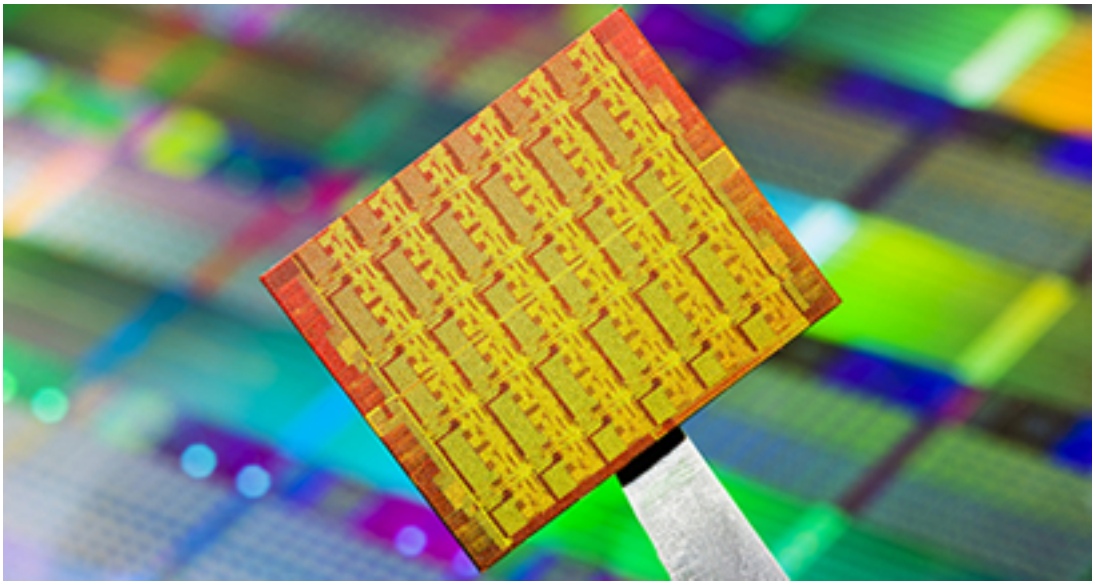
CHALMERS



Synchronization and memory consistency on Intel Single-chip Cloud Computer

Master of Science Thesis in Programme Computer Systems and Networks

Ivan Walulya

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Synchronization and memory consistency on Intel Single-chip Cloud Computer

Ivan Walulya

Examiner: Philippas Tsigas

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover: Image of the SCC wafer showing outlines of multiple tiles placed on the wafer. source: http://communities.intel.com/community/marc]

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

The Single-chip Cloud Computer (SCC) is an experimental multicore processor created by Intel Labs for the many-core research community. The chip is built to study many-core processors, their programmability and scalability while utilising message-passing as a communication model. The chip has a distributed memory architecture that combines fast-access on-chip memory with large amounts of off-chip private and shared memory. Additionally, its design is meant to favour message-passing over the traditional shared-memory programming as is the norm for distributed memory systems. To this effect, the platform deliberately provides neither hardware supported cache-coherence, nor atomic memory read/write operations across cores and the on-chip memory, also known as message passing buffer is quite small.

The SCC provides support for very fast communications among the cores with reduced latency. This allows for the creation of very efficient message-passing protocols and support for message-passing programming model. This design employs explicit exchange of messages among different processors, implicitly avoiding data consistency issues arising from concurrent data access and merges both communication and synchronization.

However, due to the limited size of the message passing buffers, the message-passing is ideal for transfer of small amounts of data, but not very efficient for large data transfers. In addition, replicating all datasets and exchanging them as messages is less efficient and more wasteful than using the data directly in shared memory. In some cases, the message data read from the main memory, is passed through the message passing buffers and eventually written back to the same memory modules. Besides, the chip provides access to shared memory allowing the cores to share data without necessarily copying it among the cores over the on-chip network.

In this thesis, we develop procedures for sharing data among multiple cores; concurrently coordinated by message-passing on the Single-chip Cloud Computer. We further make and investigate a proposition that, for architectures that combine message-passing with shared-memory, the message-passing is not necessarily essential for data-transfer but for coordinating shared-memory access and synchronization of operations on the different cores.

Acknowledgements

I would like to express my sincere gratitude to my project supervisors Ioannis Nikolakopoulos and Daniel Cederman for all the support, guidance and encouragement they availed. Most of all the time they took off their busy schedules to discuss my progress throughout the project. I would also like to thank my examiner Philippas Tsigas, first all for giving me an opportunity to perform my research with the Distributed Computing and Systems group, then for his guidance and working tirelessly with me on this thesis project.

My study in Sweden was fully funded by the Swedish Institute, and I want to pass on a token of appreciation to all the staff at Swedish Institute for making it all possible.

Finally I'd like to thank my family for their love, support, and encouragement all through my academic life.

Ivan Walulya
Göteborg, July 2013

Contents

List of Figures

1.0 Introduction

MANY-core or multicore processors are processors with many cores on a single chip. The trend to many core architectural design was a response to diminishing performance gains achieved by increasing the speed of a single processor. This decrease in gains is mainly a result of memory access latency brought about by the differences in processor and memory speeds. Additionally, the exponential increase in power dissipation with increase in processor operating frequency has also created a power wall for processor manufactures.

To mitigate the increasing power demands and continue to achieve significant performance boosts without increasing the clock rate, chip manufacturers decided to add multiple lower clock speed cores on a single chip. These multi-core processors can also significantly reduce overall system power dissipation by utilizing dynamic frequency/-voltage scaling techniques. These techniques exploit program behavior to reduce power dissipation and are usually employed at core level. Some examples of existing many-core chips are the Tilera with 100 cores [3], the IMB Cyclops64 with 160 thread engine [4], and the Single-Chip cloud computer with 48 single ISA, X86 architecture cores with Linux loaded on each core [5].

However, as the number of cores per chip is increased, the complexity of scaling hardware supported cache coherence across cores becomes of great concern in architecture designs [5]. The overhead incurred in maintaining cache coherency across cores may surpass the performance improvements gained through addition of cores.

This has compelled some processor designers to eliminate hardware supported cache coherency so as to increase the core count on the chip. With the elimination of hardware coherence, application developers are left with two alternatives. One is software managed cache coherence, and the other is shifting to the message-passing programming model (viewing the cores as a cluster of computers connected by a mesh network).

The Single-chip Cloud Computer (SCC) is a 48-core homogeneous experimental processor created by Intel Labs for the many-core research community. The chip is built with three main objectives; one is to study many-core processors, their programmability and scalability while utilizing message-passing as a communication model. Second, explore the performance and energy consumption characteristics of on-die 2D mesh fabric. And finally, investigate the benefits and challenges of fine-grained dynamic voltage and frequency scaling at application level.

The design of the processor prototype is meant to favor message-passing over shared-memory programming. To this effect, the platform deliberately provides neither hardware supported cache-coherence, nor atomic memory read/write operations across cores. The cores communicate using a message passing architecture that permits very fast data sharing with low latency through the on-die mesh network. Hence, the platform is explored by application developers as a cluster-on-chip or a distributed system with cores connected through a Network-on-Chip(NoC).

Traditionally we implement concurrent data structures in shared-memory to coordinate the execution of various threads. Different threads call the object methods to access

the shared objects. Access to the shared object may require synchronization to avoid race conditions. A race condition arises when the output of a concurrent execution is dependent on the sequence or relative ordering of the method calls. To overcome pitfalls that arise due to race conditions, concurrent executions are synchronized using atomicity or mutual exclusion in critical sections.

Mutual exclusion can trivially be achieved using locks guarding a critical section. A thread requiring access to shared memory must acquire a lock, and releases the lock afterwards. However lock synchronization has many pitfalls; these include blocking, deadlocks, convoying effects, priority inversions on multithreaded machines, among others[6]. Non-blocking data structures have been constructed to overcome these hazards. "Non-blocking" as used in this context, is a guarantee that system progress cannot be indefinitely blocked by a slow or halted process.

Developing non-blocking concurrent data structures is nontrivial, and correct implementations of these data structures is a demanding task for application developers. However; over time researchers have developed very efficient implementations of non-blocking data structures [7, 8, 9, 10, 11, 12, 13].

In literature several progress conditions for these non-blocking data structures have been defined such as wait-free and lock-free [14]. A *wait-free* concurrent object has each concurrent operation finishing in a finite number of its steps, despite other concurrent operations. A *lock-free* object guarantees that at least some method call will complete within a finite number of its steps.

Nonetheless, most of the non-blocking data structures we have found in literature are suited for shared-memory programming with hardware supported cache coherency and universal atomic instructions. This raises the question that: "How can we develop similar data structures for message-passing programming model or new many-core multiprocessor systems, analyse these structures theoretically (prove linearizability) and empirically (performance measurements)?" On the new many-core architectures; communication overhead, core locality on the chip and memory layout all have an influence on system performance and scalability of developed data structures .

Concurrent First-in-first-out (FIFO) queue is one such data structure that has become fundamental to applications in shared memory multiprocessors. In this thesis we develop and study various implementations of a statically allocated FIFO queue ported to the single-chip cloud computer.

1.1 Problem Statement

Message-passing is very ideal for transferring small amounts of data, but not efficient for large data transfers among cores. Hence, while a lot of research done into fast efficient message passing protocols for sharing data among cores. We propose that, for architectures that combine message-passing with shared-memory, the message-passing may not necessarily be essential for data-transfer, but for coordinating shared-memory access and synchronization of operations on the different cores. The SCC being a hybrid system (provides both shared and private memory), avails shared memory that can be used for the shared data structures. The data in shared off-chip memory can be used

in place, contrary to incurring overhead while passing data among cores or the space misuse associated with replicating data on all cores.

The primary aim of this thesis project is to develop concurrent data structures coordinated by message-passing on the single-chip cloud computer.

1.2 Justification

The study will form a first phase of research into concurrent data structures on many-core or multicore systems. This ultimately aims to provide readily available data structures and synchronization constructs to application developers on these platforms. It will also provide a body of information to both the student researcher and the research group on these nascent research platforms.

1.3 Methodology

Study the single-chip cloud computer, its architecture capabilities, trade-offs and hardware constraints. Design and develop message-passing protocols to synchronize memory access and concurrent data structures using the C-programming language. Undoubtedly, all this will require as the first step a comprehensive literature review of concurrent data structures themselves.

1.4 Literature review

The FIFO queue is studied widely as a concurrent data structure for both statically [7, 10] and dynamically allocated memory [8, 9, 10, 11, 12, 13]. A queue is classically a sequence of typed items, with the enqueue operation adding an item to the end of the sequence and the dequeue operation returning the item at the front and removes it from the sequence. A concurrent queue is usually desired to be a linearizable implementation of a classical sequential queue. Where linearizability is a correctness property that each method call appears to take effect atomically at some point between the invocation and response, and that the ordering of non-concurrent operations should be preserved [15].

In recent years, many researchers have developed efficient algorithms for concurrent FIFO queues. Michael and Scott [16] present a two-lock algorithm that performs better than a single coarse-grained lock. In their implementation, two locks a maintained one for the head and the other for the tail, allowing one enqueue and one dequeue proceed concurrently. The authors suggested this implementation as the best option for hardware architectures with non-universal atomic primitives e.g. test&set. They also present a linearizable lock-free queue based on Compare-and-Swap (CAS) atomic hardware instructions, which allows for operations at both ends of the queue to proceed concurrently. They used modification counters for each pointer to overcome the ABA problem.

Gottlieb *et al.* [8] present blocking algorithms based on a ”replace-add” primitive which seems universal. They do not use locks, but the algorithm can suffer from starvation. A slow thread can indefinitely halt the progress of faster threads or the system as

a whole.

Stone [9] presents a Compare-and-Swap queue algorithm which allows for an arbitrary number of concurrent enqueue and dequeue operations. However, this algorithm is not linearizable and not non-blocking as a faulty or slow enqueuer could block all the dequeuers.

Tsigas and Zhang [7] present a simple non-blocking, linearizable queue based a static circular array. They lower contention on key variables by allowing the tail/-head pointer to lag at most m positions behind the actual tail/head of the queue, thus allowing only one in every m operations to update the tail/head pointers using a *compare&swap*. This is because, contention to shared-variables degrades performance and generates a lot of traffic on the interconnection network. They also introduce a mechanism to deal with pointer recycling or A-B-A problem which results from algorithms using the*compare&swap* primitive. This is simply done by designing the queue as a circular array, with the circular array acting as a counter.

Prakash, Lee, and Johnson [13] present a simple efficient non-blocking algorithm that uses *compare&swap* instruction for synchronisation. The algorithm is non-blocking. This property is achieved by utilizing a helper mechanism, where a process can complete an operation started by another process. Another property of this data structure is that both enqueue and dequeue operations are independent of queue size. To overcome the A-B-A difficulty of *compare&swap* constructions, the algorithm uses a singly linked list with double sized pointers. One half is the actual pointer while the other half is the counter. The counter is then further divided into a mark-bit and the counter value; with the mark bit indicating whether the node object is to be dequeued.

Valois [11] presents a lock-free singly-linked list that allows concurrent traversal, insertion, and deletion by multiple processors. This algorithm uses a sentinel node at the head of the list to avoid problems that arise due to empty lists or single-item lists. The algorithm has a drawback; it allows the tail pointer to lag behind the head pointer; making freeing or reusing of dequeued nodes unsafe. Not reusing or freeing memory would make the algorithm not very efficient on space and thus unacceptable. The author thus suggests a mechanism to allocate and free-memory based on reference counting. Thus, a node is only freed when its reference count goes to zero.

Virtually all the algorithms above are reliant on the availability of the *compare&swap* hardware primitive and are suited for the shared memory programming model. As its been noted [17], designing non-blocking implementations for many simple sequential data structures is not possible using classical non-universal synchronization primitives (test&set, read, write, fetch&add and swap). Therefore, non-blocking implementations cannot be easily ported to distributed memory systems or architectures that do not readily avail universal atomic operations like *compare&swap* or load-linked/store-conditional.

2.0 Single-chip Cloud Computer - SCC

Single-chip Cloud Computer (SCC) is an experimental multicore processor created by Intel Labs intended for the many core research community [5]. The chip consists of a 6x4 mesh of dual-core tiles, with each tile having two cores. Each core is x86 architecture, hence can support compilers and operating systems require for full application development. The chip supports dynamic frequency and voltage scaling for application-level fine-grained power and clock management. It is divided into multiple voltage and frequency domains, some are configurable at start-up, while others may be varied by applications during runtime.

The die has four on-chip memory controllers as shown in figure 1, making it capable of addressing upto a maximum of 64GB off-chip memory in addition to a small chunk of fast, low latency buffers located on each tile. The fast local memory or on-die memory is mainly utilized to facilitate message-passing communication on the chip.

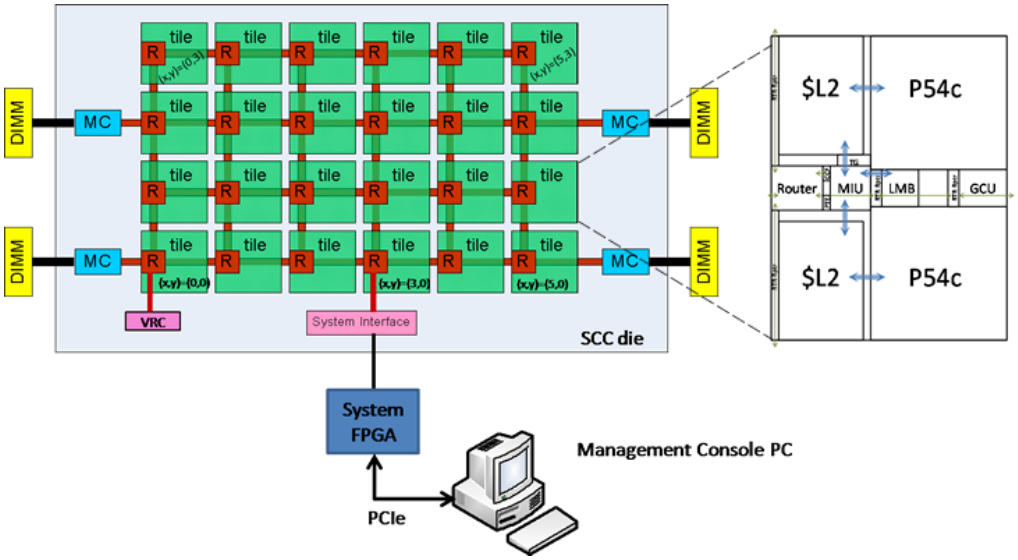


Figure 1: SCC Architecture overview[1]

The off-chip memory is divided into segments private to each core and segments that can be mapped as shared by user applications and accessible to all cores. However, the default configuration is that, each core has a private memory region on off-chip memory and shared-access to the local on-chip memory buffers (Message Passing Buffers).

The cores on the chip boot Linux operating system by default, and provide a programming environment with various APIs to application programmers. Some researchers choose to use the chip without loading an operating system (bare-metal), while others boot custom operating systems suited for multicore systems similar to the SCC [18].

A board management microcontroller (BMC) controls the entire system; it initializes and shuts down critical system functions. A 64-bit PC running Linux operating system

(Management Console MCPC) manages the SCC board over a PCI-e bus using software provided by Intel Labs. This software enables developers to load Linux images on any single core or a set of cores, manipulate configuration registers and load/run applications on the cores. Applications to configure the SCC platform can also be written and run on the Management Console.

2.1 Memory architecture and access modes

The tiles on the chip are sectioned into 4 memory regions. A memory controller is assigned to each region, and is capable of accessing a maximum capacity of 16GB DDR3 memory. The default boot-up configuration is that, memory at each controller is divided evenly to the nearest MB among the 12 cores in memory region as private memory. While, the surplus memory is allocated as shared memory.

Cores belonging to the same memory region access their private memory only through the controller assigned to the region. However, when accessing the shared memory regions, a core may go through any of the four memory controllers, whichever is closest to the block of shared memory that the core is trying to access. Routing to the destination memory-controller is through the on-chip 2D-mesh.

Divisions between the shared and private memory spaces are programmable and configurable using entries in the lookup table and the page-tables of the core. Therefore, the memory can also be mapped as either ”totally shared” or ”totally private”.

Shared memory can be exploited for exchanging data among cores and hosting shared data structures e.g. concurrent lists, graphs among others. It can be mapped either as non-cacheable or cacheable with software managed coherency among multiple caches. To take advantage of the new write combine buffers and cache invalidation, the off-chip shared memory can also be mapped onto a core as Message Passing Buffer Type (MPBT).

SRAM, also referred to as the message-passing buffer is added to each tile and is capable of fast read/write operations. 16KB of SRAM on each tile, making up the equivalent of 384KB of on-chip memory. These 24 on-die message buffers are accessible by any core or through the system interface. Cores located at different locations on the chip experience considerable disparity in access latency while accessing the same buffer. Furthermore, access to these buffers is typically internal to a message-passing model or protocol.

L1 instruction and data caches are 16KB each, and a write combine buffer was added to facilitate and coalesce memory write operations. The SCC core also has a new cache memory type, the Message Passing Buffer Type (MPBT) and a new instruction (CL1INVMB) to aid software managed cache coherence.

Cache lines can be modified with the status bit MPBT. This bit differentiates message-passing data from regular memory data in the L1 cache. And the instrunction, CL1INVMB, invalidates all cache lines marked with MPBT in one clock cycle.

The 256KB L2 cache is private to each core with its associated controller. The

L2 cache is write-back[1] only, no-write allocate[2] and has no hardware cache coherency. There is no mechanism to invalidate data in the L2 cache, unlike the L1 cache; thus the CL1INVMB does not affect data in the L2 cache.

A core can have only a single pending request for memory and will stall until data is returned. On a missed write, the processor continues operation until another read or write miss occurs. On completion of the pending write request, the core continues normal operation. Therefore, with these blocking reads and non-blocking memory write transactions, message-exchange models should make use local reads and remote writes.

Each tile is connected to the mesh via a Mesh Interface Unit (MIU), which intercepts cache misses and translates the address from core virtual address to system physical address using the LUT. The MIU then adds the request to the appropriate queue which could be a DDR3 request, a Message Passing Buffer access or a local configuration register access queue. For traffic destined for the tile, the MIU forwards the data to the appropriate destination on the tile.

The cores being 32-bit, and yet the system memory controllers address up to 34 bits main memory address space each. Look up tables (LUTS) are added to translate core-physical addresses to system-physical addresses. The Local lookup table (LUT) contains configuration registers used to determine the destination and the access mode of memory requests.

2.1.1 Memory Access

As shown in Figure 2; normal private memory access goes through the L1 and L2 caches, then to the disk memory if there are cache misses. Data in shared memory may be configured to bypass L2 cache and go to L1 cache or bypass both caches as uncached shared memory. With the memory data type MPBT, data read from this memory is cached only in the L1 cache. The new CL1INVMB instruction invalidates all such lines in the L1 cache. The following reads or writes to these invalid MPBT cache lines are bound to miss, and cause data to be fetched or written to memory.

A Write combine buffer (WCB) was added to the core because it supports only one outstanding write request. The write combine buffer allows the cache to proceed and process other requests without stalling waiting for data to be written to memory. When data and corresponding address are added to the write combine buffer, the write operation is complete from the core's perspective; the write buffer proceeds to add the data to memory. The WCB is active for all memory mapped as MPBT irrespective of the actual destination memory type.

The write buffer combines data to a block of memory and fills up a cache line before writing data to the destination memory location. This write-merge optimization allows more efficient use of the L1 caches and limits the number of memory writes.

[1] write-back: data is written temporarily only to the cache, memory write is delayed until the data block is about to be modified or replaced

[2] No-write allocate: on a missed write, data is written directly into memory and not fetched into the cache

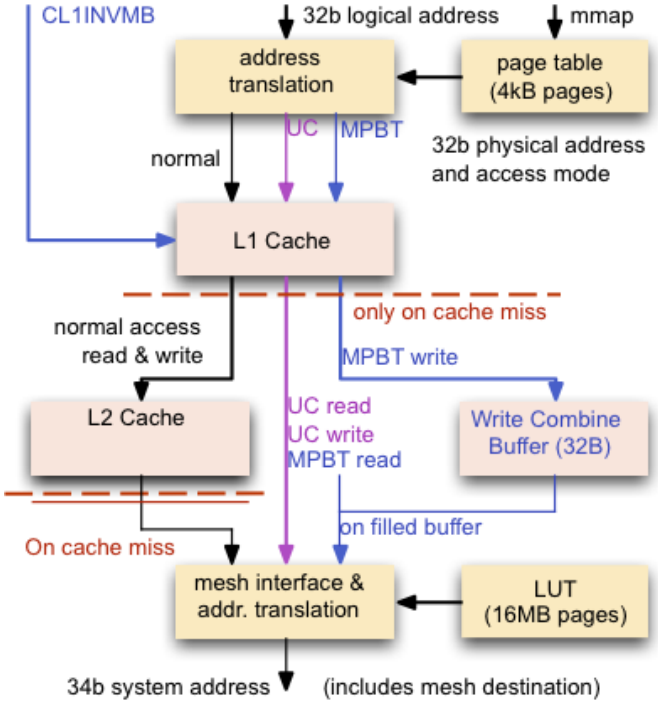


Figure 2: Memory access mechanisms on SCC [2]

However, the merging of memory writes can be a source of read-after-write inconsistencies. A read from a core to a previously written memory address may return stale data values, as the write buffer holds the updated value. Similarly, writes by one core to MPB memory may not be observed by other cores reading from the same memory. One possible solution is to flush the buffer to ensure that data is written to the destination by performing dummy writes to non contiguous memory locations.

The uncached shared memory access mode is also available. Data read from this memory skips the caches and read/write requests are delivered directly to the MIU. As mentioned earlier, there is no cache coherence and the SCC provides no snooping, snarfing or any mechanisms for cache coherence; consequently the programmer explicitly manages data coherence among cores.

Message routing on the mesh employs Virtual Cut-Through[19, 20] switching with X-Y routing. The X-Y routing by Wang et al.[21] is a deterministic distributed routing algorithm. Packets consist of flits which may include header, body and tail flits and routers are identified by coordinates (x, y). Packets from the source core to the destination core first move along the X direction and then along the Y direction (i.e. packets are routed first horizontally and then vertically to the destination). With this deterministic routing, messages from one tile to another always take the same route and messages exchanged between the two tiles take different paths i.e messages from A to B will take

a different path from messages from B to A.

To achieve flow control and avoid deadlocks, the SCC routers use multiple credits with virtual cut-through. Credits are assigned to each link from one router to another. A router only sends a packet to another router if it has credit from that router and when a router forwards a packet, it sends back the credit to the sender.

2.2 SCC Configuration Registers

Configuration registers are available to allow applications control the operating modes of various hardware components of the SCC. The registers are always initialized with default values, and they control core startup configuration, clock divider settings, interrupt handling, and cache configuration. The configuration registers are in the control register buffer (CRB) of the MIU, and they can all be memory mapped by an application.

Each core's L2 cache controller has a configuration register writable by any other core or through the system interface. The L2 cache configuration register is used to enable or disable the core's L2 cache. The Global Clock Unit (GCU) configuration register is used to configure the tile frequency for the predefined router and memory clocks.

Each core is assigned a test-and-set register. This register is used to build locks instrumental in constructing synchronization or communication protocols. Reading from the register returns a 0 if the register was not previously read and not reset while a write resets the register. This register is also memory-mapped and can be accessed by all cores.

2.2.1 FPGA and Global Registers

The Rocky Lake system FPGA bridges the MCPC to the SCC silicon via the chip's system interface (SIF). It allows users to set up the chip environment, control applications executing on the SCC and develop MCPC applications that communicate with the SCC chip. Additionally, the FPGA provides I/O capabilities to the chip, allowing software applications to directly access or communicate with servers via the internet. It can also use external storage such as SATA disk drives for fast local storage.

Application level registers have been added to the system FPGA. These registers can be accessed through read/write requests from the MCPC or the chip cores. The registers are in three categories; a global timestamp counter, atomic increment counters, and global interrupt registers. The registers are memory mapped, and access to them is through the Look Up table. The Global Timestamp Counter provides a common time base to all cores. Although each core has its own timestamp counter, these core local timestamp counters are not synchronized.

2.2.2 Global Interrupt Controllers

The SCC FPGA has 48 interrupt registers for each of the categories; status registers, mask registers, reset registers and request registers, one of each type for each core.

A core can interrupt any other core by setting the destination core's corresponding bit in the source core's interrupt register. Therefore, the requesting core should write to its request register, setting the bit corresponding to interrupted core.

If the destination core Interrupt Mask-bit corresponding to the source core is unset, the interrupt notification is triggered. The interrupted core calls its interrupt handler and checks its status register values to determine the source core of the interrupt request. After handling the interrupt; the interrupted core must clear the local interrupt bit and set the reset register bit corresponding to the source core in the FPGA to force the controller clear the matching status bit, thus allowing the core to receive new incoming interrupts.

To experiment with the interrupt registers provided by the FPGA, we implemented inter-core interrupts as out-of-band message notifications.

2.2.3 Out-of-band notifications

To utilise the message-passing capabilities of the chip, various APIs have been developed. RCCE was developed as a light weight message passing infrastructure suited for the SCC processor [22]. The API was developed to run on the cores without an operating system, thus each core is considered as a single unit of execution.

RCCE employs a symmetric namespace memory model; operations to allocate or deallocate memory are run as collective operations on all active cores. This allows the cores to have a homogeneous view of memory at any state during computation. The environment is best suited for Single Instruction Multiple Data (SIMD) model of programming, with all cores starting at the same logical time and finishing after the application is completed on all cores.

With a single unit of execution per core, all communications are blocking, and this is a significant limitation of the API. Another limitation is that, the environment can not transfer messages larger than 8KB in size from a core. If the messages are larger, they are tokenized into smaller chunks then transferred.

RCCE was extended to iRCCE, adding non-blocking point-to-point communication capabilities [23]. iRCCE employs request handlers to monitor the progress of a communication and ensure that it is not stuck. iRCCE also introduces pipelining to increase the bandwidth of the communications while still utilizing the limited 8 KB MPB.

iRCCE is non-blocking but introduces polling, or repeatedly checking on the progress of the communication, these CPU cycles could be used to carry out other operations as a core waits for a communication to complete.

The cores have the capability to interrupt each other, and this could be exploited to notify cores on completion of a communication. We subsequently decided to investigate exploiting inter-core interrupts to implement non-blocking message passing.

One obvious challenge was how to deliver kernel level interrupts to processes running in userspace. We implemented a kernel module, which registers a kernel handler for hardware interrupts, and signals processes in the userspace about the occurrence of the interrupt.

This notification mechanism enables a process running on a core to alert processes running on other cores about the occurrence of an event e.g. arrival of a message, freeing up of memory, writing a value to a memory location. Each process running on a core registers its pid with the kernel driver at the start of execution. When a process requires to notify another on a different core about an event, it triggers an interrupt on the destination core. On receiving the interrupt; the interrupt handler on the destination core sends a UNIX signal to all registered processes. This signaling allows the kernel mode driver to signal user mode processes about hardware interrupts and thus forwarding the message from another core to any process on the destination core.

We implemented the mentioned procedures, developing a kernel module to handle interrupts and user mode APIs to respond to the kernel signaling. However, the delays were significantly high on the interrupt handling as the FPGA has a much lower clock speed than the cores.

Another drawback was that the network driver on the cores uses the same interrupt pin which increases the number interrupt notifications and causes delays. Therefore, we left this open for future investigation when the FPGA is faster on the SCC. However, we propose it as a viable option for non-blocking message passing on the platform.

3.0 Shared Data Structures

In multiprocessor computing, concurrent shared data structures are typically used to organize data and synchronize access to this data by multiple units of computation i.e. threads, processes, cores. These data structures are associated with liveness properties and safety properties that are provided by their method specifications. To achieve liveness and safety properties, the data structure should provide for threads simultaneously accessing the same data object to reach consensus as to which thread makes progress. The consensus can be achieved using synchronization primitives.

3.1 Safety

Safety of a concurrent data structure is a property that ensures that, despite the various possible interleavings of concurrent executions, the state of the data structure is correct. Safety properties determine the state of the data structure considering various interleavings of method calls by different processes. Several safety properties are suggested in literature [6] (sequential consistency, linearizability, and quiescent consistency), which may be used to argue the correctness of an implementation by mapping concurrent executions to sequential equivalents.

Mutual exclusion is normally used to synchronize concurrent access to elements of the data structure and allow consistent interleaving of method calls. Mutual exclusion is commonly achieved by using locks to create critical sections in which operations that change the state of the data structured are called. However, all mutual exclusion mechanisms lead to the key question; what to do after failing to get a lock already held by another thread or process.

In a multithreaded environment, the only efficient alternative is to yield the processor to another thread, that may not require the lock. This is often referred to as blocking. Context switching is expensive, thus blocking is only efficient if the expected lock delay is long.

In a multiprocessor environment, a process may keep trying to acquire the lock, this is called spinning or busy waiting, and the lock is called a spin lock. Spinning normally results into high contention for the lock and degrades performance of the data structure. Various implementations have been devised to lower contention on the lock, and these are dependent on the construction of the lock.

Locks constructed using test&set registers have been improved upon using test-test-and-set register alternatives. With the test-test-and-set lock, the process only tries to set the lock only after it is observed to be free. This is most useful on the cache-coherent, bus architecture multiprocessors, where the test-and-set is broadcast on the bus. Spinning by repeatedly calling test-and-set causes delays on the bus, and also forces cache updates on all processors owning a cached copy of the lock.

On the contrary, with the test-test-and-set, repeated tests/reads on the lock could use the cached value and reduce bus traffic. When the lock is eventually released, the cached copies are invalid, and a cache miss will occur, thus forcing the processor to read the new value of the lock.

Another improvement to the test-and-set lock, is to have a test-test-and-set with exponential back-off. If on observing the lock as free on the first test; a process fails to acquire the lock on the second step, this is a clear indication of high contention. It would thus be more efficient for the process to back-off, and reduce the contention for the lock.

Apart from high memory contention and sequential bottlenecks, locking has many other drawbacks, mainly in multithreaded environments. Priority inversion, when a lower priority threaded delays a high priority thread. Deadlocks also occur when locks to the same data structure are acquired in different orders.

One critical draw back to implementations dependent on locks is that, a slow or delayed thread can delay faster threads.This is at times referred to as convoying, and occurs when, a process holding the lock is interrupted. All other threads are delayed until the interrupted thread releases the lock. In other cases, the thread holding the lock is halted or crashes, then other threads are delayed indefinitely. Therefore, these algorithms are generally classified as blocking.

Effects of memory contention and sequential bottlenecks can be reduced using fine-grained synchronization. Multiple locks are used to synchronize portions of the object independently, with method calls only interfering when they access the same portion of the object at the same time. This allows multiple operations to execute concurrently if they access different portions of the data structure.

An alternative to mutual exclusion or locking, is to develop non-blocking algorithms. Non-blocking in this context defines the requirement that indefinite delay or failure of a process or thread should not prevent other threads from making progress.

3.2 Liveness

Liveness is a property of concurrent data structures, that as long as there are non-halted processes, there will be system-wide progress. Progress conditions for non-blocking data structures have been classified as wait-free and lock-free. Lock-freedom ensures system wide progress, while wait-freedom guarantees thread level progress.

A *lock-free* algorithm guarantees that at least one method call will finish within a finite number of its steps. A *wait-free* concurrent object has each concurrent operation finishing in a finite number of its steps, despite other concurrent operations. Therefore a wait-free implementation is also lock-free, but with the stronger guarantee that there will be no starvation.

3.3 Hardware support for concurrent data structures

Herlihy [17] observed that designing non-blocking implementations for many simple sequential data structures is not possible using classical non-universal synchronization primitives (test&set, read, write, fetch&add and swap). Therefore, lock-free or wait-free implementations can only be constructed using hardware operations that atomically combine a load and a store operation. The most popular of these operations are *compare & swap* (CAS) and *load-linkedstore-conditional*.

Traditionally, concurrent shared data structures are constructed for shared memory programming, with a uniform address space and all threads having a uniform view of memory. Additionally, most non-blocking implementations of these data structures are developed for modern processors that provide atomic (load&store) synchronization instructions. However hardware that supports shared memory models does not scale very well with an increasing number of cores, and traditional cache-coherence is hard to achieve on new many-core architectures.

Consequently, from a hardware perspective, it is preferable to have a message-passing model, despite shared-memory programming being more desirable and very convenient at the software level. This in addition to other factors mentioned, motivated us to develop shared-memory constructs on hardware with very low latency message-passing support like the SCC.

A good starting point for developing shared data structures on SCC is the slogan, "Do not communicate by sharing memory; instead share memory by communicating", taken from the GO programming language. To this effect, we develop data structures to which concurrent access is explicitly coordinated by message passing.

3.4 FIFO Queues on SCC

A FIFO queue is a collections data structure that offers *first-in-first-out* fairness. This data structure provides an *enqueue* operation which adds an item to the end of the queue or the tail and a *dequeue* operation that removes and returns the item at the front of the queue, also referred to as the head. A concurrent queue should have enqueue and dequeue operations whose execution history maintains the FIFO ordering of the data structure. Thus, a concurrent queue is desired to be linearizable to a sequential execution with the same method calls.

In this thesis, we developed three implementations of a FIFO queue on the SCC inspired by ideas from shared-memory programming and distributed data structures. One is an implementation of the blocking two-lock algorithm by Michael and Scott [16] which provides fine-grained synchronization by using a distinct lock for the head and for the tail pointers of the queue. The other two algorithms are developed with ideas taken from distributed data structures, having the data structure owned by one node and all other nodes requesting exclusive ownership to an item of the data structure at a time.

We utilize the shared off-chip memory as statically mapped memory. This is because it is mapped by each executing process into its address space and not allocated by the operating system. This memory is mapped to different virtual addresses for each process. Therefore, we use memory offsets instead of actual virtual-memory addresses when passing pointers to an address in memory from one core to another.

On receiving the offset; the core has to add the offset to the base address of its mapping so as to calculate the actual location of the object in consideration. This technique is adopted from cluster-programming because although the SCC has shared memory, it does not support the shared memory model with a uniform virtual address space across all cores.

3.5 Double-lock Queue

For the Lock based queue, we implement the MCS two-lock queue as a static memory linked-list or array stored in shared off-chip memory. We store pointers to tail and head indexes in the fast access message-passing buffer. The algorithm uses two separate locks for the head, and tail, thus allowing one enqueue operation to proceed in parallel with a dequeue operation.

We implemented the locks using test&set registers; the SCC has 48 test&set registers with each core having one of the registers. We memory map the registers as non cacheable memory and use the semantics of the register to create a spin lock. Reading the register returns a zero if no process has previously read and not reset the register, otherwise a one is returned. A register write by any core causes it to reset. Figure 3 shows implementations for acquiring and releasing the lock built using a memory mapped test&set register.

Figure 3: Lock implementation using a memory mapped test&set register

```
int lock(lockaddr) {
  while (!((*(lockaddr)) & 0x01));
  return 0;
}
```

(a) acquire lock

```
int unlock(lockaddr) {
  *(lockaddr) = 0x0;
  return 0;
}
```

(b) release lock

The cores experience different latencies when accessing on-chip MPB or off-chip memory. With this in mind, we sought to increase parallelism by limiting the work inside the critical section to updating head and tail pointers. The thread holds the lock only for the duration of updating the respective pointer, then releases the lock. This is achievable because statically allocated memory holds the data structure. The statically allocated memory improves performance by reducing the number of memory allocations, and avoiding consistency issues that may be incurred while reclaiming dynamically allocated memory.

Consequently, the algorithm may be blocking and not lock-free, but it allows for multiple concurrent memory reads/writes in the high latency off-chip shared memory, thus having multiple enqueuers and dequeuers make progress at the same time. We also observed that; as the platform does not support cache coherence, the spin locks bear no additional cost typically associated with cache-coherence protocols.

3.5.1 Queue Methods

Figure 5 presents commented pseudo-code of the two-lock queue and its methods. To enqueue an item to the list, a process/core acquires the *enqueue_lock* or *tail_lock* and reads the current tail pointer. In this algorithm, the tail pointer holds the offset of the next empty index in the array and not the last added item; thus it refers to a place holder in the queue. The enqueuer increments the tail pointer and writes back the new value

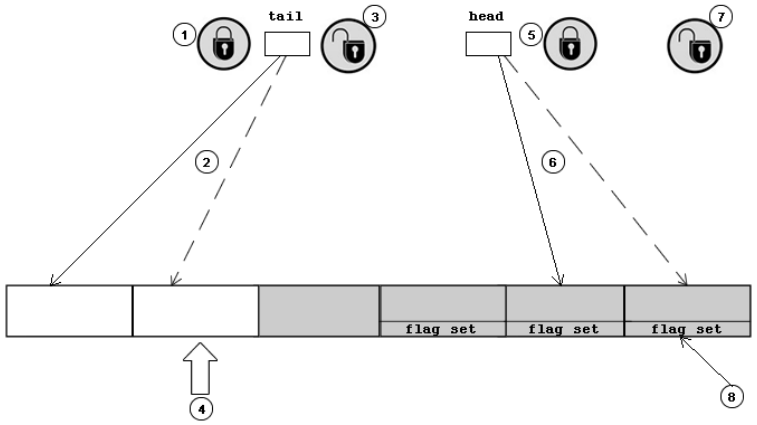


Figure 4: enqueue() and dequeue() methods of the two-lock queue. The enqueue() method acquires the enqueue lock in step 1, reads and updates the tail pointer to the next offset index (step 2), then releases the lock. The enqueuer proceeds to add the node to the memory pointed to by the offset (step 4). To dequeue a node from the list, a node acquires the dequeue node (step 5). On acquiring the lock, the node reads the head pointer (step 6), checks that it does not point to the same location as the tail. The dequeuer, then updates the head pointer and releases the lock (step 7). In step 8, the dequeuer spins on the node flag to confirm that the node has been enqueured before dequeueing it.

of the pointer to the Message-Passing Buffers (MPB). Then releases the lock; allowing other processes to proceed and contend for the enqueue_lock. After releasing the lock, the enqueuer proceeds to add the data item to the address location read in step 2 of Figure 4 to the shared off-chip memory. To calculate the virtual address in the core's virtual memory, the offset read is added to the starting address of the off-chip memory as mapped on the enqueueing core.

```
struct  node_t     { value: type_t; flag: uint; };
struct queue_t     { head_offset: uint; tail_offset: uint;};

init_queue(queue_t* Q)
    Q->head_offset = Q->tail_offset= NULL
    write_to_mpb(Q)                                  # Store queue pointers in MPB
    enqueue_lock = dequeue_lock = FREE


void enqueue(queue_t* Q, data_type value)

E1:  acquire_lock(enqueue_lock)                      # Acquire enqueue lock to read tail offset
E2:       Q = read_from_mpb()                        # Fetch queue pointers from MPB
E3:       mem_offset = Q->tail_offset                # Read current tail offset
E4:       Q->tail_offset += LINE_SIZE
E5:       write_to_mpb(Q->tail_offset)
E6:  release_lock(enqueue_lock);                     # Release lock
E7:  node_shm = (node_t*) (shm_startaddr+mem_offset) # Locate virtual address location using tail-offset
E8:  node_shm->value = value;                        # Add enqueued value to memory
E9:  node_shm->flagset = SET;                        # Set flag to indicate completed enqueue
E10:  end


data_type dequeue(queue_t* Q)

D1:  acquire_lock(dequeue_lock)                      # Acquire dequeue lock to read Head offset
D2:       Q = read_from_mpb()
D3:       if ( Q->head_offset == Q->tail_offset) then # Is the queue empty?
D4:       release_lock(dequeue_lock)                 # Release dequeue lock
D5:       return NULL                                # Return queue was empty
D6:       endif
D7:       mem_offset = Q->head_offset                # Read Head offset
D8:       Q->head_offset += LINE_SIZE;
D9:       write_to_mpb(Q->head_offset)
D10:  release_lock(dequeue_lock);                    # Release dequeue lock
D11:  node_shm = (node_t*) (shm_startaddr+mem_offset)
D12:
D13:  while (node_shm->flagset!= SET) do
        ;
       end
                                                     # Wait for completion of enqueue operation
D14:
D15:  value = node_shm->value                        # Read node value
D16:  node_shm->flagset = NOP                        # Clear memory for reuse
D17:  return value                                   # Successful, return value
```

Figure 5: Two-lock concurrent queue operations

When designing the algorithm, we realized that updating the pointer without adding the data item to memory could lead to a situation where a dequeuer reads a null value from a node that is yet to be added to the list. An item is logically added to the list as soon as the tail pointer position is incremented and written back, albeit the data item not having been added to the shared off-chip memory. To prevent this from happening, we added a flag bit to the data item to indicate that the data item has absolutely been enqueued to the list.

To dequeue a data item; a process acquires the *dequeue_lock*, reads the queue_t pointer, and checks if the head_offset is equal to the tail_offset. If so, then the queue is momentarily empty, and the dequeuer releases the lock returning a -1 to the caller which indicates an empty queue. On the other hand, if the queue is not empty, the dequeuer reads the current queue's head offset. Then increments the head offset pointer, writes the new value back to MPB and releases the *dequeue_lock*. After releasing the lock, proceeds to check the node's dirty flag to confirm that the dequeued data item has been enqueued previously. In case the flag is not set, the dequeueing processes spins on the flag until its set. When the node's flag is set, the dequeuer can proceed and read the value of the node, returning this value as the dequeued item.

This spinning on the dirty flag during a dequeue operation adds more blocking to an already blocking data structure, and a delay or failure of an enqueueing processes could lead to indefinite spinning of a dequeueing process.

3.6 Message-passing based queues

To avoid the use of blocking synchronization primitives like locks, we created a queue where message-passing controls concurrent access. In this abstraction, one node (the server) owns the data structure, and all other nodes (the clients) request exclusive ownership to elements of the data structure at a given time.

The queue data structure resides in off-chip shared-memory, with the server holding pointers to the head and tail of the queue in its private memory. If a core wishes to add an item to the queue, it must request for the position of the queue tail from the server. A node wishing to remove an item from the list also acquires the head position in a similar procedure. The server handles Enqueue/Dequeue requests in a round-robin fashion, and correspondingly updates to the queue pointers as only this core has write access to these pointers.

We implemented the message-passing mechanism utilizing receiver side placement (i.e. the sender writes the data to the receiver's local MPB buffers) and notification flags to perform flow control. The flow control is necessary to prevent a core from processing an already processed message or overwriting unread messages on the receiver end.

We use notification flags and polling to discover arrival of new messages and also enable reuse of memory after processing a message. Receiver side placement of the messages also makes polling more efficient, because all the cores spin on local memory and reduce network traffic while polling for incoming messages. We also realized that, with one server handling requests from multiple clients, this arrangement results in a

gather pattern [2]. Therefore, the server's sequential polling is faster on local memory; as the clients execute requests in parallel. This in essence also supports the decision to implement receiver side placement model of message-passing.

To manage the message-passing buffers, the server allocates slots in its local MPB memory, one for each of the participating cores. Therefore, each client core can have only one outstanding request at a time. The server also requires a slot on the client core's local MPB memory for writing responses for the requesting core.

On initialization, each core maps the shared memory into its address space in non-cached (NCM) mode. The cores map the memory to different virtual addresses, making the exchange of address pointers impossible. Instead, different cores exchange offsets from the starting address of the shared memory.

3.6.1 Queue Methods

Figures 6 and 7 present commented pseudo-code for the server and client operations of the data structure. In this implementation, the server/manager core allocates message slots in its MPB, one for each client core. Each core has fixed message slots; therefore a core writes request messages to the server into the allocated slot. The server polls on the slot flags for incoming requests. The server also initializes the queue in shared memory by clearing the allocated memory and creating queue pointers to the data structure. The server stores these pointers in its private memory and no other core performs updates on these pointers. It allocates a separate set of slots in the local MPB on each client core, and these slots used to write responses from the server to the client cores. After each request, the client core proceeds to spin on this receive buffer for a response from the server.

A core wishing to enqueue an item to the list, sends a request for the queue head-offset to the server, as illustrated in Figure 6 (EC 3). After request, the core spins on a slot in its local MPB for a response from the server. On receiving the enqueue request message(Figure 6), the server updates the queue tail pointer and returns the previous offset to the requesting core. In this implementation, the tail-offset always refers to the next empty slot in the shared memory. Statically allocated memory is the main motivation for this use of the tail pointers.

After a response message from the server, the enqueueing client node proceeds to add the data to the memory location pointed to by the returned offset. To complete the enqueue operation, the process sets the flag on the memory slot to true.

To dequeue a data item, the core requests for the head offset instead of the tail offset as above. If the queue is empty, the server returns a negative offset to notify the dequeuer of an empty queue. Else; if the queue is not empty, the server responds with the current head offset and updates the head pointer accordingly. However, as noted in the two-lock implementation, in this mechanism we also run into a situation where a memory slot assigned to the dequeuer is yet to be filled added by the enqueuer. We maintain the use of dirty flags to indicate that the enqueuer has successfully added the data item to memory

```
struct  node_t     { value: type_t; flag: uint; };
struct queue_t     { head_offset: uint; tail_offset: uint;};
struct  request_t    { memsize: size_t; CMD: uint; };
struct response_t     { mem_offset: uint; CMD: uint; memsize: size_t;};

init_queue()
    request_addr = map_memory();                    # Request slots
    response_addr = map_memory();                   # Response slots


void enqueue(request_t* REQ, data_type value, request_addr)
EC1:  REQ->memsize = sizeof(int)
EC2:  REQ->cmd = ENQ
EC3:  write_to_mpb(request_addr, REQ)               # Send request to server
EC4:  wait_for_response(response_addr, response_t* RES )
    # poll server for response
EC5:
EC6:  node_shm = (node_t*) (shm_startaddr+RES->mem_offset)
    # Use offset in response to find assigned memory slot
EC7:  node_shm->value = value;
EC8:  node_shm->flagset = SET;                      # Set flag to indicate completed Enqueue operation
EC9:  clear_memory(response_addr)
EC10:  end


data_type dequeue(request_t* REQ, request_addr)
DC1:  REQ->memsize = sizeof(int)
DC2:  REQ->cmd = DEQ
DC3:  write_to_mpb(request_addr, REQ)
DC4:  wait_for_response(response_addr, response_t* RES )
DC5:  if ( RES->mem_offset == NULL) then
DC6:       return NULL                              # Queue is empty
DC7:       endif
DC8:  node_shm = (node_t*) (shm_startaddr+mem_offset)
DC9:  while (node_shm->flagset!= SET) do
        ;
       end
                                                    # Wait for completed Enqueue operation
DC10:
DC11:  value = node_shm->value
DC12:  node_shm->flagset = NOP
DC13:  clear_memory(response_addr)                  # Clear message buffers
DC14:  return value                                 # Return dequeued value
```

Figure 6: Client Operations

If the server responds with the the head-offset, the dequeuer spins until the dirty flag on the memory slot is true, and the data item can be dequeued from the list. However, the process should clear the response message buffer so that it does not read stale data upon issuing another request before returning from this method call.

```
init_queue(queue_t* Q)
    Q->head_offset = Q->tail_offset= NULL        # Initialize queue pointers
    client_response_addrs[No_of_cores]
                                                 # Map client core-local MPB slots for passing responses
    for ( core=0;core<No_of_cores; core++)
    do
    client_response_addrs[core] = map_response_addr(core)
    end
    req_startaddr = map_memory();                # Map client request memory buffers


void server(queue_t* Q)
S1:   while ( i<No_of_cores) do
S2:         core_req_addr = req_startaddr+(i*sizeof(request_t))
    # Core i request location
S3:         read_from_mpb(core_req_addr,REQ)              # Fetch request from MPB slot
S4:
S5:         if ( REQ->cmd == ENQ) then                    # Enqueue Request
S6:              RES->mem_offset = Q->tail_offset;        # Add tailoffset to response message
S7:              RES->memsize = LINE_SIZE;
S8:              Q->tail_offset ← (Q->tail_offset + LINE_SIZE)
    # Advance tail pointer
S9:              RES->cmd = OK;
S10:              write_to_mpb( client_response_addrs[i], RES);
    # Return response message to core
S11:             clear_memory(core_req_addr)
S12:        else if ( REQ->cmd == DEQ) then
S13:             if ( Q->head_offset == Q->tail_offset) then
S14:                  RES->mem_offset = NULL;
S15:             else                                      # Queue not empty
S16:                  RES->mem_offset = Q->head_offset;
S17:                  Q->head_offset ← (Q->head_offset + LINE_SIZE)
S18:             endif
S19:             RES->cmd = OK;
S20:             write_to_mpb( client_response_addrs[i], RES);
S21:             clear_memory(core_req_addr)
S22:        endif
S23:        i ← (i + 1)%No_of_cores                       # Poll message buffers
S24:  end
```

Figure 7: Server operations

The use of flags to indicate completion of an enqueue process may result in blocking of the dequeue method. This blocking could be indefinite if an enqueue-ing process fails before performing the addition of a node to the allocated memory location.

We eliminate this use of flags by having the enqueuer notify the server after writing the data item to shared memory. We add an acknowledgement step to the enqueue procedure, hence the dequeue operation is executed only on nodes successfully appended to the data structure.

3.6.2 Message-passing with Acknowledgments

Figures 8 and 9 show the pseudocode for the client and server operations with an added acknowledgment phase on enqueueing an item to the data structure. In this procedure, the server maintains a private queue of pointers to locations in the shared memory. This queue stores pointers to memory slots successfully enqueued with data nodes and the server notified of the completion of the enqueue process. The ordering of pointers in this private queue constructs the FIFO ordering of the data structure implemented in this algorithm.

The server also maintains a pointer to the next empty array index in the shared off-chip memory; however the head of the queue data structure could lag behind this pointer.

Adding of an item to the queue proceeds as follows; a client requests for a pointer to the tail of the data structure in shared memory. When the server receives this request, it returns the head-offset to the requesting core and updates the pointer to next empty array index. If allocated a slot in memory, the enqueuer writes the data to memory and sends an acknowledgement to the server to notify the server about the completion of the enqueue process. The server adds the memory offset carried in the acknowledgment message to the private queue of memory.

When a dequeuer requests for the head offset, the server dequeues the head of its private logical queue of pointers to shared memory and returns this pointer to the dequeuer. As this private queue holds pointers to memory locations to which enqueue operations were completed and acknowledged by the enqueueing cores, the dequeuer can proceed and dequeue the data item from shared memory without blocking. This implementation adds an extra message exchange phase, but it solves problems arising from failed client processes during an enqueue method call.

```
struct node_t     { value: type_t; flag: uint; };
struct queue_t    { head_offset: uint; tail_offset: uint;};
struct request_t    { memsize: size_t; CMD: uint; };
struct response_t    { mem_offset: uint; CMD: uint; memsize: size_t;};

init_queue()
    request_addr = map_memory();
    response_addr = map_memory();


void enqueue(request_t* REQ, data_type value, request_addr)

C1:  REQ->memsize = sizeof(int)
C2:  REQ->cmd = ENQ
C3:  write_to_mpb(request_addr, REQ)
C4:  wait_for_response(response_addr, response_t* RES )
C5:  node_shm = (node_t*) (shm_startaddr+RES->mem_offset)
C6:  node_shm->value = value;
C7:  node_shm->flagset = SET;
C8:  REQ->cmd = ACK                          # Send notification of completed enqueue operation
C9:  REQ->mem_offset = RES->mem_offset       # Location of enqueued node
C10:  write_to_mpb(request_addr, REQ)
C11:  clear_memory(response_addr)            # Clear message response buffers
C12:  end


data_type dequeue(request_t* REQ, request_addr)

DA1:  REQ->memsize = sizeof(int)
DA2:  REQ->cmd = DEQ
DA3:  write_to_mpb(request_addr, REQ)
DA4:  wait_for_response(response_addr, response_t* RES )
DA5:  if ( RES->mem_offset == NULL) then
DA6:        return NULL                      # Queue is empty
DA7:        endif
DA8:  node_shm = (node_t*) (shm_startaddr+mem_offset)
DA9:  value = node_shm->value
DA10:  node_shm->flagset = NOP
DA11:  clear_memory(response_addr)
DA12:  return value                          #
```

Figure 8: Client Operations with acknowledgements

```
init_queue(queue_t* Q,queue_t* local_queue)
    Q->head_offset = Q->tail_offset= NULL
                                                    # Queue of pointers to enqueued nodes
    local_queue->head=local_queue->tail = NULL client_response_addrs[No_of_cores]
    for ( core=0;core<No_of_cores; core++)
    do
    client_response_addrs[core] = map_response_addr(core)
    end
    req_startaddr = map_memory()


void SERVER(queue_t* Q, queue_t* local_queue)
S1:  while ( i<No_of_cores) do
S2:      core_req_addr = req_startaddr+(i*sizeof(request_t))
S3:
S4:      read_from_mpb(core_req_addr,REQ)
S5:      if ( REQ->cmd == ENQ) then
S6:          RES->mem_offset = Q->tail_offset
S7:          RES->memsize = LINE_SIZE
S8:          Q->tail_offset ← (Q->tail_offset + LINE_SIZE)
S9:          RES->cmd = OK
S10:          write_to_mpb( client_response_addrs[i], RES)
S11:          clear_memory(core_req_addr)
S12:
S13:     else if ( REQ->cmd == DEQ) then
S14:          if ( local_queue->head == NULL) then    # Queue empty
S15:              RES->mem_offset = NULL
S16:          else
S17:              qnode_t *temp = local_queue->head
S18:              RES->mem_offset = local_queue->head->shm_offset
S19:              local_queue->head = local_queue->head->next
S20:              free(temp)
S21:          endif
S22:          RES->cmd = OK;
S23:          write_to_mpb( client_response_addrs[i], RES);
S24:          clear_memory(core_req_addr)
S25:
S26:     else if ( REQ->cmd == ACK) then
S27:          qnode_t* queue_node = new_node() # Add entry to queue of pointers to filled memory slots
S28:          queue_node->next = NULL;
S29:          queue_node->shm_offset = REQ->mem_offset;
S30:          if ( local_queue->tail == NULL) then
S31:              local_queue->tail = local_queue->head = queue_node;
S32:          else
S33:              local_queue->tail->next = queue_node;
S34:              local_queue->tail = queue_node;
S35:          endif
S36:          clear_memory(core_req_addr)
S37:     endif
S38:     i ← (i + 1)%No_of_cores
S39: end
```

Figure 9: Server operations with acknowledgements

4.0 Correctness

In this section, we show that implemented algorithms are correct by providing proofs that they satisfy desired safety and liveliness properties of a linearizable FIFO queue. Safety properties ensure that the algorithms maintain correct queue semantics while the liveliness property guarantee that in case of concurrent enqueue or dequeue method calls; one call will make progress in finite time. Therefore by definition, safety is guaranteed by conditions for consistency or correctness while liveness is determined by the progress in finite time.

4.1 Safety

Correctness of a concurrent object is usually evaluated by its equivalence to a sequential implementation of the same object[6]. Linearizability is a correctness condition that; for all interleaved operations of a concurrent execution, there is a correct equivalent sequential execution of the same method calls that preserves the real-time ordering of all method calls.

Linearizability is formally defined as a correctness condition that every method call appears to execute atomically at some point between its invocation and the response. The requirement that real-time ordering of all operations is preserved relies on the notion that operations are ordered in time. In the formal definition of linearizability, concurrent executions are modeled as a history of invocations and responses. The history asserts a partial ordering $<_H$ or "happened-before" relation on operations, the invocation always precedes the response. The partial ordering also applies to "real-time" ordering of operations in the history. Concurrent operations cannot be expressed with totality ($a <_H b$ or $b <_H a$); thus operations are modeled using partial ordering of events.

For linearizabilty of concurrent executions, however, the history should be extended to achieve total ordering of different operations. Identifying linearization points for each method call is one way of showing that an implementation of a concurrent object is linearizable. A linearization point is the single moment in time where the method call seems to "take effect" [24] and the effect is evident to other concurrent method calls on the same object. Adding linearization points to a history of a concurrent execution allows us to model the execution as a sequential execution of the same operations. This is normally termed as linearization of a concurrent execution.

4.2 Liveness

Liveness is a measure of the progress guarantee that an implementation provides in finite time. Concurrent data structures are categorized as either blocking, where a slow or delayed thread can delay others indefinitely, or non-blocking, where at least one thread is guaranteed to make progress despite delayed or slow processes operating on a shared data structure. Non-blocking implementations are further classified as either Lock-free or Wait-free.

A lock-free implementation of a method guarantees that, in finite time, at least one method call completes its execution in a finite number of its steps. On the other hand, an implementation is wait-free if it guarantees that every method call finishes execution in a finite number of its own steps. Wait-free implementations are lock-free, but often result in reduced performance and are certainly more difficult to implement correctly. Accordingly, implementations often offer lock-free progress guarantees instead of wait-free.

4.3 Proof of Safety

FIFO queue algorithms are correct if they satisfy the following requirements:

- If a dequeue method call returns an item x, then it was previously enqueued, thus $Enqueue(x) \rightarrow {}^{3} Dequeue(x)$.
- If an item x is enqueued at a node, then there is only one dequeue method call that returns x and removes the node from the queue.

We use the correctness proof base on induction described in the MS-Queue [16] to prove that the algorithms maintain the following safety properties:

1. The queue is a list of nodes that are always connected.
2. New items are only added to the end of the list.
3. Items are only removed from the beginning of the list
4. Tail always points to the next empty slot in the list.
5. Head always points to the first element in the list.

4.3.1 Lock-based algorithms

Property 1. *The queue is a list of nodes that are always connected.*

In the algorithms presented, we assume that the queue is an array of contiguous statically allocated memory slots. We allocate shared off-chip memory as static memory with consecutive memory blocks forming array elements. Thus, the queue items are always connected, and we can use increasing indexes to traverse from the beginning of the array to the end.

Property 2. *New items are only added to the end of the list.*

Items are only inserted at the index pointed to by the tail pointer. This pointer references the next empty position at the tail end of the queue and is protected by the tail lock. There, enqueue method calls add items only to the end of the list.

[3] total order or happened before relation of events

Property 3. *Items are only removed from the beginning of the list.*

Only nodes referenced by the head pointer are dequeued from the list. As from Property 4, the head always points to the first item in the list. Therefore, items are only removed from the beginning of the queue.

Property 4. *Head always points to the first node in the list.*

The head-offset is only updated by the process holding a lock. On dequeuing, the head is advanced to refer to the next item in the list, and the previously indexed element removed from the list. The queue is empty when the head points to NULL address.

Property 5. *Tail always points to the next empty slot in the list.*

As with the head pointer, the tail is also only updated by an enqueuer holding the enqueue lock. However, we chose to have the tail pointer reference the next empty index because we have a statically allocated list of entries. On using up an empty list entry; the enqueuer advances the tail pointer to refer to the next empty slot that can be used by another enqueue method call.

4.3.2 Message-passing based algorithms

Property 1. *The queue is a list of nodes that are always connected.*

The queue items are always connected as in property 1 for lock-based implementation.

Property 2. *New items are only added to the end of the list.*

Only the server advances the tail pointer which always refers to the next empty slot at the tail end of the queue (property 5). The server allocates the index of the empty slot to a requesting client and advances the tail pointer.

Property 3. *Items are only removed from the beginning of the list.*

Nodes are only deleted if pointed to by the head returned by the server. The server returns the position of the element at the beginning of the list, and only the server advances this pointer.

Property 4. *Head always points to the first item in the list.*

The server advances the head pointer after responding to a dequeue request, ensuring that it points to the next node in the list. The node pointed to is considered the first item in the list because the previous item is removed by the dequeuer.

Property 5. *Tail always points to the next empty slot in the list.*

The tail always points to the next empty slot because its only advanced on using up a list node. The server ensures that the tail never lags the end and that it never points to a node that is being deleted by another process.

Lemma 1. *If a dequeue method call returns an item x, then it was previously enqueued, thus Enqueue(x) → Dequeue(x).*

Proof. The head always lags the tail, and an array of contiguously stored elements makes up the queue. Therefore, if a dequeue method call returns an item, then the item was previously enqueued. In implementations that use a flag to indicate completion of an enqueue operation, the dequeue method call only returns a value if another process or method call successfully set the element's flag as enqueued. □

Lemma 2. *If an item x is enqueued at a node, then there is only one dequeue method call that returns x and removes the node from the queue.*

Proof. In the lock-based algorithm, a dequeue operation only succeeds to remove an item from the list after exclusively holding the dequeue lock and thus ensuring that no other dequeuer removes the same item.

In the message-passing based algorithms, dequeue operations are all coordinated by the server. The server allocates which array index is to be dequeued, and only the server advances the head pointer after each dequeue method call.

□

Theorem 1. *The presented FIFO queues are linearizable.*

Proof. For the FIFO queue, if by ordering the concurrent method calls by their linearization points we achieve a correct sequential execution of the same method calls, hen the queue implementation is linearizable to an abstract sequential FIFO queue. The implemented algorithms are linearizable as we can identify linearization points during each method call.

For the two-lock implementation, the locks create critical sections, which serve as linearization points for the method calls. For the implementations that are based on message-passing, the server serializes the method calls. This occurs because the server handles requests in a round-robin manner thus they appear sequential in nature.

Properties listed in section 4.3 show that queue pointers reflect a stable state of the queue. From Lemmas 1 and 2, the enqueue method of an item always precedes its dequeue and the order in which items are dequeued is identical to the order in which items are dequeued. Therefore, the concurrent FIFO queues are linearizable to abstract sequential queues. □

4.4 Liveness

In this section, we discuss the liveness properties of the concurrent queue implementations presented. Liveness properties are guarantees that there will be progress at system level or even better at per process level within a finite time. One popular model of proving system progress is to show that, there are no indefinite delays resulting from a halted or delayed process, and that all loops within the implementation terminate infinitely often.

The message-passing based queue algorithms utilize a *Server-Client* model of interaction. In this model, all client progress is dependent on the server, because of the request-response interactions to ascertain the values of the head and tail pointers. This dependence of all clients on the server implies that any system progress is also reliant on the server. Therefore, we discuss liveness properties with a stringent assumption that, *the server process can not be halted or indefinitely interrupted by system scheduler or other processes running on the server core* i.e. fault-free server.

Communication between the server and clients can be modeled using LogPmodel by Culler et.al [25]. This model summarizes a ServerClient communication pattern using a few parameters; these are, the send overhead O_s, receive overhead O_r and the server processing time S. The *send overhead* O_s is the time required to assemble the message and write the request to the message-passing buffers. Because of the implemented a remote write, local-read pattern of message passing, the write overhead dependent on the distance from the destination. The *receive overhead* O_s denotes the waiting time from when a read request is issued until data is returned from the buffers. This overhead also includes cost of disassembling the message. Receives are local operations, therefore independent of the distance from the server.

Server processing time S_t describes the waiting time before the server handles a client request plus the cost of handling the request. The server handles requests in a round-robin fashion, thus S_t is a function of the number of participating cores $\mathrm{O}(n)$.

Lemma 3. *Server responds to every client request in finite time.*

Proof. Each client can have only one outstanding request to the server. The latency of each request is the time for a client to assemble the request, send the request to the server, plus the delay for server to process the request and return a response to the client. This latency also includes the time the client takes to disassemble the response message. The roundtrip time for a single request to the server is approximately $T_{req} = S_t + 2(O_s + O_r)$. All the parameters in the expression for T_{req} are bounded in time, therefore a fault-free server responds to every client request in finite time.

□

4.4.1 Message-passing based algorithm

The enqueue method call of the message-passing based algorithm loops on line $EC4$ (figure 6) while it polls for a memory slot allocation from the server. The arbitration of multiple requests on the server is round-robin. Therefore, the number of steps an enqueue operation loops is dependent on the number of requests processed by the server. With the specified assumption that the server is never halted, a delay in the loop that is greater than the 'send-receive' message passing overhead is a result of the server processing requests from other processes. If there is significant delay inside the loop, then other processes complete operations during the same duration. From Lemma 3, it is guaranteed that a response is eventually returned by the server in finite time, and

the loop is exited. Therefore, the enqueue method call terminates in finite despite other enqueue operations that are also dependent on the server.

The dequeue operation also loops in step $DC4$ (figure 6) polling the server for the position of the queue head. As mentioned above, round-robin arbitration is the only additional source of delay for the execution of this loop. If the server is never halted or indefinitely interrupted, this loop is bound to exit in finite time.

However, in this implementation we use flags to notify about successful enqueue operations. The dequeue method call loops again after receiving the pointer to the queue head. The loop at line $DC9$, is for the dequeuer to check that an item was successfully enqueued to the assigned slot, before reading the nodes value. The dequeue method call only fails to terminate this loop, if the enqueue failed to successfully add data items to the allocated slot. Consequently, the dequeue operation is blocked with respect to the corresponding enqueue operation. Other operations will succeed despite the possibility of having blocked dequeue operations. More specifically, this implementation can have a slow enqueuer delay a fast dequeuer and the dequeuer can be delayed indefinitely in case the enqueue fails to complete. But this blocking is only at enqueue-dequeue process pair level and does not deter progress of other processes i.e. system progress.

To the best of our knowledge, there is no term to describe this phenomenon where an operation is blocking with respect to a single operation. We coined the term "pairwise blocking" to describe this behavior.

Pair-wise blocking

> The behavior that an operation is blocking with respect to at most one other operation and the progress of other operations is not affected. In the context of a queue data structure, an algorithm is pair-wise blocking if an enqueue operation can block the corresponding dequeue operation without blocking the progress of other enqueue or dequeue operations.

Theorem 2. *Message-passing based algorithm is pairwise-blocking.*

Proof. In this algorithm, a data item is logically added to the list as soon as the server advances the head pointer and returns a response to an enqueue request. The server proceeds to handle other requests as the previous enqueuer adds data the the assigned node of the list. On receiving a dequeue request, the server responds with the header position without prior knowledge as to whether the enqueue at the header position succeeded.

The dequeue process reads flags (line $DC9$) at the memory location to confirm that the item was successfully added to the list. It continuously polls the flags waiting for the the enqueue method call to terminate. If the enqueuer is slow, it delays a fast dequeuer and it case it fails, the dequeuer is delayed indefinitely. This has been specified as pairwise blocking. Therefore, we appropriately classified this algorithm as pairwise blocking. □

4.4.2 Message-passing based algorithm with acknowledgments

To eliminate the possibility of infinitely blocked operations like the dequeue mentioned previously, we added acknowledgments after successful enqueue operations. In message-passing with acknowlegments, the enqueue method call on lines $C8 - C10$ (Figure 8), notifies the server about a successfully adding a node to the data structure. A node is logically added to the data structure, if the enqeueur sends an acknowledgement to the server. In this way, the server maintains record of added nodes, and only allocates successfully enqueued memory slots to a dequeuer.

Theorem 3. *Message-passing based algorithm with acknowledgements is non-blocking.*

Proof. From Lemma 3, the loops in this algorithm are only dependent on the server and are non-blocking with respect to other operations. Adding acknowledgments to the enqueue procedure provides a solution to the blocking of dequeue operations that is dependent on the corresponding enqueue operation. Therefore, the message-passing based queue algorithm with acknowledgments is non-blocking.

□

5.0 Performance

To investigate the performance, fairness and scalability of the algorithms described in 3.4; we implemented the algorithms on the Intel SCC. An SCC system running cores at 533MHz, mesh routers at 800MHz and 800MHz DDR3 memory was used to perform the experiments. All algorithms and testing code are written in C and compiled with the *icc* targeted for the P54C architecture. The version of icc only validated for GCC version 3.4.0 was used, and experimental Linux version 3.1.4scc Image loaded on each of the 48 the cores. In the experiments, we treat each core as a single unit of execution and have only one application thread/process running per core.

In order to synchronize the execution of program code on different cores and be able to model contention conditions, we implemented a barrier to run at the start of every execution. The barrier implementation forces the cores to start executing the algorithms at roughly the same time after initialization of memory buffers and configuration registers. We use atomic increment counters in the FPGA to construct an increment barrier. Each counter has a pair of registers; initialization counter register and the atomic counter register. Before the start of the program, the counter is initialized to 0. Every running process reads the atomic counter register to increment the counter. Then, the process spins on the initialization counter register until the value is equal to the number of executing cores.

We run the algorithms for 600ms per execution to measure throughput on each core. Each executing core chooses randomly an enqueue or a dequeue operation with equal probability. To achieve this random selection of operations, we assigned a string of randomly generated bits with a probability of 0.5 to each core. During execution, the core reads the string bit by bit, and the read bit determines which operation is performed by the core. We maintain the same bit string per core across the execution of different algorithms and changing system configuration.

Each execution was replicated 12 times and the resulting throughput values analyzed to have a $\alpha = 0.05$ level of significance (Student's t-test). The averages of the replicated performance figures are what we present as the results.

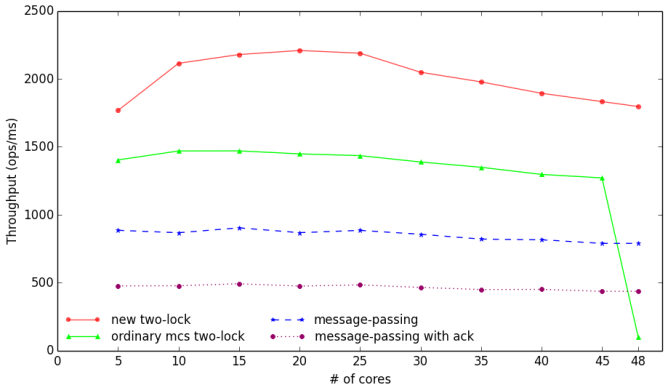
We varied contention levels by adding "other work" or "dummy executions" to the experiments. The dummy executions consist of 1000 or 2000 increment operations. Adding dummy work to the experiment serves a purpose of giving a more realistic view of method calls to a queue by an application. We also randomized the execution and duration of the dummy work on the different cores, as we compared the algorithms under different contention levels. To randomize the execution time of dummy work, we generated a string of numbers (0,1,2), varying the distribution of the numbers in the string. Depending on the value read from the string, a core decides on how much dummy work to perform after an enqueue or dequeue operation.

We measured system throughput as the number of successful enqueue and dequeue operations per millisecond. While, fairness as introduced in [26] was used to analyze how well various cores perform in relation to the system as a whole and other participating cores. This in turn, helped to understand the level of starvation-freedom that can be

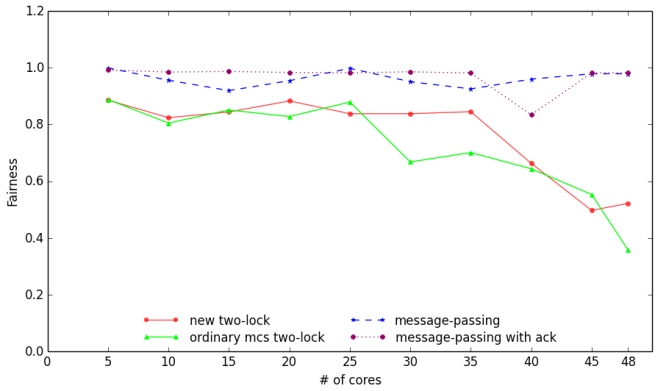
achieved by the different algorithm implementations. We calculate fairness using the formal definition [26]:

$$fairness_{\Delta t} = min \left\{ \frac{N.min\left(n_{i_{\Delta t}}\right)}{\sum_i n_{i_{\Delta t}}}, \frac{\sum_i n_{i_{\Delta t}}}{N.max\left(n_{i_{\Delta t}}\right)} \right\} \quad (1)$$

where $n_{i_{\Delta t}}$ is the throughput (operations performed in interval Δt) of a core i and N is the number of participating cores. Fairness values approximating 1 indicate fair system behavior while values close to 0 indicate favored performance by some cores relative to others. Fairness value of 0 indicates starvation[4] of one or more cores.



(a) System throughput



(b) System fairness

Figure 10: Throughput and fairness at high contention

Figure 10 presents the system throughput and the fairness of the different algorithms. For comparison of the presented approach to two-lock queue data structure, we also implemented the original MS two-lock concurrent queue [16] on the SCC.

[4]Starvation - A core fails to complete any operation during a run of the experiment.

Figure 10(a) shows the system performance for the various algorithms. From this figure, we observe that the new implementation of the MS two-lock concurrent data structure with early release of the lock achieves higher system throughput than the original implementation at high contention. The figures also show that message-passing based algorithms achieve much less system throughput, however, with very high level of fairness to all participating cores. From figure 10(b), fairness of the lock-based algorithms drops as we increase the number of cores due to increasing contention for the head and tail pointers. This observation led us to investigate the performance and fairness of the system as we vary the placement of the locks on the chip. We present the results of this experimentation later in this section.

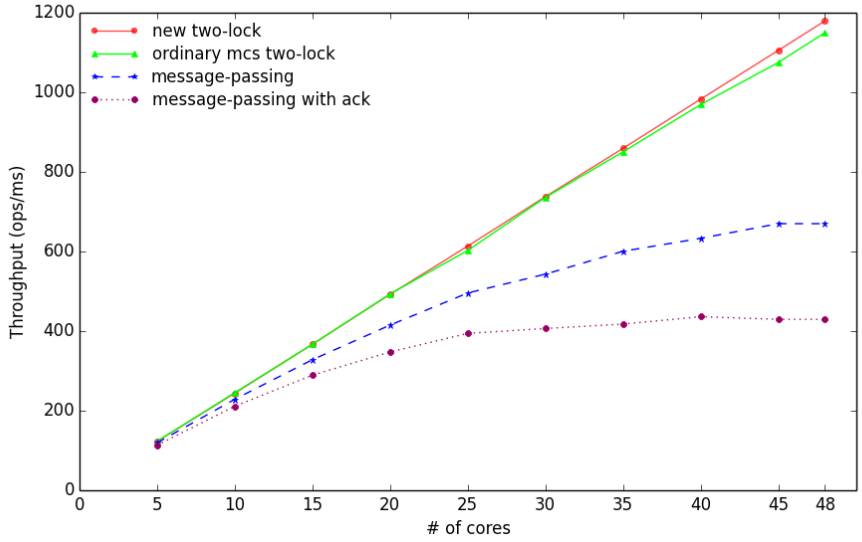


Figure 11:
Performance with an increasing number of cores at low contention

Figure 11 plots the system throughput under low contention conditions. Under these conditions, all algorithms give a linear system performance improvement with increasing number cores. One key observation is that the lock-based algorithms give almost identical performance figures.

In figure 10(b), we observed a significant drop in system fairness for the 2-lock algorithm as we increased the number of executing cores. We further investigated the performance of the system as a function of lock placement on the chip. In this experiment, we selected 5 cores scattered evenly around the chip. Then observed the performance of these cores as we increase the number of participating cores, with an additional 5 cores per run. The additional cores were also scattered uniformly around the chip so as to spread the traffic almost evenly on the mesh network. When running the experiments, we activated a single core on each tile until all the tiles become active, after this step, we started activating the second core on every tile.

Additionally, we changed the positions of the lock and then repeated experiment, so as to investigate how the performance of the system varies with lock placement in the grid. XY routing as used on the SCC motivated the decision to perform these experiments because it does not evenly spread traffic over the whole network. It normally causes the highest load in the middle of the chip. This creates the need to find an optimal location for the locks, such that the performance of the system does not degrade greatly as we increase the number of executing cores.

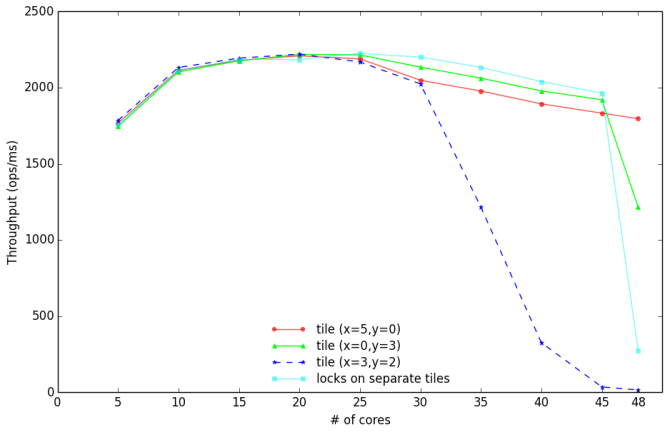
We plot the results in figure 12. Evidently, the performance starts to drops after activating more than 25 cores (i.e. all tiles active). The degradation of performance varies with the location of the locks, with the highest performance degradation experienced when we place the lock at tile (x=3, y=2).

When the lock was placed in the middle of the chip, tile (x=3,y=2), the high contention for the lock created very high traffic load in the middle of the chip which degraded performance to almost a halt. This stalling of the system was initially unexpected. We expected the data structure to allow at least one enqueueing and one dequeuing core to make progress at each moment in time. However on further analysis of the system configurations, we realized that as we increase contention for the lock, thereby increasing the traffic load on the system, the mesh network gets overloaded. With the network overloaded, the number of cores that make progress reduces tremendously. The core holding the lock delays to release the lock due to the high congestion on the network as a result of cores spinning to acquire the lock. The contending cores for the lock can not acquire it. This creates situation where the lock is not used, and thus no progress achieved by the system. This deterioration in system throughput is evident in 12(a), and it is worst when we placed the lock in the middle of the mesh.

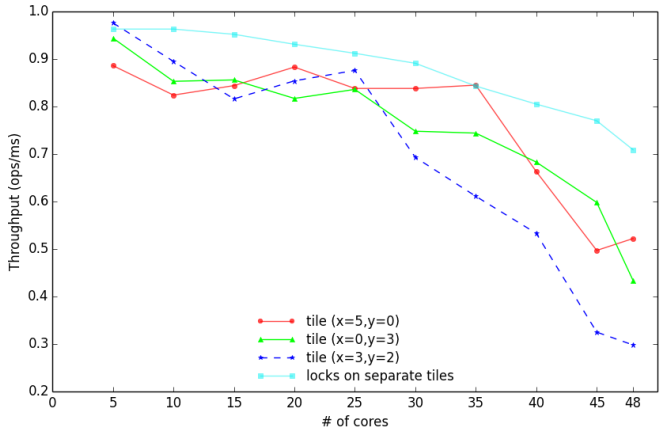
The poor performance of the locks with many participating cores at high contention led us to investigate their performance under low contention. The results of these tests are presented in figure 13, illustrating that, under low contention, congestion does not impact lock acquisition and release and we have a linear performance improvement with the increase in participating cores. One interesting result is that the system performance is identical irrespective of the location of the locks. Furthermore, we can observe that, placing the locks in the middle of the chip, we still achieve very good performance even with all 48 cores running. This confirmed the earlier hypothesis that network congestion delays release of the lock thus deteriorating system performance.

We also analyzed the throughput of each core relative to its distance from the locks. For this analysis, we plot the throughput of each core against the number of executing cores. This allows us to observe how increasing contention affects the core's throughput with regards to its location and distance from the locks. Figure 15 presents graphs of the results from 5 cores and different lock placements as discussed previously. As the figure shows, the performance of each core degrades with an increasing number of executing cores.

It is also evident that, for a lower number of executing cores, the cores closer to the locks have a higher throughput than those further from the lock. However as we increase the number of cores, the throughput of these cores deteriorates, and in some



(a) System throughput for different lock positions



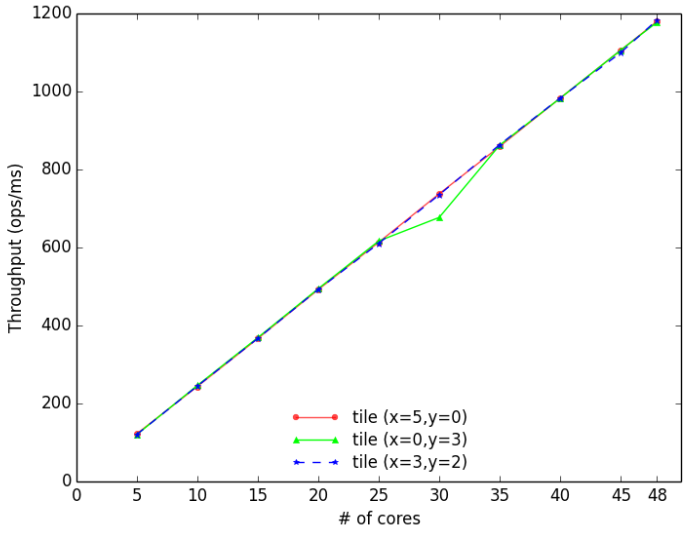
(b) Fairness

Figure 12: System performance (a) and fairness (b) as a function of lock location under high contention setting

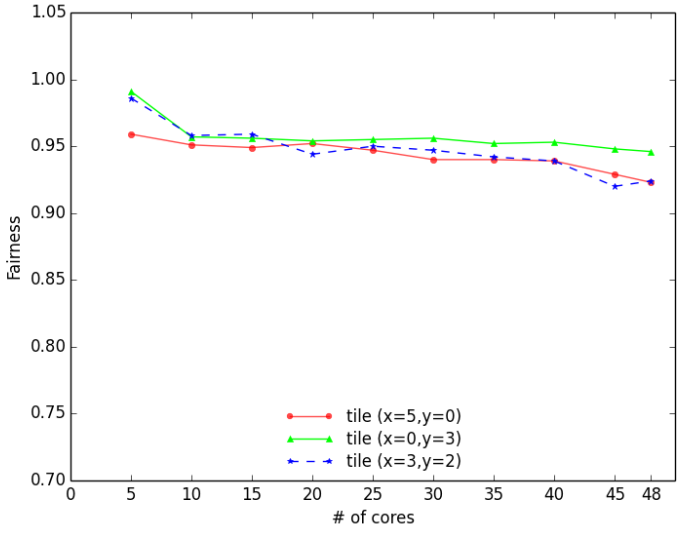
cases (figure 15), it is worse than for cores further from the locks. This is due to the congestion that builds up in the region of the chip that stores the lock, as the number of cores that contend for the locks increases.

To reduce the congestion on a single area of the chip for the 2-lock algorithm, we placed the locks in different locations on the chip. In this case, the cores achieved almost identical performance as no single core has a much better latency advantage with regards to the distance from both locks. This is evident in figure 14(d).

Overall, the results from the two-lock algorithm show that the position of the locks on the chip does have a significant effect on the general performance of the system. This is



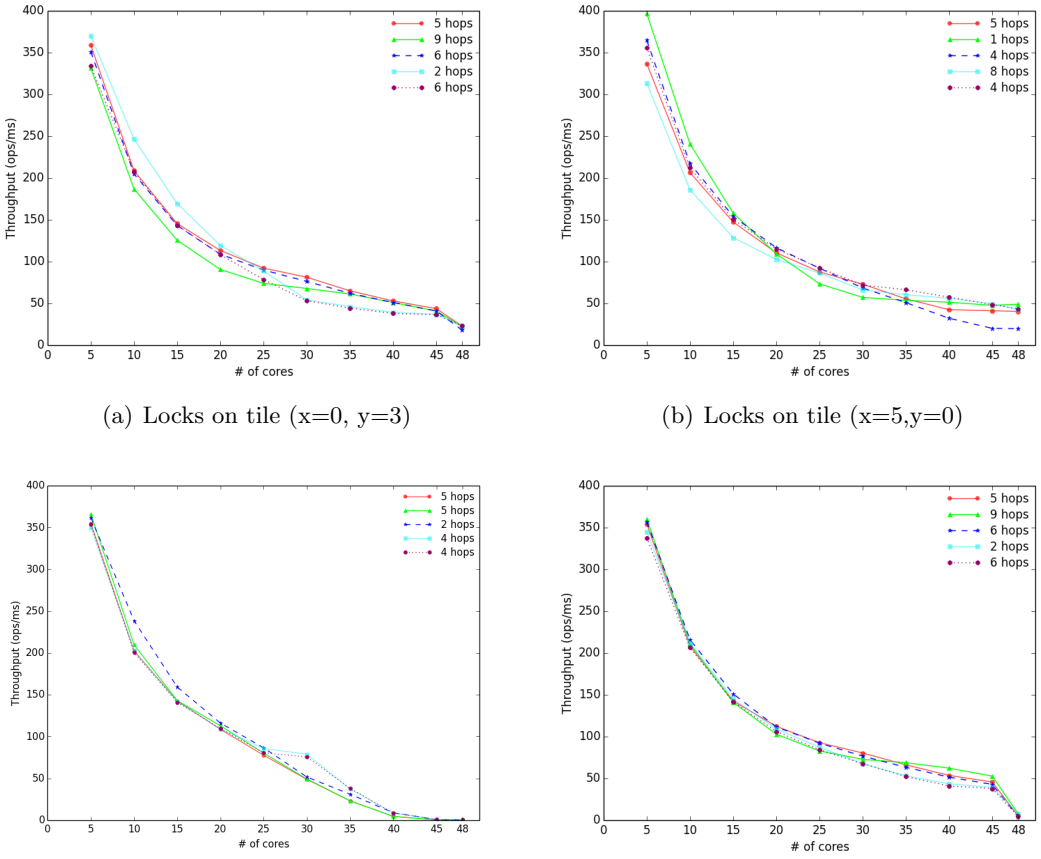
(a) System throughput



(b) Fairness

Figure 13: System performance (a) and fairness (b) as a function of lock location under low contention conditions

more evident as we increase the number of participating cores. As mentioned previously, positioning the lock in the middle of the grid always offers the worst performance and in some cases stalls the algorithms deteriorating performance to a halt. It has also been



(a) Locks on tile (x=0, y=3)

(b) Locks on tile (x=5,y=0)

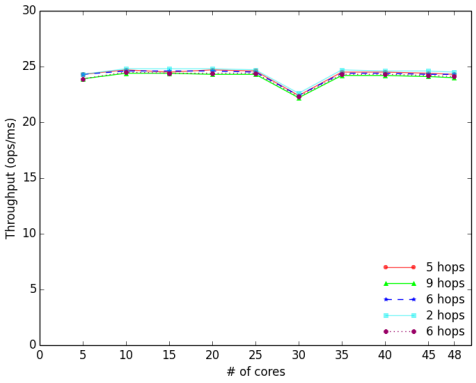
(c) Locks on tile (x=3,y=2)

(d) Separate tiles (x=0,y=3) and (x=5,y=0)

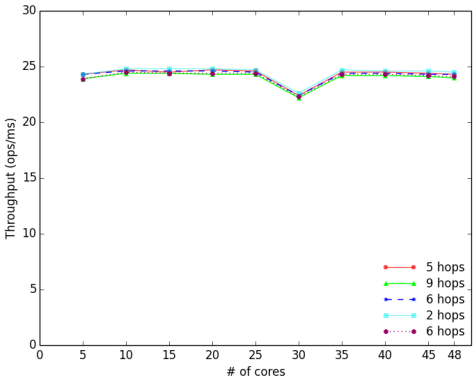
Figure 14: Core throughput with changes in lock location and an increasing number of executing cores at high contention. The lines in the different graphs represent the same core, thus show its throughput changes relative to the other cores, as its distance from the lock changes in the various runs of the computation.

observed that the lock-based algorithms give a higher performance throughput than the message-passing based algorithms, however, with a lot of bandwidth wasted in spinning for the lock and creating congestion on the network.

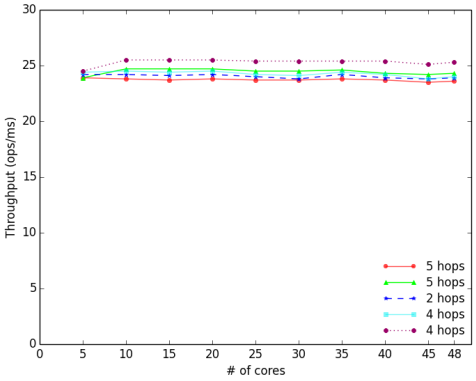
The messaging-passing based data structure, the server core creates a bottleneck for the execution of the algorithms as it serializes the requests for memory allocation. From the assumptions of Amdahl's law for multicore processors [27], we expected a performance improvement, if we run the cores asymmetrically over the homogeneous/symmetric system configuration. SCC provides for different tiles running at different frequencies, consequently, we can enhance performance by running the server core at a higher frequency than the rest of the cores on the chip. We performed experiments to test this theory, but we could not get consistent results over multiple runs of the experiment. The



(a) Locks on tile (x=0, y=3)



(b) Locks on tile (x=0, y=3)



(c) Locks on tile (x=3,y=2)

Figure 15: Core throughput with changes in lock location and an increasing number of executing cores at low contention

hardware gave erroneous values on reads from the message-passing buffers. These errors were intermittent, we believe they resulted from running the core writing to the buffer at a different frequency from the core reading from the buffer. We propose this as an avenue for further work on our message-passing based algorithms.

6.0 Conclusion

In this thesis, we explored procedures for synchronization on a distributed memory system. We presented a very fast and simple concurrent queue implementation based on the two lock queue by Micheal Scott. This queue provides a separate lock for the head and tail pointers, allowing one enqueue operation to proceed in parallel with a dequeue operation. We ported this queue to a distributed memory system, storing the queue in global shared memory, and using locks to synchronize the queue pointers stored in very fast message-passing buffers. We further improved the throughput of the queue by reducing the size of the critical sections in the algorithm. For hardware architectures with simple atomic operations such as *test_&_set*, we recommend using this approach. As the queue is very fast, and provides high throughput for a low number of concurrent processors.

We also presented queue implementation based on message-passing. In this implementation, exclusivity is awarded by the owner of the data structure. We implemented the data structure as owned by one processors, and other processors request exclusive access to the head or tail end of the data structure. This queue implementation can be exploited to exchange large amounts of data among processors in a cluster. The implementation incurs a lot of overhead, and should only be used for data structures that carry a large amount of data in the nodes of the queue.

6.1 Future work

The use of flags to indicate completion of the enqueue method increases bandwidth usage and memory access to the lock-based data structure. This added spinning on the dequeue process adds more blocking to an already blocking data structure. One avenue for further research is reducing the critical section size without relying on flags for correctness. Thus, release enqueue lock early, but also ensure that nodes are not dequeued before they are completely added to the data structure by the enqueuer.

All programs executing on the cores share the same locks, including the Unix operating system loaded on the cores. This usually leads to unanticipated behavior. A lock acquired by one process could be released by the operating system or any other program executing a write on the register used for the lock. This could best be addressed by designers of the hardware, probably by providing additional Test&Set registers to the hardware.

For the messaging-passing based procedures, the server handles requests in a round-robin fashion, thus requests are not handled in the order they are issued by the cores. We could also investigate further how to handle requests while maintaining the order in which they are issued by the cores. However this results into a queueing problem, which essentially we are trying to solve.

More research could also be done in the area of optimizing the performance of the messaging-passing based data structures. An interesting comparison would be attained by building these structures using the message-passing APIs available on the SCC instead of directly accessing the message-passing buffers.

Another area for further research is exploiting the inter-core interrupts to implement non-blocking message-passing protocols that do not require polling. The cores have the capability to interrupt each other, and this could be exploited for notification of other cores on completion of an event.

Bibliography

[1] "SCC External Architecture Specification (EAS)", Intel Cooporation (November 2010).

[2] R. Rotta, On efficient message passing on the intel scc, in: MARC Symposium, 2011, pp. 53–58.

[3] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, A. Agarwal, On-chip interconnection architecture of the tile processor, Micro, IEEE 27 (5) (2007) 15–31.

[4] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, G. Gao, A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture, in: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, 2006, pp. 10 pp.–.

[5] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, T. Mattson, A 48-core ia-32 message-passing processor with dvfs in 45nm cmos, in: Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International, 2010, pp. 108–109.

[6] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[7] P. Tsigas, 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, SPAA '01, ACM, 2001, pp. 134–143.

[8] A. Gottlieb, B. Lubachevsky, L. Rudolph, Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors (1981).

[9] J. M. Stone, A simple and correct shared-queue algorithm using compare-and-swap, in: Supercomputing '90., Proceedings of, 1990, pp. 495–504.

[10] J. D. Valois, Implementing lock-free queues, in: In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV, 1994, pp. 64–69.

[11] J. D. Valois, Lock-free linked lists using compare-and-swap, in: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, PODC '95, ACM, 1995, pp. 214–222.

[12] E. Ladan-mozes, N. Shavit, An optimistic approach to lock-free fifo queues, in: In Proceedings of the 18th International Symposium on Distributed Computing, LNCS 3274, Springer, 2004, pp. 117–131.

[13] S. Prakash, Y. H. Lee, T. Johnson, A nonblocking algorithm for shared queues using compare-and-swap, IEEE Trans. Comput. 43 (5) (1994) 548–559.

[14] M. Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. 13 (1) (1991) 124–149.

[15] M. P. Herlihy, J. M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.

[16] M. M. Michael, M. L. Scott, Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, in: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, PODC '96, ACM, 1996, pp. 267–275.

[17] M. P. Herlihy, Impossibility and universality results for wait-free synchronization, in: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing, PODC '88, ACM, New York, NY, USA, 1988, pp. 276–290.

[18] S. Peter, A. Schupbach, D. Menzi, T. Roscoe, Early experience with the barrelfish os and the single-chip cloud computer.

[19] D. D. Kandlur, K. G. Shin, Traffic routing for multicomputer networks with virtual cut-through capability, IEEE Trans. Comput. 41 (10) (1992) 1257–1270.

[20] P. Kermani, L. Kleinrock, Virtual cut-through: a new computer communication switching technique, Computer Networks 3 (1979) 267–286.

[21] W. Zhang, L. Hou, J. Wang, S. Geng, W. Wu, Comparison research between xy and odd-even routing algorithm of a 2-dimension 3x3 mesh topology network-on-chip, in: Intelligent Systems, 2009. GCIS '09. WRI Global Congress on, Vol. 3, 2009, pp. 329–333.

[22] T. Mattson, R. van der Wijngaart, "RCCE: a Small Library for Many-Core Communication", Intel Cooporation (May 2010).

[23] C. Clauss, S. Lankes, P. Reble, T. Bemmerl, Evaluation and improvements of programming models for the intel scc many-core processor, in: High Performance Computing and Simulation (HPCS), 2011 International Conference on, 2011, pp. 525–532.

[24] M. P. Herlihy, J. M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems 12 (1990) 463–492.

[25] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, Logp: towards a realistic model of parallel computation, SIGPLAN Not. 28 (7) (1993) 1–12.

[26] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, A study of the behavior of synchronization methods in commonly used languages and systems, in: Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, 2013, pp. 1309–1320.

[27] M. Hill, M. Marty, Amdahl's law in the multicore era, Computer 41 (7) (2008) 33–38.

[28] R. Kumar, D. Tullsen, N. Jouppi, P. Ranganathan, Heterogeneous chip multiprocessors, Computer 38 (11) (2005) 32–38.

[29] D. Cederman, P. Tsigas, Supporting lock-free composition of concurrent data objects: Moving data between containers, IEEE Transactions on Computers 99 (PrePrints) (2012) 1.

[30] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa, The design and implementation of a first-generation cell processor - a multi-core soc, in: Integrated Circuit Design and Technology, 2005. ICICDT 2005. 2005 International Conference on, 2005, pp. 49–52.

[31] H. Massalin, C. Pu, A lock-free multiprocessor os kernel (1991).