermediate nodes or latency associated with long point-to-point links. Attempting to rationalize about the benefit of applying a given approach to our problem then requires a thorough understanding of the proposal.

As a baseline, our evaluation considers a very straightforward approach that intuitively maps to a mesh topology. In this baseline – termed a fully-distributed directory – each node in the system acts as the home node for a subset of the mutex space (e.g. 1/16th of the mutexes have their home in any particular node in a 16-node system). Whenever another node wishes to acquire a mutex, it need only request it of the home node. If the mutex is present, the home sends it to the requester; if it is not present, the home can either NACK the request, queue it up to be satisfied when the current holder releases, or any other strategy of those commonly used by directory-based cache coherence protocols. The specific response is not of concern in this work's current incarnation as we are now only attempting to minimize the expected contention-free common case – where a node's request for a mutex is the only such request in the system. We

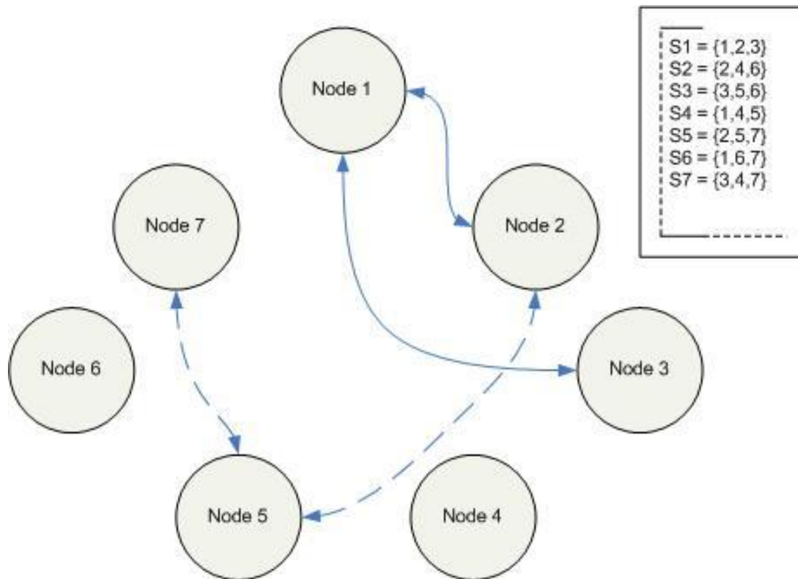

Figure 1. Maekawa Example

will use the straightforward approach of average hop count as a proxy for performance in such a system.

The best proposal coming from the systems realm given our constraints and the dimensions we would like to optimize for is an elegant proposal by Maekawa [1]. In this system, properties of finite projective planes are applied to nodes in a system to create a number of sets equal to the number of nodes. When a node wishes to atomically update a data item, it generates requests only to the nodes in a pre-determined set. We call this set its “home set” of which it is typically a member for performance reasons, though it is not strictly required. All sets in the system are created to ensure the following properties:

1. Between any two nodes, there exists a common set
2. Between any two sets, there exists a common node

For example, consider the 7-node system in Figure 1 with the sets given in the text box. The connections for sets 1 & 5 (S1 & S5) are given by the solid and dashed connectors, respectively (note that node 1 is a member of S1 and node 5 of S5 to reduce the number of connections

required for each set). Consider the situation where both node 1 and node 5 generate a request for the same resource at the same time. By design, node 1's home set is S1 and node 5's is S5. Since these sets have a common node – node 2 – whichever request is processed by node 2 first will receive the resource while the other will not (it is either queued, NACKed, etc.). It can be seen that the selection of the other sets ensures that there is at least one common node for any two sets and any two racing requests will be unambiguously ordered. In order to resolve races involving more than 2 nodes, extra message classes are introduced that impact the latency of requests. However, since we're initially concerned with evaluating the best-case performance, these are not explored in more detail.

This proposal is an example of one where message count cannot be directly related to latency because this system assumes point-to-point connections between every node in a set – a constraint that quickly becomes infeasible when the node count increases in a physical network. In order to mimic point-to-point connections in a mesh, one can use the network to route messages between all the nodes in a set. Given the constraints of a finite projective plane, however makes optimally laying out the nodes very difficult and in the end perform worse than the baseline. This is due to the fact that locating all the nodes in one set close to one another creates issues when each of these nodes must in turn be connected to multiple other sets – ideally not containing any of the nodes in the current set – requiring an intelligent approach to laying out the nodes to globally minimize distance. While an optimal approach was not identified, nor was exhaustive testing of configurations performed, simulations run on a 4x4 mesh, iterating millions of configurations resulted in no better than an average round trip hop count of 5.875 compared to the average hop count of 5 in the baseline. This approach can be easily applied to other topologies and shown to continue to perform worse than the naïve baseline implementation.

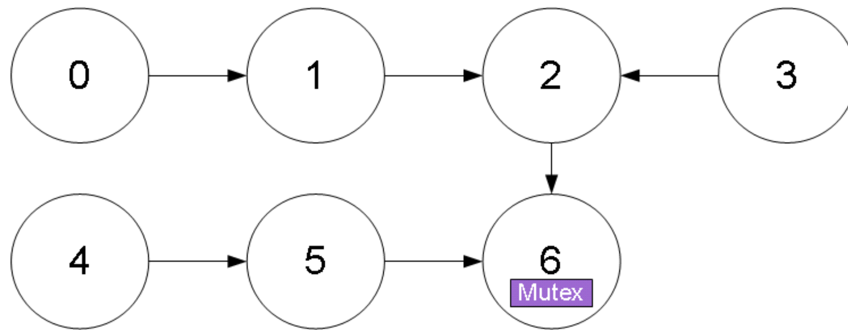

Figure 2. Raymond's Unrooted Trees

While Maekawa's concern with communicating with as few nodes as possible in the system is also nice for our purposes, it is more important that the distance to the farthest node is minimized to reduce round-trip latency. So, while the number of nodes that need to receive messages can be made relatively small, it does not optimize the dimensions we are concerned with if the time it takes to communicate with all of them is relatively long.

The best proposal in the database realm that can be made to fit within our constraints is that proposed by Raymond [4]. In this simple system shown in Figure 2, unrooted trees are used to create a sort of linked list from one node to the node holding a mutex. Whenever a node wishes to acquire the mutex, it makes a request of its neighbor that it knows to be the next nearest to the mutex. In this way, the request can traverse down the tree to the current holder of the mutex and be transmitted back. On the journey back, each node updates itself to indicate a new one of its neighbors is the root of the tree to the mutex. Unlike Maekawa, Raymond’s claim of message count can actually be directly related to hop count in the sort of system we envision as our implementation. In the proposal, a new message is generated whenever a node requests a mutex from its neighbor. This message is generated whether the node generates the initial request or if it is just relaying a request from one of its neighbors. This results in each message in the traversal being equal to one hop. Unfortunately this proposal can easily suffer from pathological cases resulting from the fact that the traversal a request takes essentially follows the path the

mutex took to its current destination. If that path was indirect (e.g. a spiraling path from the outside of the mesh to the center), the path the request takes can be far from optimal. This feature makes characterizing the proposal in terms of average hop count challenging, but unnecessary given the fact that it doesn't allow multiple nodes to respond to mutex requests so it degenerates to the baseline case in the best-case scenario (where the path to the mutex is direct).

As a result, no method was uncovered in the current literature that could outperform the naïve baseline case when the goal was to optimize the round trip latency (as measured by hop count) to acquire a mutex. While each of the presented proposals succeeds in avoiding hot spots in the network - unlike a centrally-located directory - the latency characteristics make it undesirable for our purposes. Ideally a system would exist that would avoid creating hot spots in the network while at the same time improving latency to acquire a mutex.

3. Proposal

In order to provide such a system as mentioned in the previous section, we attempt to couple the nice ordering properties of a ring (though a bus, tree or other inherently ordered topology would also work) with a more scalable communication substrate to allow system designers to balance the latency of making a request (transported on the scalable substrate, termed 'transport' or 'xport' latency) with the latency of ordering racing requests (in the ring, termed 'rotational' latency). In our system, all nodes are divided into multiple logical rings. Each ring is then responsible for a subset of the mutex space. For instance in a 16-node mesh, one configuration could be rings of size 4, in which case each of the 4 rings in the system would be responsible for one quarter of the mutex space and each node in a given ring would know the status of every

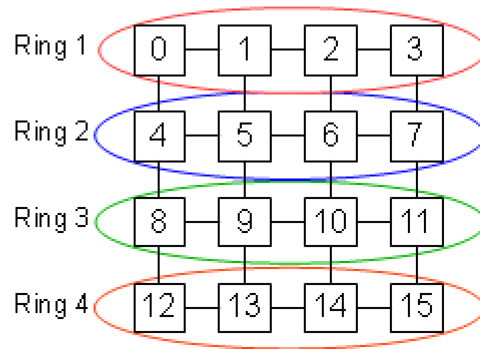

Figure 3. 4x4 Mesh Example

mutex in its responsible space (even if another node in the ring most recently responded to a request for a mutex). When a node wishes to acquire a mutex, it need only transmit its request to the nearest node in the appropriate ring. Once received at a node in the proper ring (hereon termed the "responder"), the responder waits for its "turn" to respond to requests as identified by a token that is circulated among the nodes in the ring. Once it receives the token, if the requested mutex is available, the responder will positively respond to the requester, update its mutex directory information, and subsequently notify the other nodes in the ring that the mutex is unavailable by piggybacking a message on the token.

As an example of operation, consider the 4x4 mesh configured with each row as a logical ordering ring as shown in Figure 3. In this configuration, consider the mutexes being distributed such that Ring 4 is responsible for the upper one quarter of the mutex space[1]. Therefore, when node 0, for instance, requests a mutex from the upper one quarter of the mutex space, it will direct its request to node 12 (the closest node in Ring 4). Once node 12 receives the request, it waits to receive the ordering token (an average rotational latency of 1.5 cycles for a 4-node ring). Once it receives the token – assuming the mutex is available – it generates a positive response for node 0 to let it know that it now possesses the mutex, updates its mutex information to

[1] The "mutex space" is the set of all mutexes in the system and can map at any desired granularity to the addressable memory. For example, each cache block could have its own mutex, meaning the mutex space would consist of a number of mutexes equal to addressable memory divided by cache block size.

indicate the corresponding mutex is now unavailable and sends a message to node 13 on the next cycle along with the ordering token to indicate that the mutex is no longer available.

It can be seen from this example that any other request for the same mutex would be unambiguously ordered with respect to node 0's request and only one would succeed. As mentioned previously, the mechanism for handling a racing request could then be any of a number of well-understood mechanisms used in handling contention in a directory-based coherence protocol (queuing, NACKing, etc.).

Now this configuration begs the question: "How are the tokens and any update messages transmitted from the nodes on the right side of the mesh (e.g. Node 15) to the nodes on the left side (e.g. Node 12)?" For the token this is a simple matter of utilizing an "implied token," meaning that each node knows which clock cycles it is in possession of the token. For instance, Node 12 can know that it has the token on cycle 0 and every 4th clock cycle after that. To address the update messages - assuming a lightly loaded system - a single "busy" wire can connect each of the nodes in a ring to indicate if an update is in flight. If an update is inflight, then no node may respond to a request even if it has the token. The update would then traverse the nodes in the ring as any other packet in the network would. If it is decided by the designer that requests are too frequent to utilize a single wire, more wires can be added to the network to subdivide the ring's mutex space. For instance, one wire can indicate an update is inflight for a mutex in the upper half of this ring's mutex space while another wire corresponds to the lower half. The actual meaning of each wire is completely up to the designer who may require a different definition for each wire (e.g. each wire can correspond to a more complex hash function of the mutex as opposed to upper/lower) if it will improve the response time for their expected request patterns.

This approach can be extended beyond merely ensuring correctness to further improve performance. If all that is required is a handful of wires to connect nodes in a ring, the nodes can be almost arbitrarily distributed in the system to further reduce the average transportation latency (the latency for a request to travel from a requestor to a responder) for a node to make a request. While this opens numerous optimization possibilities, the evaluations performed later in this paper will assume the simple linear arrangement of nodes in a ring as represented in Figure 3 and will only explore the case of a single request in an otherwise idle system (a situation which should closely approximate the case of a single busy wire in a system with an average request rate less than the time it takes the average message to traverse the entire ring).

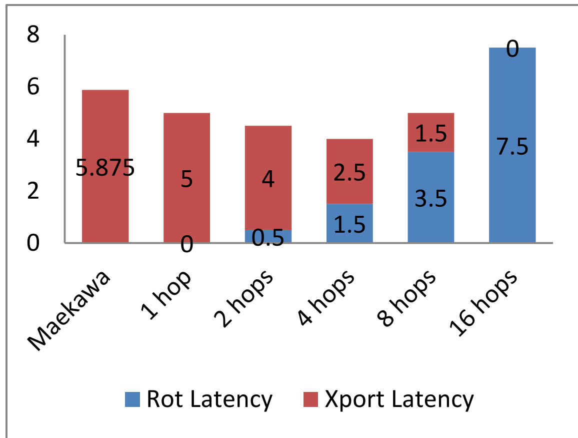

Figure 4. 4x4 Mesh Results

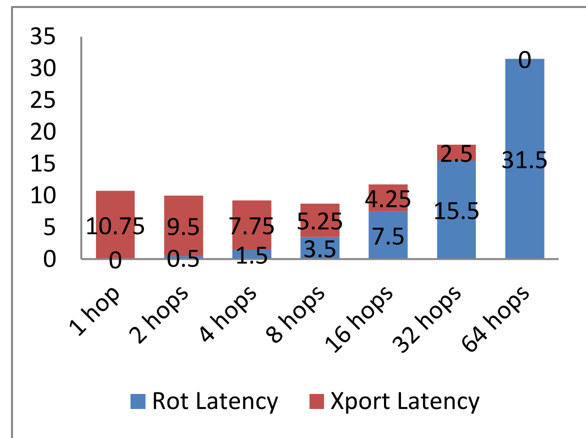

Figure 5. 8x8 Mesh Results

In addition to the choice of implementation of communicating tokens and updates and the distribution of the nodes in the rings, the system designer has the flexibility in sizing the rings relative to the underlying topology to optimize the latency of acquiring a mutex. As Figure 4 and Figure 5 show, there is a sweet spot (given the linear organization of nodes in a ring) for node size relative to the number of nodes in the mesh. Similar graphs can be obtained for other topologies and ring sizes as shown in Figure 6 and Figure 7 for the flattened butterfly and torus topologies, respectively.

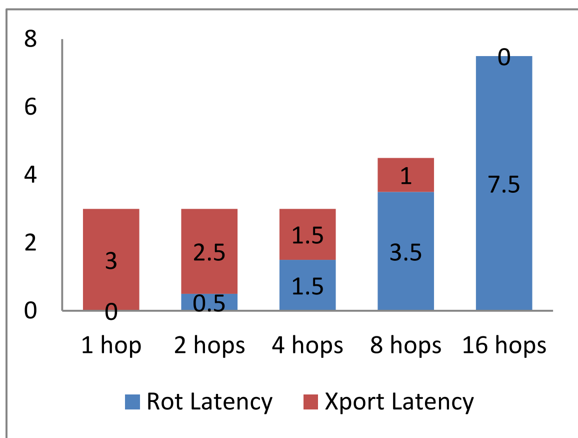

Figure 6. 4x4 Flattened Butterfly Results

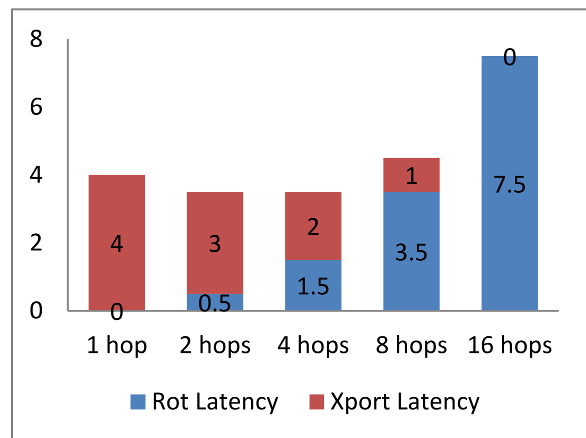

Figure 7. 4x4 Torus Results

4. Application

In order to evaluate the proposal more rigorously and on realistic workloads, we would like to simulate its implementation on GPGPU applications. Specifically, we would like to implement fast acquisition of mutexes to improve the speed at which CUDA applications can perform atomic operations. While current GPUs support a large number of atomic operations, their use is discouraged due to their high-latency characteristics. Unfortunately, many parallel workloads share non-trivial amounts of data and need to use atomic operations. This unnecessarily limits the extent to which high performance computing applications can be ported to GPGPU.

Despite their use being discouraged, we have identified at least 4 benchmarks across the CUDA SDK [5] and Parboil [7] benchmark suites, along with 2 in-house CUDA applications that appear to utilize atomic instructions to a non-trivial extent (see Table 1 for listing and descriptions). If given a fast and efficient method for performing these atomic operations, we believe this number of applications would greatly increase.

Name	**Suite**	**Description**
Mandelbrot	CUDA SDK	This sample uses CUDA to compute and display the Mandelbrot set interactively. It also illustrates the use of "double single" arithmetic to improve precision when zooming a long way into the pattern. [5]
MRI Gridding	Parboil	Computes a regular grid of data representing an MR scan by weighted interpolation of actual acquired data points. The regular grid can then be converted into an image by an FFT. [7]
Histo	Parboil	Computes a moderately large, 2-D saturating histogram with

		a maximum bin count of 255. Input datasets represent a silicon wafer validation application in which the input points are distributed in a roughly 2-D Gaussian pattern. [7]
BFS	Parboil	Computes the shortest-path cost from a single source to every other reachable node in a graph of uniform edge weights by means of a breadth-first search. [7]
Cortical Column Model	In-House	Work queue implementation using atomic increments to enforce processing order of cortical column model.

Table 1. Benchmarks with Non-Trivial Atomic Usage

For our simulation infrastructure, we will use GPGPU-Sim [2], a GPU simulator that is widely used for research on GPGPU tradeoffs. Unfortunately, atomics are one area where GPGPU-Sim is relatively weak and – possibly a contributing factor – documentation on how GPUs support atomics is sparse. As such, we will be required to add reasonable support to GPGPU-Sim prior to implementing our changes. Our goal will be to add support that is consistent with available documentation, but we don't anticipate validating all of our design choices against hardware as that's a difficult and time consuming task and without documentation to verify, it's impossible to know for sure. Instead, where there may be some doubt as to implementation, we hope to implement all feasible solutions to see how they respond to our proposal. Where not feasible to implement all options or where otherwise deemed appropriate, we can run microbenchmarks to tease out certain characteristics of a GPU's implementation details.

Some GPU implementation details that are not well understood that may be important to understand for our modeling and simulation are summarized in Table 2 along with descriptions of microbenchmarks that may be used to uncover the actual details.

Detail	Microbenchmark Description
Do functional units exist at the L2 to execute atomic instructions at the L2?	1. Determine L2 hit latency a. With 1 thread, iterate through array that's too large for L1 b. Once access rolls back around to block that should have been evicted from L1, determine latency to get data from that access (should equal 1 round trip to L2) 2. Decide if atomic operations that more or less than 2X this L2 hit latency a. Iterate through same array as before, but this time with atomic instructions
Determine coherence between atomic and non-atomic instructions that are executed concurrently.	Threads on separate SMs: 1. One updates atomically, the other not 2. One increments data by one value (e.g. 7 or 10), the other by another (e.g. 3 or 1000) 3. See if the non-atomic operation ever sees the update of the atomic operation

Table 2. Microbenchmarks to Discover GPU Implementation Details

5. Future Work

Following evaluation of the proposal with GPGPU atomics, we would like to advance the concept in a number of ways. First, the current proposal is only concerned with reducing latency in the contention-free case. While we believe this is a reasonable assumption for most workloads, thorough analysis will need to be done in the face of high contention to determine if more development needs to be done. As discussed in the "Application" section, we believe that simple methods of NACKing or queuing will be sufficient, but that needs to be proven. Additionally, certain parameters of the NACK or queue mechanism need to be evaluated, such as buffering, ordering, and other optimizations. While many proposals for these have been

presented in cache coherency literature, determining which strikes the right balance for distributed mutual exclusion is necessary.

Along with exploring performance with contention, an interesting study could be done with regard to distributing the nodes in the ordering topology. As mentioned previously, the fact that relatively few "busy" wires are required per ring allows almost arbitrary placement of nodes within the same ring. This provides a broad design space to explore different layouts of nodes in a ring. While we've briefly evaluated 2-dimensional layouts of nodes (where all nodes in a ring are still co-located) and found them to perform worse than a 1-dimensional, linear layout, distributing nodes where they're not co-located opens many possibilities for reducing transport latency. As long as the "busy" wires are designed in an appropriate way, the rotational latency should be unaffected on average.

In addition to GPGPU atomics, another interesting application of the work is Atomic Coherence. As this was the motivating concept for developing the proposal in the first place, it would be an interesting exercise to see how well Atomic Coherence could perform on an electrical interconnect that implemented fast distributed mutual exclusion. It would provide a nice proof of concept for Atomic Coherence's latency-tolerance studies and a good evaluation of whether it is actually feasible on an electrical interconnect.

One final application we'd like to explore is that of memory consistency. Much as Atomic Coherence provides a means for simplifying cache coherence, we believe there is much that can be done to simplify memory consistency, from both a hardware simplification perspective and a logical reasoning perspective. If a memory reference is required to acquire a mutex prior to performing a load or store instruction, that processor can know unambiguously that no other processor in the system is also making a reference to that memory location. This allows non-

speculative retirement of memory instructions without fear of violating consistency. Additionally, if this mutex acquisition can be accomplished in a handful of cycles, a processor can be simply modified to ensure that the mutex is acquired while the instruction is inflight (e.g. by adding a handful of pipe stages to cover the latency). This should save power (by reducing speculative execution) while maintaining performance.

In addition to these applications, as mentioned in the introduction there exists a wide range of applications that could greatly benefit from a mechanism for quickly enforcing mutual exclusion. As computing enters a new multicore era, it becomes more important than ever to provide programmers with skills and tools to make parallel programming easier. What was once a specialized realm, populated with experts in the field, parallel computing is becoming ubiquitous and extensive work has been done to evaluate how we might take advantage of this abundant parallelism with tools and a workforce that are used to working sequentially. Tools such as RapidMind [9] and parallel libraries such as Intel's thread building blocks (TBB) [6] seek to abstract parallel programming from the programmer, while studies are underway and proposals are being made to push parallel programming earlier in the training process for future programmers. We believe that by making mutual exclusion fast, a portion of unnecessarily complex techniques (those to avoid mutual exclusion) can be avoided and be coupled with other methods to make parallel programming easier and more intuitive. Also, along with improving coding of future applications, there could be re-evaluation of existing applications to improve their performance and/or simplicity by undoing convoluted methods for avoiding mutual exclusion that have been done in the past.

6. Conclusion

We have presented a mechanism for enforcing mutual exclusion utilizing ordering properties of certain interconnect topologies that can be logically laid on top of virtually any other interconnection fabric with little penalty. By properly sizing the ordering fabric with latency characteristics of the physical interconnect, system designers can greatly improve the latency to acquire a mutex in a system. In so doing, this method allows the designer to de-couple the latency between transmitting requests for a mutex and ordering racing requests. Additionally, the system designer has many other design choices to optimize the mutex acquisition network given their expected communication patterns and latency tolerances (e.g. optimizing the number and/or meaning of busy wires in the ordering fabric).

In addition, we have presented a path for applying this concept to atomic operations in GPGPU. While GPGPU is a hot topic in computing and extensive work has been done to improve the ability of programmers to write GPGPU applications, the fact remains that it is difficult to write functionally correct code that also performs well. Additionally, many workloads have not been ported to GPGPU from the high performance computing realm because they rely on atomic updates of shared data. By making atomic operations faster, not only can these workloads realize the benefits of GPGPU computing, but new classes of applications can be developed more quickly when the functional requirement is made easier without significantly impacting performance.

References

[1] M. Maekawa, “An algorithm for mutual exclusion in decentralized systems,” ACM Transactions on Computer Systems (TOCS), vol. 3, no. 2, p. 145–159, 1985.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, 2009, p. 163–174.

[3] D. Vantrease, M. H. Lipasti, and N. Binkert, “Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols,” in Proceedings High-Performance Computer Architecture, 2011.

[4] K. Raymond, “A tree-based algorithm for distributed mutual exclusion,” ACM Transactions on Computer Systems (TOCS), vol. 7, no. 1, p. 61–77, 1989.

[5] “CUDA Toolkit 2.3 Downloads | NVIDIA Developer Zone.” [Online]. Available: http://developer.nvidia.com/cuda-toolkit-23-downloads. [Accessed: 03-Jun-2011].

[6] C. Pheatt, “Intel® threading building blocks,” Journal of Computing Sciences in Colleges, vol. 23, no. 4, p. 298–298, 2008.

[7] “Parboil Benchmark suite.” [Online]. Available: http://impact.crhc.illinois.edu/parboil.php. [Accessed: 03-Jun-2011].

[8] C. Bienia and K. Li, “PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors.”

[9] M. Monteyne, “Rapidmind multi-core development platform,” RapidMind Inc., Waterloo, Canada, February, 2008.

[10] S. Che et al., “Rodinia: A benchmark suite for heterogeneous computing,” in Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, 2009, p. 44–54.

[11] R. Rajwar and J. Goodman, “Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution,” Microarchitecture, IEEE/ACM International Symposium on, vol. 0, p. 294, 2001.

[12] M. Martin, M. Hill, and D. Wood, “Token coherence: Decoupling performance and correctness,” in ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 2003, vol. 30, p. 182–193.