A Comparative study of the Cache Coherence and Moving Computation to Data Approach

Master of Science Thesis in the Programme Networks and Distributed Systems

ANIMESH BISWAS

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, December 2012

Examiner: Per Stenström

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden

Department of Computer Science and Engineering
Göteborg, Sweden December 2012

TO MY FAMILY

-ANIMESH BISWAS

Acknowledgements

This thesis could not have been prepared without the excellent supervision of Per Stenström. The necessary resources and information have been supplied by Chalmers University of Technology. Therefore, I want to thank Per Stenström and Chalmers University of Technology.

Abstract

As multicore computers are gaining widespread use, the main challenge for the future is how to design multicore systems that scale to hundreds of processor cores. One of the problems is how to implement the so popular shared memory model efficiently for future many-core systems. A critical mechanism to realize shared memory is to use cache coherence which allows multiple copies of a single memory block to be distributed across the caches attached to each core. Unfortunately, it is challenging to scale cache coherence mechanisms to hundreds of cores due to the latencies associated with communicating values and due to the complexity of the mechanism. An interesting alternative that has been considered in a recent research project is to allow only a single copy of a shared data structure and to force the computations that manipulate that data structure to be executed on the core that owns the data structure. While some tentative ideas for how to envision such a systems has been isolated, it is not clear how to implement such a system efficiently and whether the performance will be competitive with standard cache coherence solutions.

This thesis project aims at implementing a single-copy shared memory model on a Tilera system with 64 cores. In the project, a run-time system is designed and implemented, parallel applications are mapped to it and the performance of the system is established and compared with a system employing cache coherence.

Single-copy memory model will perform better than the coherent-based shared memory model as long as shared data size is smaller than the primary cache size. Because, in the single-copy memory model, core that is responsible for modifying shared data will encounter cache hit each time after the first access. On the other hand, in the coherent-based shared memory model, cores have to bring shared data from the remote memory for each access. However, coherent-based approach will start performing better when shared data size will exceed the primary cache size. Because bringing data from the remote cache takes less CPU cycle than from the main memory.

From the experimental result based on the critical sections of different applications (linear solver equation, Radiosity application of SPLASH-2 benchmark), I have found that single-copy memory model shows significant performance improvement over the coherent-based shared memory model. Because, in the single-copy memory model, core that modifies shared data structure encounters cache hit most of the time while executing critical section. Cache hit during accessing the shared data structures saves substantial amount of CPU cycles.

Keywords: Cache Coherence, Moving Computation, Shared Memory

Index

1. Introduction..1
2. Background..2
2.1 Shared Memory Multiprocessor Systems..2
2.2 Cache Coherence..4
2.3 Lock-based Synchronization Protocol..5
2.4 Problem in Cache Coherent Shared memory..6
3. Moving Computation Approach..7
3.1 Principle and Paradigm..7
3.2 Trade off between Moving Computation and Cache Coherence 8
4. Implementation..9
4.1 Overview of the TILE64 Machine..9
4.2 Tilera-64 Machine Facility for Moving Computation..10
5. Hypothesis Testing..12
5.1 Hypothesis Statement..12
5.2 Testing Methodology..12
5.3 Test Result..14
6. Microbenchmark Modification..16
7. Application Case Study..20
7.1 Linear Solver Equation Algorithm..20
7.2 Radiosity Application..21
8. Concluding Remarks..23
References..24

1. Introduction

A shared memory model is the most popular programming model used in mainstream multicore computers where each processor has its own private cache. The presence of cache in shared memory multiprocessor systems improve performance by reducing memory access time as well as memory traffic. When a processor needs certain data for either read or write, it first fetches data from remote memory module into its private cache. Further demand of same data by the processor can be meet from its private cache. Avoiding same data request over the interconnection network also decreases bandwidth requirement.

When multiple copies of a shared data block are distributed across caches attached to cores, then local modifications to any single cache copy of that data will lead to a global inconsistent view of memory. This is known as the cache coherence problem. A hardware cache coherent protocol is widely used in order to provide consistent view of shared memory systems. At any time a load instruction from a processor will get the most recent updated value and it does not depend on what values are in caches or main memory. The main principle behind cache coherent protocols is the single writer and multiple reader invariant (SWMR) which basically means only one core can have write permission when it is in modified (M) state and zero or more processor have read permission when they are in the shared (S) state [1].

In parallel programming, cache coherence allows threads to access shared data and the synchronization mechanism (e.g.lock) ensures the correct behavior of a parallel program. Conventional lock-based techniques are the most widely used synchronization technique. In lock-based synchronization mechanism, one process is allowed to enter a critical section grabbing the lock. Allowing one thread in a critical section at a time ensures correct behavior of a parallel program. Other threads must wait until the thread that executing critical section exits and release the lock. When a thread that enters the critical section, it needs to bring copies of the shared data structure from the remote memory into the private cache. Bringing shared data from the remote memory takes a significant amount of CPU cycles. The situation is even worse when hundreds of threads tries to access the same shared memory.

A new ”single-copy” shared memory model has been recently proposed in [3], which allows only a single copy of a shared data structure in one private cache and moving computation to the core which can only access the shared data structure. In short, it means that requested operations for manipulating shared data should be executed by the core who owns that data structure on behalf of threads that actually needs to access the shared resources. As there will be only one and the same core who actually modifies the data then loading shared data from remote memory should be done only once. Unlike a lock-based approach, further remote memory access is not needed. Because there will always be a cache hit after the first time. This is the main benefit of this approach in comparison with the lock-based approach.

The objective of this thesis is to find out under which conditions the single copy shared memory model performs better than the cache coherent shared memory model. In order to do this, first I have formulated a hypothesis and then I have setup a number of experiments to test the hypothesis through measurements.

I have hypothesized that moving computation to the data approach will be inferior than the lock-based approach as long as shared data size is smaller than the primary cache size. When data size exceeds the primary cache size, lock-based approach starts performing better than moving computation approach.

Several critical sections of different applications (linear solver equation, Radiosity application of SPLASH-2 benchmark) have been considered in order to test the hypothesis. By measuring execution time of different critical sections, I have found that moving computation approach shows significant performance improvement. Because core that is responsible for modifying shared data structure, encounters cache hit most of the time during execution of the critical section. But in the lock-based approach, shared data must be fetched from the remote cache in order to manipulate the data. Cache hit during shared data access, saves substantial amount of CPU cycle.

Advantage of single copy shared memory over cache coherent shared memory can be achieved as long as shared data structures of applications is being fit into L1 cache. In this way, overall execution time of applications can be reduced with the moving computation approach.

In Section 2, overview of the shared memory multiprocessor system, cache coherence protocol and lock-based synchronization mechanism are discussed briefly. Section 3 describes principle of the moving computation approach. Section 4 contains Tilera-64 machine facility for the moving computation to the data approach and overview of the Tilera-64 machine. Section 5 describes hypothesis statement, testing methodology and the testing result. Section 6 provides the details descriptions of the microbenchmark modifications. Section 7 contains parallel application case study results. Finally, Section 8 concludes the thesis by summarizing overall findings from the experimental result and providing the direction for the future work.

2. Background

2.1 Shared Memory Multiprocessor Systems

Cache coherent shared memory systems are widely used in today's commercial multicore architectures[1]. Parallel machines based on shared memory architecture are gaining wide spread acceptance in both research and commercial use. This architecture dominates not only in high performance computing system but also in end user systems (e.g. laptops, desktops and mobile devices).

Figure 1 shows a shared memory multiprocessor system organization where each processor has its own cache. Processors and shared memory modules are connected by a inter-connection network (e.g. Bus). Private caches attached to each processor help to reduce average memory access time and memory traffic.

In shared memory multiprocessor systems, processors interact via simple load and stores to the shared memory. This memory model also provides the facility of automated migration and replication of shared data structures in private caches [5]. But the replication of shared data in one or more caches at the same time can make the memory system incoherent. Cache copies of the shared memory needs to be consistent in order to ensure a coherent view of the memory. This problem is known as the cache coherence or cache consistency problem. Cache coherency is normally enforced by invalidate or update based cache coherence protocols.

Even though cache coherence allows threads to access shared data structures simultaneously, but only the synchronization mechanism provides the guarantee of correct execution of a parallel program. Synchronization mechanisms enforce threads to satisfy certain condition first before executing a sequence of actions so that operations associated with the shared data can be executed in one atomic step. Several synchronization mechanisms exist in today's multiprocessor system (e.g. lock, semaphore etc). In lock-based approach, a thread is allowed to modify shared data only when it can successfully grab the lock. After finishing the modification, a thread should release the lock so that other thread can modify the shared data. This way, lock-based techniques ensure that only one thread is allowed to modify shared data at a time.

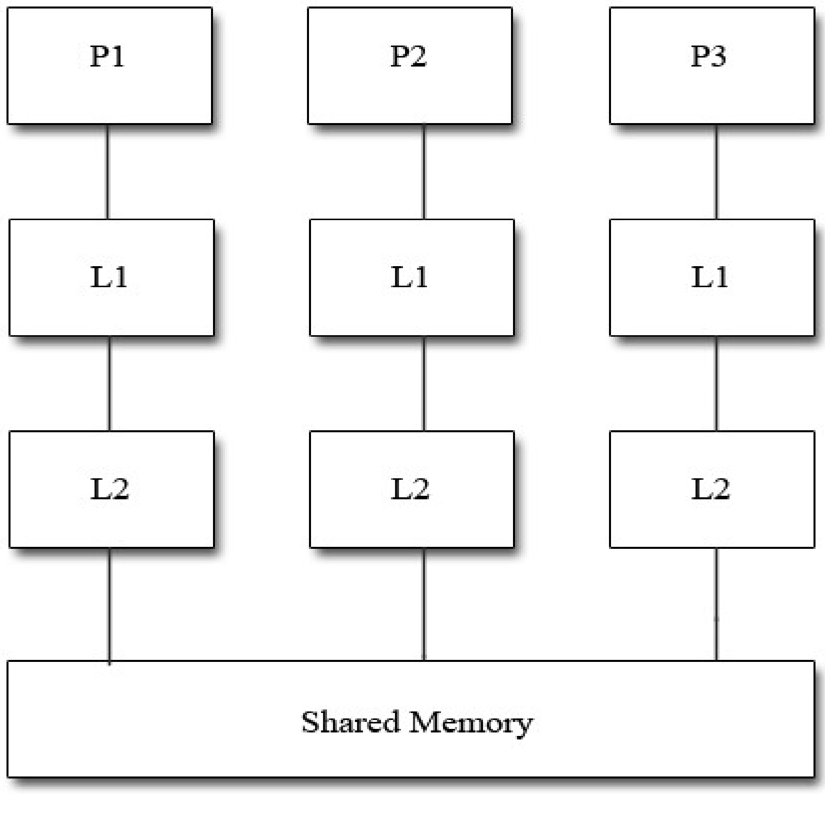


Figure 1 : Shared memory architecture

Cache coherence and synchronization mechanisms are described briefly in the following two sections.

2.2 Cache Coherence

Cache plays a vital role to reduce average memory access time in shared memory systems. The principle of locality allows caches to satisfy most of the memory request issued by a processor. Moreover, it does not require to inject memory request into the interconnecting network when requested data is available in the primary cache. In this way, the bandwidth requirement can also be greatly reduced. However, the presence of cache in the shared memory architecture raises the cache coherence problem.

Cache coherent memory allows to have copies of same memory block present in different caches of processors. If one processor modifies a copy of the shared memory location that exists in its private cache, then the other processors will still be able to read stale copies from their caches unless appropriate action is taken.

A mechanism is needed to ensure that updates are being propagated throughout the systems when modification of shared memory location is done so that copies of the memory block remain consistent in different caches as well as in main memory. Update propagation can be done by invalidating copies of memory blocks which are present in caches or updating copies with the newly modified value. The mechanism that ensures memory to be consistent is called a cache coherence protocol.

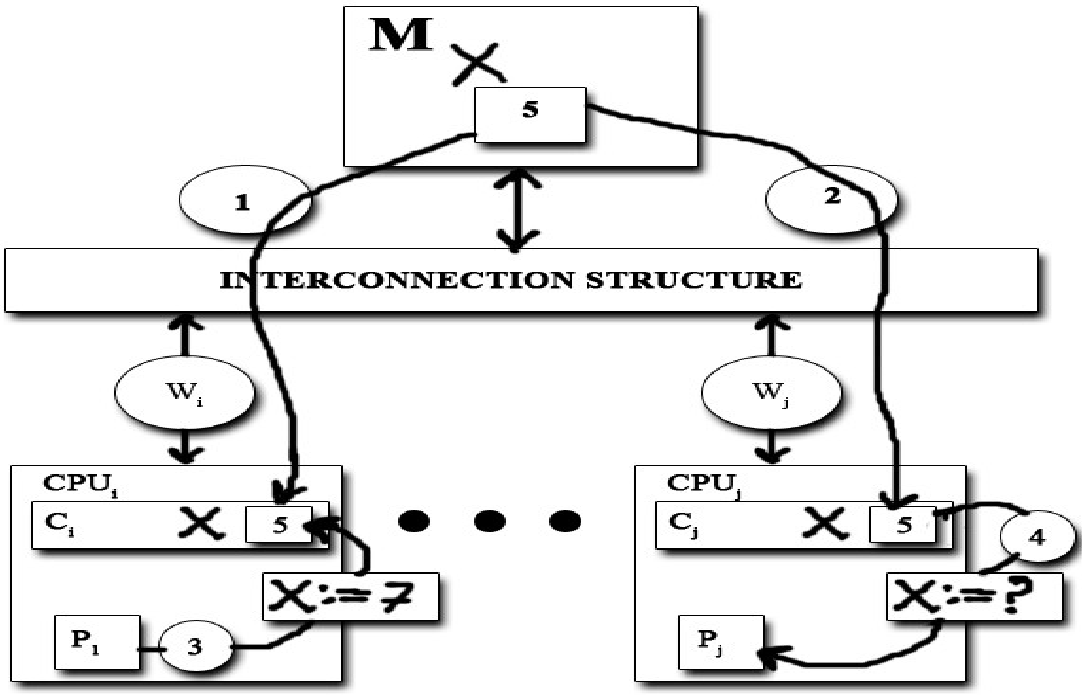


Figure 2 : Cache coherence problem

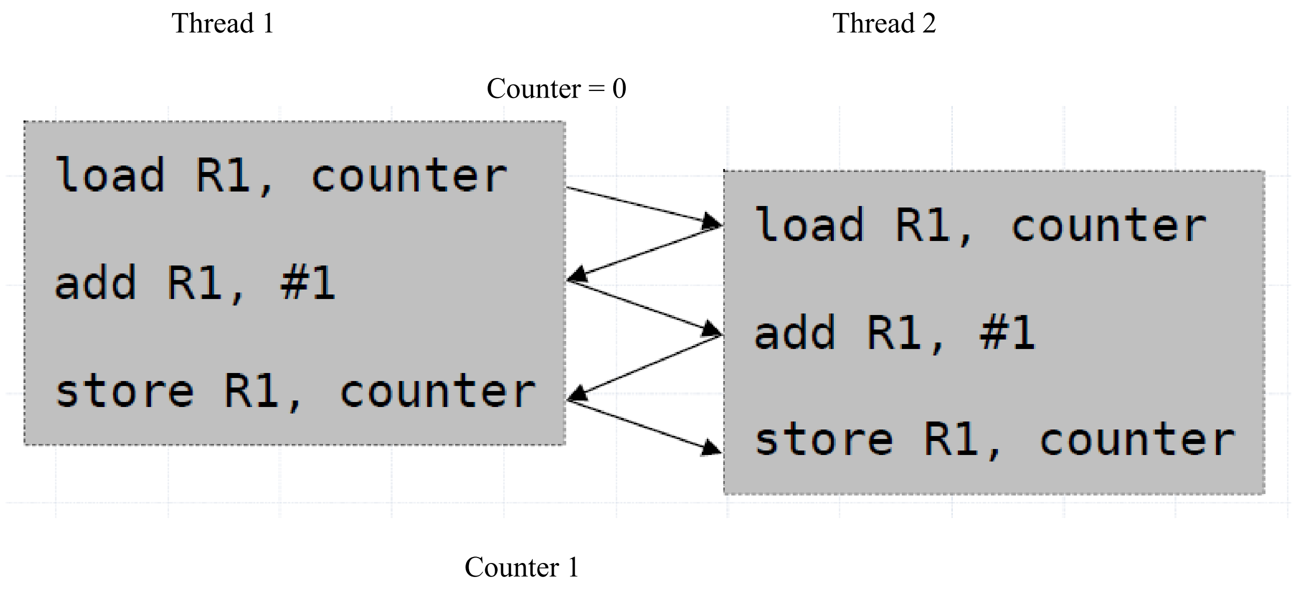
Figure 2 illustrates a shared memory system where M is the main memory and each processor Pi has a private cache Ci. In order to understand the coherence problem, let us assume that Pi and Pj brings the same variable X from the main memory M in their respective caches. If Pi modifies X which located in Ci then copy of X in Cj will become inconsistent. Invalidation based cache coherence protocol can be used to maintain consistency among caches.

Invalidation based cache coherence protocol ensures that there should be one valid copy of a memory block at any time. More specifically, a copy that has been modified most recently would be considered as the valid copy. Before modifying a cache copy, other copies of the same block should be invalidated. According to the Figure 2, a copy of X in Cj should be invalidated before Pi modifies X. A subsequent read request from processor Pj will cause a coherence miss. Pj needs to bring a recent modified version of the memory block from the remote cache Ci. In this way, any read request returns the latest write (which is the principle of the memory coherence). This is how invalidation based protocol keeps memory consistent.

2.3 Lock-based Synchronization Protocol

A synchronization mechanism refers to the coordination of threads that guarantees correct execution of a program and avoid unexpected race conditions. A conventional lock-based technique provides safe access to resources shared among multiple processors. But the resources must be protected by locks. The main purpose of a lock is to ensure serial access to the protected resources. Without programmer's effort it is hard to write correct parallel programs using the lock-based technique. Because improper use of locking can make a system deadlock.

Simultaneous access to shared data structures which are not guarded by locks can lead to race condition. In that case, correct behavior of the program can not be expected. Necessity of using synchronization mechanism can be observed from the example in Figure 3.



```
load R1, counter
add R1, #1
store R1, counter
```

```
load R1, counter
add R1, #1
store R1, counter
```

Figure 3 : Unexpected behavior of a parallel program

For the above example, let us assume that two thread named P1 and P2 each wants to increment the shared variable named counter by 1 and the sequence of instructions are executed according to the arrow direction. In that case, the final value of counter is 1 instead of the expected result of 2. This situation happens because the operations are interleaved. To get the correct value of the counter variable, the increment operations needs to be executed without interleaving. Lock-based technique enables a thread to execute instructions atomically (in one step). This way, synchronization mechanism ensures correct behavior of a parallel program.

2.4 Problem in Cache Coherent Shared Memory

Figure 4 demonstrates a simple shared counter update example where each thread increments shared counter by 1 after successfully grabbing the lock. This example highlights the key problem of the coherent-based shared memory system.

Let us assume that thread 1 has accessed the shared data before thread 2. Thread 1 should first bring the shared counter variable in L1 cache from remote memory in order to modify it. According to the invalidation based cache coherence protocol, thread 1 invalidates other copies (if any exist) of the counter located in different caches of other processors after modifying its own cache copy of the counter. Invalidation prevents other processor to read old value of the shared counter. When thread 2 wants to access the same shared data, it will bring a copy from thread 1 cache where this shared data has been accessed recently. In this way, threads spend huge amount of CPU cycle for bringing shared data structure in the private cache. If we consider the scenario when hundreds of thread wants to increment the shared counter, then thousand of CPU cycles will be needed for only fetching shared data from remote cache.

```
Thread 1                Thread 2

Lock()                  Lock()

Counter++               Counter++

UnLock()                UnLock()
```

Figure 4 : Shared counter update

But if we assume that there is a dedicated processor who has only exclusive access of the shared data and other threads who want to increment counter value are enforced to send request to that processor so that it can update the counter variable on behalf of them. In that case, the shared counter should be brought only once into the private memory of the dedicated processor. After that, when the dedicated processor wants to access the counter again, then this counter will be a cache hit each time. Thousands of cycle for bringing shared data in the case of cache coherent shared memory system does not need any more for this model. This is the main benefit of this new memory model called moving computation to the data [2] over the coherent based shared memory system.

Next section describes the details description of the moving computation approach.

3. The Moving Computation Approach

3.1 Principle and Paradigm

The basic principle of this approach is that the shared data area should be accessed by only one and the same core. This core is the owner of that shared data structure. Other cores who need to access (read/modify) the shared data structure will send request to the owner core. The owner core sends the acknowledgement back to the requester after modifying the shared data based on the request. Strict serialization of the accesses to the shared memory location can be ensured by allowing only one processing entity to do modification of the shared resources[7]. According to the principle, cores will have the following primary roles [2].

•**User Processing Entity (UPE)**: a processing entity who executes a normal application (non-CS) code.
•**Resource Guardian (RG)**: a processing entity that has the exclusive access to a shared data area and executes critical sections based on requests received from UPE instances.

In the conventional lock-based technique, a thread modifies a shared data structure with the computation only when it is allowed to enter critical section. Unlike this approach, a thread is enforced to send the computation to the resource guardian who has the authority to access shared data structure and thread waits until it receives acknowledgement from the resource guardian. When resource guardian (RG) receives the computation from UPE instances, it applies computation to the shared data structure. After finishing execution of the critical section, RG sends acknowledgment to the user processing entity (UPE).

In order to understand the whole procedure, we can consider the following simple example.

```
Thread 1              Thread  2

Lock()                Lock()

x  = x+C              x = x+C

UnLock()              UnLock()
```

Figure 5 : Shared update with constant

Thread 1 and Thread 2 want to increment shared variable x by constant C. Either one of them will be able to modify the value of x at a time as the critical section is protected by lock. In moving computation approach, threads will send constant (C) to the RG who is actually responsible for manipulating shared data. After receiving the constant from the threads RG will increase value of x by C. However, critical section code is kept permanently in the RG and necessary parameter (e.g. C) for that code has been moved to the RG.

A general programming structure for the moving computation approach can be defined based on the above description.

UPE instances	Resource Guardian
1. send computation	2. receive computation
	3. apply computation to the shared data
5. receive Ack	4. send Ack

Figure 6 : Basic programming framework

Cache coherence will not be needed any more for the moving computation approach because accessing shared data is only performed by the resource guardian. Moreover, if the shared data is kept permanently in the private cache of the resource guardian then memory traffic will be reduced significantly [7].

3.2 Trade-off between Moving Computation and Cache Coherence

This section describes benefits and limitations of the moving computation approach with respect to invalidation based cache coherence protocols. Performance improvement mainly depends on the shared data size and L1 cache size. Following two cases highlight this issue more briefly.

(a) Shared data size = 2 x L1 Cache Size :

Let us assume that two processors P1 and P2 want to access a shared data structure (e.g. array, linkedlist, etc.) and P1 will access the shared data before P2. The size of the shared data structure is two times bigger than L1 cache size. According to the assumption, half of the shared data would be enough to fill the L1 cache.

In the case of the invalidation based cache coherence protocol, P1 will bring first half of the data into L1 cache from the main memory. After accessing the first half, P1 will have to bring second half of the data structure from the main memory. As half of the shared data fills the L1 cache, second half of the data will replace the present content (first half) of the primary cache of P1. When P2 wants to access the first half of the data structure, it will bring this portion from the main memory and second half of the data should be brought from the cache of P1.

In the moving computation approach, RG will bring first half of the data from the main memory when it will receive first request to manipulate the shared data either from P1 or P2. RG will then bring second half of the data from main memory after modifying the first half. When satisfying the second request, resource guardian will encounter a cache miss due to presence of the second half of the shared data instead of the first half in L1 cache. In that case, resource guardian will have to bring the first half of the data from the main memory. Again, this portion of the data will replace second half of the data. In order to modify the second half of the data, RG will have to bring the second half from the main memory.

When data size exceeds the cache size, processor brings data from the remote memory in both cases. Resource guardian have to bring data from the main memory two times (first and second half). But in the cache coherence protocol based shared memory, a processor will bring part of the data from the main memory and will bring other part from the remote cache. It is costly to bring data from the main memory than remote cache in terms of CPU cycles. That's why cache coherence performs better than moving computation approach in this particular situation.

(b) Shared data size < L1 Cache Size :

Resource guardian will fetch the data from main memory during first access. After that it will be a cache hit each time. But according to the coherence based approach, a thread brings the data from the remote cache for each access. CPU cycle used for bringing purpose does not need in the case of moving computation approach. I have described benefit of the moving computation approach more briefly in Section 2.3.

4. Implementation

4.1 Overview of the TILE64 Machine

The Tile processor architecture consists of 64 identical general purpose computers connected by five 8x8 mesh networks. Figure 7 shows the diagram of the 64-tile TILE64 processor.

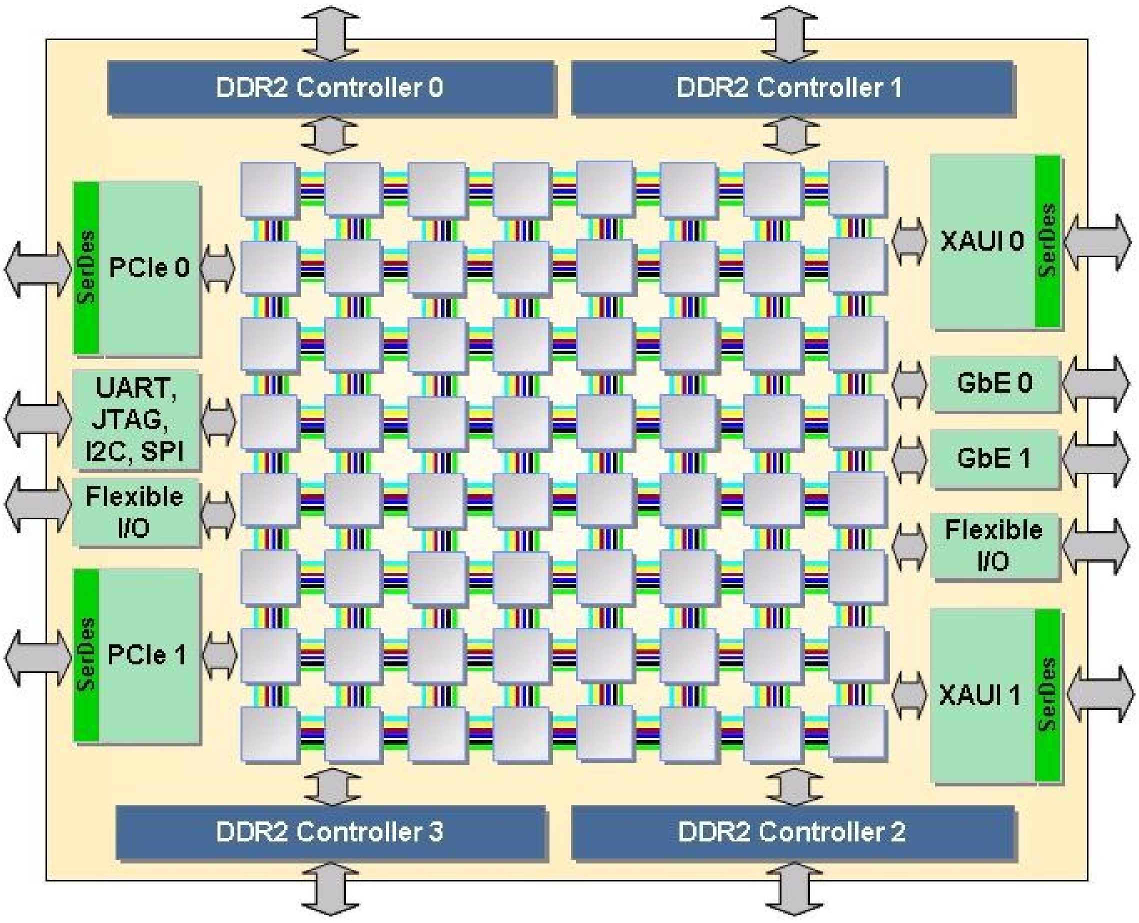


Figure 7: High level overview of the 8x8 grid of processors. I/O devices and memory controllers connect around the edges of the network (picture taken from "*wikipedia*").

Tile is the basic unit of Tile processor architecture. Each core is a fully featured computing system and capable of running entire operating system independently. Processor engine, cache engine and switch engine are the components of a tile. Each processor has a processor engine of 32-bit 3-stage VLIW pipeline with 64 registers. Processor runs at 1GHz clock speed and can perform 192 million 32-bit operations per second.

Each tile has a two level hierarchy cache system. L1(16KB) consists of a two-way set associative L1(8KB) data cache and a L1(8KB) instruction cache. L2(64KB) cache is a two-way set associative unified cache. A distributed L3 cache is formed by the L2 cache of each tiles. This cache is shared among all the tiles. Cache coherence for the shared memory is maintained in the following way.

A block of main memory is mapped on the L2 cache of a tile. This tile is called the home core for the particular memory address block. Core brings data block from the home L2 cache in the case of L1 cache miss. If the data is mapped into the local L2 cache then it simply fetches the data from the local L2 cache. If the data is mapped into the remote L2 cache, then core normally sends message over the dynamic network to the home tile in order to fetch the particular data block. Each home core maintains a directory of cores who are sharing the cache lines. If a remote core modifies a copy of shared cache line which is located in the private cache, it sends updated copy to the home core. After receiving the updated copy, home core sends invalidation message to the other core are currently sharing the the cache line.

Parallel threads running in different tiles can exchange messages, streams of data and scalar operands through the User Dynamic Network (UDN) [8]. More specifically, this network provides the facility of transferring data (integer, double, structure, etc.) between tiles. In this thesis, moving computation from the UPE instances to RG as well as transferring acknowledgement from RG to UPE instances is done by the UDN.

UDN consists of four hardware queues and one catch-all queue. A receive side buffer of a tile (128 words) is shared between the hardware queues. Tilera provides hardware demultiplexing based on a tag associated with a data packet. When a packet is reached at destination end, then a tag match occurs. Based on the matching, hardware demultiplexes the data into the appropriate queue. Data packet is stored in the catch-all queue if the tag does not match with the tag that the receiver is interested in [8].

4.2 Tilera-64 Machine Facility for Moving Computation

The main intention behind using this machine is that it provides facility to transfer computation from core to core. Tilera has built in C library functions which can be used to send and receive computation from processors to processors. For example,

```
tmc_udn_send_1(header, UDN2_DEMUX_TAG,data);
```

Figure 8 : Sending one word data packet.

This function call can be used to send one word (4 byte) size data packet. There are three parameters used in this function which are header, a demux tag and data. Header acts like address of a processor to whom the data is to be sent. There are five hardware FIFO queues associated with each processor where data is actually stored after being reached at destination processor. When data is available either one of the queues, processor fetches the data from that queue. Sender should specify the queue number on which data will be stored at the receiving end. Figure 10 describes a common scenario of sending and receiving computation between one core to another core in the Tilera-64 machine. According to Figure 8, UDN2_DEMUX_TAG indicates that data will be available in the second queue. The final parameter is the data that will be sent. There are also functions for receiving packets. For example,

data0 = tmc_udn2_receive();

Figure 9 : Fetching one word from 2nd demux queue.

A core that wants to receive data packets, should execute the above receive statement and will wait for the data to be available in the demux queue. According to Figure 9, thread will listen in the second demux queue for the data. A thread can fetch at most one word size data packet by calling the receive function once. More specifically, receive procedure should be called N times to get N packets from the queue. Receive statement is a blocking function call. A thread can not be allowed to execute next instruction until a receive function call returns data. Programmer should be aware while calling the receive function. If the proper sending statement is not found against receive call then the system will become deadlock. This situation happens often when queue number is mentioned wrongly in either sending or receiving statement.

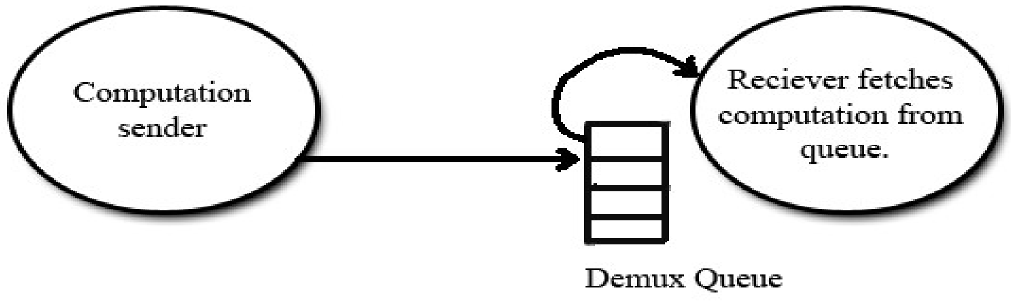


Figure 10 : Transferring computation between core to core of Tilera-64 machine

The following functions have been used in this thesis for sending and receiving computation.

Data Type Size	**Sending Functions**	**Receiving Functions**
One word size data type	tmc_udn_send_1()	tmc_udn0_receive()
Type size more than one word (double)	tmc_udn_send_buffer ()	tmc_udn0_receive_buffer()

Pair of one word size data type (Integer,Integer)	tmc_udn_send_2 ()	Word1 = tmc_udn0_receive ()
		
		Word2 = tmc_udn0_receive ()

Table 1 : Sending and receiving functions used for modifying critical sections.

5. Hypothesis Testing

5.1 Hypothesis Statement

Moving computation to the data approach will perform better than the lock-based approach as long as shared data size is smaller than the primary cache size. When data size exceeds the primary cache size, lock-based approach starts performing better than moving computation approach.

5.2 Testing Methodology

Critical section execution time (CSET) is measured using Tilera's built in C library function called *get_cycle_count()*. This function call returns CPU cycles that have been spent so far.

Figure 11 shows the CSET measurement procedure for the lock-based approach. *totalcycleCount* is the difference between CPU cycles spent before beginning of the critical section execution and CPU cycle is needed until the end of the critical section execution.

```
start = get_cycle_count()
    Lock
      C S
    UnLock()
end = get_cycle_count()
totalcycleCount = end-start
```

Figure 11 : Lock-based critical section execution time

Several factors need to be considered to measure CSET for the moving computation approach. Total execution time is the summation of the following factors.

- Sending computation time
- Receiving computation time

- Critical section execution time (RG applies computation to the shared data)
- Sending Acknowledgement (If any) time
- Receiving Acknowledgement (If any) time
- Header (processor address) creation time

Processor who wants to receive computation should wait until the computation is reached at its end. When a computation arrives at destination processor, it is first stored in one of the built in hardware queues of the receiver. After that, the processor fetches the computation from that queue. In Tilera-64 machine, waiting time of a processor for receiving computation is proportional to amount of time that a computation sender spends before sending computation. Because processor that wants to receive computation, starts spinning over a queue until it receives computation. In Figure 12, we see that amount of time that a receiver spends for spinning is almost equal time that a sender spends before sending computation. The same thing happens in the case of receiving acknowledgement from the RG. Spinning for receiving computation or acknowledgement wastes substantial amount of CPU cycles. Waiting time to receive computation or acknowledgement completely depends on the how long a sender spends time before sending computation. It basically varies from application to application. For example, in the linear solver equation, each processor spends huge amount of CPU cycles to manipulate a set of rows of an array before entering into the critical sections. In this particular case, RG has just spent the same amount of time for spinning over the queue (on which computation will arrive) in order to receive computation. If we consider this huge amount of waiting time then only the waiting time dominates the total execution time of the lock-based approach. Moreover, during measuring the execution time of the critical sections of the radiosity application, I have found that each critical sections has different amount waiting time associated with it.

In the moving computation approach, resource guardian starts waiting for a computation from the beginning of a program execution. If we start measuring the receiving computation time from the beginning, then we are actually measuring the time that all UPE instances spend before sending computing to the resource guardian for modifying shared data. But for the lock-based approach, we have measured CSET time only when a thread enters into the critical section. Moreover, if we count waiting time for receiving acknowledgement from the resource guardian, then we are actually counting shared data modification time twice. Because the time of shared data modification done by resource guardian is proportional to the waiting time for receiving acknowledgement. But execution time of shared data access is already counted once (3rd factor) in the total CSET measurement time. Waiting time for receiving computation is ignored completely for measurement of the moving computation approach. For the moving computation approach, receiving mechanism should be implemented in such a way that waiting time can be minimized as much as possible. However, this thesis does not solve the efficiency issue of receiving mechanism.

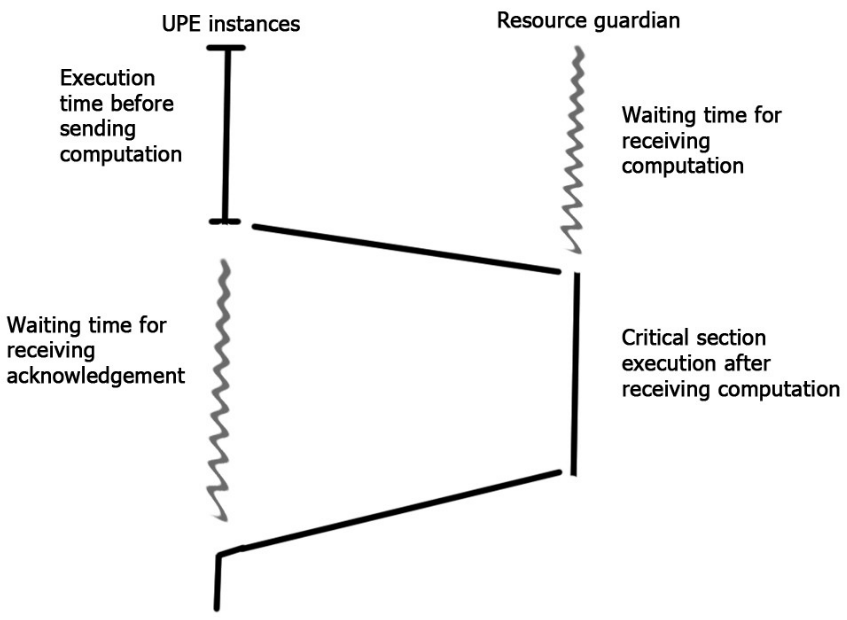

UPE instances
Resource guardian
Execution time before sending computation
Waiting time for receiving computation
Waiting time for receiving acknowledgement
Critical section execution after receiving computation

Figure 12 : Waiting time proportional to execution time before sending computation

5.3 Test Result

In order to test the hypothesis, a shared array is accessed in the critical section. Execution time is measured for the following two cases.

I) Array Size < L1 Cache Size

II) Array Size > L1 Cache Size

According to the Section 3.2, moving computation approach will perform better than lock-based approach when array size is smaller than the primary cache. Because RG will encounter cache hit each time after the first access. On the other hand, lock-based approach will start performing better when array size will exceed the cache size. Because bringing data from the remote cache takes less CPU cycle than from the main memory. CSET is measured according to the procedures described in Section 5.2.

Shared Array manipulation :

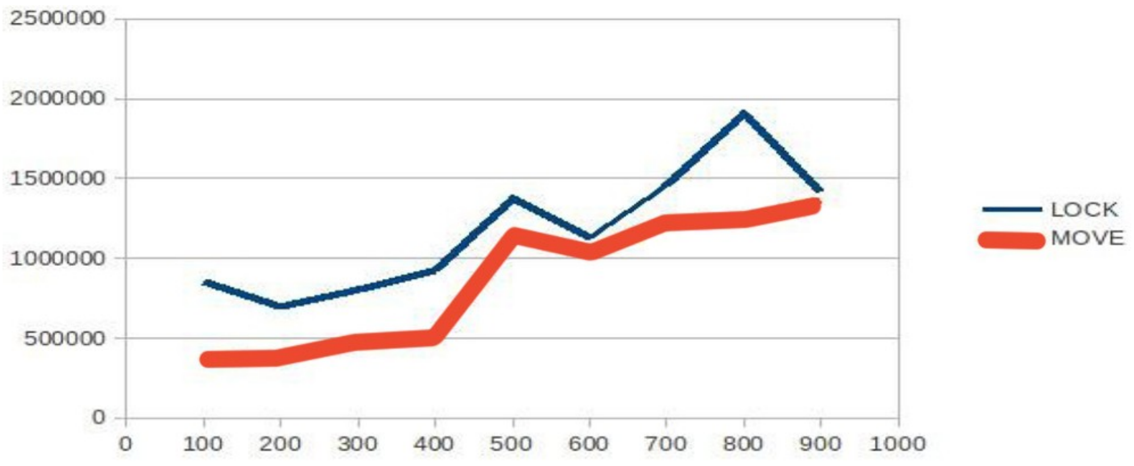


Figure 13.a Array Length < L1 cache size. Bold line indicates moving computation approach and the thin one represents lock-based approach.

X-axis represents the array length and Y- axis represents the execution time (CPU cycle) in Figure 13(a&b). Here, we can see that moving computation approach always takes less CPU cycles than the lock-based approach. Even though, distance between two approach decreases in two points (e.g. 600, 900), but these two approaches never intersects with each other. When the array size (900) is increased enough to almost fill out the L1 cache size, then we see that lock-based approach starts to decrease significantly.

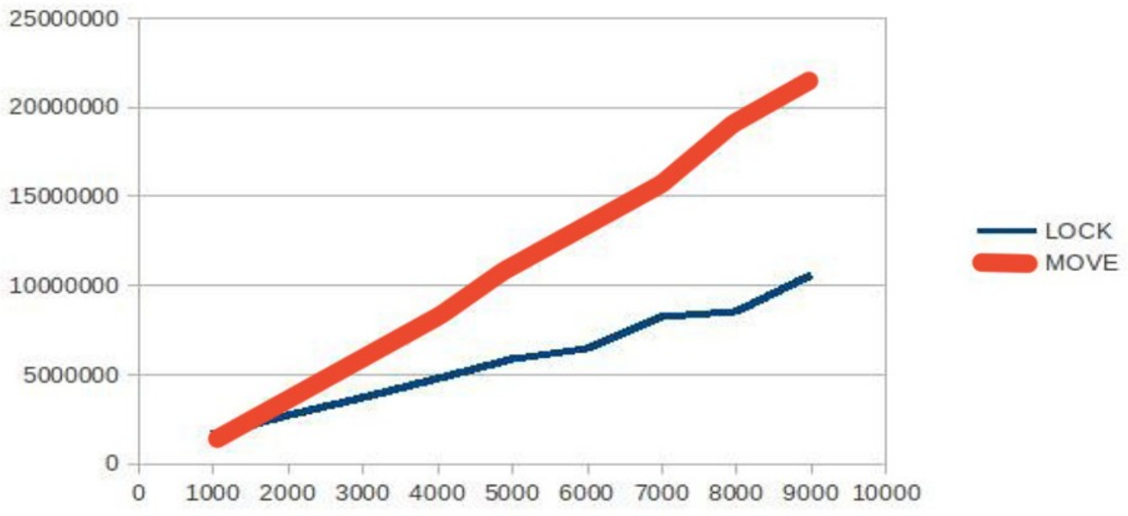


Figure 13.b Array Length > L1 cache size.Bold line indicates moving computation approach and the thin one represents lock-based approach.

From the above graph, we can see that two approaches shows almost opposite characteristics than Fig. 13.a. Difference between two approaches increases proportionally with respect to the array size (X-axis). Starting point of the graph (array size is 1000) is the threshold point for the moving computation approach. After that, moving computation approach grows linearly over time.

Next section describes how modification of microbenchmark is done.

6. Microbenchmark Modification

This section describes the microbenchmark that has been modified according to the moving computation approach. Before discussing the modifications, we have to remember that access to the shared data structure should only be done by the resource guardian. A thread who executes normal application code and wants to access the shared data structure, should send request to the resource guardian. When a thread sends a request to the resource guardian, it should send necessary computations that a resource guardian needs to modify the shared data structure. After finishing execution of the critical section, resource guardian sends acknowledgement to the requesting processor.

Shared Array Modification

```
  for(i=0;i<N;i++)

shared_array[i]+= constant(C)
```

Figure 14 : Shared Array Modification critical section

When a thread enters the critical sections, it adds constant (C) with each element of the shared array. In the moving computation approach, a thread sends constant (C) to the resource guardian. When a resource guardian receives the constant, it adds that constant with each element of the shared array. Each thread sends the constant (one word size) by calling the following functions.

tmc_udn_send_1(header, UDN0_DEMUX_TAG, (unsigned long) (c))

Figure 15 : Sending one word size data

The first parameter of this function call is the address of the resource guardian processor. Second parameter is the queue number of the resource guardian where words will be stored when it is reached at the receiver end. The third parameter is the data that is sent to the resource guardian.

Resource guardian receives the constant by calling the tmc_udn0_receive() function. This blocking function returns one word data when it is available in the first queue of the resource guardian. After receiving the constant, resource guardian executes the code mentioned in Figure 14.

Linear Solver Equation

In the lock-based technique, a thread adds local_sum to the shared variable called total_sum after entering the critical section. Both total_sum and local_sum are double type data.

```
total_sum += local_sum
```

Figure 16 : Linear Solver Equation critical section

In Tilera-64 machine, sending and receiving a packet of words (data size more than one word) is typically done by calling tmc_udn_send_buffer() and tmc_udn0_receive_buffer() functions. These functions takes memory reference of the data structure as parameter instead of value. However, before sending the double type local_sum, each thread puts the local_sum in the structure called “Sum”. A thread sends the memory reference of the structure by calling the following function.

tmc_udn_send_buffer (header, UDN0_DEMUX_TAG, val, struct_size_in_words);

Figure 17 : Sending double type data

Resource guardian receives the “Sum” structure by calling tmc_udn0_receive_buffer(recv, struct_size). Third parameter of the receive_buffer() is the memory reference of a “Sum” structure and fourth parameter is the size of the structure in words. When packet of the structure is reached in the first demux queue of the resource guardian, then the resource guardian dereferences the “recv” memory reference to get the local_sum value. After that, resource guardian executes the code stated in Figure 18 (a).

total_sum +=
recv->local_sum;
	typedef struct{
	    double local_sum;
	}Sum ;

Figure 18: (a) Fetching local_sum value by resource guardian. (b) “Sum” structure.

Radiosity Benchmark Application

The following four critical sections have been modified according to the moving computation approach.

```
Element* p = global->free_element ;

global->free_element = p->center ;

global->n_free_elements-- ;
```

Figure 19 : Get Element critical section

Get Element

Each thread sends a request to the resource guardian to get memory reference of the “Element” type structure. A thread sends the Id (processor number which is one word size) to the resource guardian by using the tmc_udn_send_1() function. When resource guardian receives a request, it basically executes the code mentioned in Figure 19 and sends the memory reference “p” to the requesting thread as an acknowledgement. Resource guardian sends memory reference (one word size data) by calling the tmc_udn_send_1() function. Before sending the “Element” type memory reference, resource guardian takes the address of the requesting thread by using the processors Id as the index from the address array. This array contains all the address of the processors. Address of a destination processor should be specified as a argument of the tmc_udn_send_1() call.

Get Patch

Getting a patch is similar like the above “Get Element” critical section. The only difference is that thread enters into the critical section to get memory reference of the structure named “Patch” instead of the “Element” type memory reference. For this reason, this critical section has also been modified in the similar way.

Get Interaction and Free Interaction

Thread enters into the “Get Interaction” critical section to take a memory reference of a “Interaction” type structure and returns a “Interaction” type memory references by entering the “Free Interaction” critical section. Both critical sections are protected by the same lock. This is why, resource guardian executes both critical sections on behalf of the threads who want to get “Interaction” type memory reference as well as who want to return “Interaction” type memory reference.

tmc_udn_send_2 (RG address, UDN0_DEMUX_TAG, senderId, 0) function is used to send request for getting “Interaction” type structure. Resource guardian uses this senderId to send acknowledgement back to the requester after executing the critical section.

In “Free Interaction” critical section, thread returns “Interaction” type memory reference by calling the tmc_udn_send_2 (RG address, UDN0_DEMUX_TAG, senderId, memory_address). Parameters of the function call is similar to the above function. Fourth parameter is memory reference of “Interaction” type structure that a processor wants to return.

In order to receive one of the above requests (either getting or returning a interaction memory address), resource guardian will call the tmc_udn0_receive(). This receive function call returns one word data. For this reason, resource guardian have to call this receive function twice to get two words (senderId, memory_address/0). Execution of a critical section depends on the second word which either memory_address or 0. I have used “0” as the fourth parameter for the getting interaction request function call in order to differentiate requests. However, if the resource guardian receives a “0” then it will execute “Get Interaction” critical section. Otherwise it will execute “Free Interaction” critical section. More specifically, when resource guardian receives a memory address from the free interaction request, it adds the reference to the “Interaction” type memory reference linked list. In the case of getting interaction request, it returns the head of the linked list to the requesting core.

Resource guardian sends acknowledgement back to the requesting thread in both cases (get and free). Sending acknowledgement is done by calling the tmc_udn_send_1() function. If the resource guardian receives “Get Interaction” request then it sends memory reference to the requesting core. For free interaction, it just sends a dummy value to notify the requesting thread that the memory reference has been added to the linked list of the “Interaction” type memory reference. To understand the whole procedure, consider the following pseudo code based figure.

UPE instance - sends get/free interaction request	**Resource guardian**
Send2 (thread id, memory_address/0)	thread id = receive()
//Both memory_address and 0 have one word size	second_word = receive()
	if (second_word == 0)
	{
	// execute get interaction critical section
	// send memory reference as Ack
	} else{
Ack = receive()	// execute free interaction critical section
// Either returns memory reference or dummy value.	// send dummy value as Ack to notify the //requester that CS-execution is finished.
	}

Figure 20 : Get Interaction and Free Interaction pseudo code

Task Counter

After entering the critical section, thread executes the below code in order set the flag value either by 0 or 1. In the moving computation approach, a thread sends request to the resource guardian by calling tmc_udn_send_1 (RG_address, UDN0_DEMUX_TAG, threadId) function. When resource guardian receives the request, it executes the above code illustrated in Figure 21. After executing the critical section, it sends the flag value to the requesting thread as acknowledgement. tmc_udn_send_1 (thread_address, UDN0_DEMUX_TAG, flag_value) is used to send flag value to the requesting processor.

```
Long flag = 0;

if (task_counter == 0)
    flag = 1 ;

task_counter++ ;

if( task_counter >=
    n_processors )

    task_counter = 0 ;
```

Figure 21: Task counter critical section

7. Application Case Study

In this section, experimental result of critical sections of different parallel application are discussed. Moving computation approach shows significant performance improvement over the lock-based approach for all of the critical sections. The term “Improvement%” denotes performance improvement (%) of moving computation approach over lock-based approach.

7.1 Linear Solver Equation Algorithm

In the linear solver equation algorithm, threads manipulates set of rows of an NxN matrix in order to calculate local_sum. After entering the critical section, they add the local_sum with shared variable called total_sum.

Iteration count	Improvement(%)
21	81%

Table 2 : Performance improvement of linear solver equation

Linear solver equation algorithm normally terminates when it reaches certain threshold point. For this experiment, i have considered fixed number of iterations (21) instead of letting this algorithm to converge. I have found that moving computation to the data approach shows drastic improvement (81%) than lock-based approach.

7.2 Radiosity Application

Critical section	Improvement(%)
Get Element	74%
Get and Free Interaction	82 %
Get Patch	41%
Task Counter	61%

Table 3 : Performance comparison of different critical sections of Radiosity Application

Table 3 describes CSET improvement of moving computation approach over lock-based approach. Each critical sections mentioned in the above table are measured individually. Performance improvement depends on the number of times a critical section is executed. For example, both “Get Element” and “Get Patch” critical sections are almost similar. But total number execution of “Get Patch” critical section is much more less than “Get Element” critical section. For this reason, “Get Patch” shows less improvement ratio than “Get Element”. However, highest improvement ratio is found in “Get and Free Interactions” critical sections and the lowest one is in “Get Patch“.

Shared data size of linear solver equation algorithm and “Task Counter” critical section of the Radiosity application are smaller than the private cache size. For this reason, RG always encounters cache hit after the first time access. But, in the lock-based approach, threads have to bring the latest copy of the shared data from the remote cache each time. That's why moving computation shows performance gain for these critical sections.

In the “Get Element”, “Get Patch” and “Get and Free Interactions” critical sections, size of the shared data structure (linkedlist) is much more larger than the primary cache. Unlike the shared array modification, thread manipulates one or two elements of the linkedlist in each access instead of modifying the whole shared data structure at a time. For example, during the execution of the “Get Element”, “Get Patch” critical sections, threads take the head of the shared linkedlist and set the next adjacent element of the head as the head of the linkedlist. In order to do this, cores bring the head of the linkedlist from the remote cache of a core where this linkedlist was accessed recently. But the next adjacent element of the head should have to bring from the main memory. Because this element will be accessed for the first time. The adjacent element of the head will be the next head of the linkedlist for the next execution. On the other hand, RG will bring head and the next element form the main memory during first execution. For the second request, RG will encounter a cache hit for the head (second element in that case). However, RG have to bring the third element from the main memory which will be the head of the linkedlist. As we can see that resource guardian will encounter cache hit for the head of the linkedlist but cores have to bring the head from the remote cache for each access in the coherent-based shared memory model.

“Get Interactions” and “Free Interactions” critical section are protected by the same lock. This is why, resource guardian executes both critical section on behalf of the threads. “Get Interactions” behaves similar like “Get Element” and “Get Patch”. Thread enters “Free Interaction” critical section to put back a element in the shared linkedlist. Returned element is actually inserted in front of the linkedlist and its next pointer points to the current head of the list. In this way, returned element becomes the head of the linkedlist. In the lock-based approach, cores that want to return a element, have to bring the current head from the remote cache. But, in the moving computation approach, resource guardian encounters cache hit for the current head of the linkedlist. Because resource guardian brings the head node in early access (because of either “Get interaction” or “Free Interaction” critical section execution). Encountering cache hit for the head node of the linkedlist improves performance substantially in the case of moving computation approach.

8. Concluding Remarks

The performance trade-off between lock-based approach and moving computation to the data approach is discussed in this thesis. From the hypothesis testing result, it can be said that a significant amount of performance gain can be achieved by the moving computation approach as long as data structure fits into the private cache of the resource guardian. But, when the shared data size exceeds the primary cache size, lock-based approach starts performing better than the moving computation approach. Because bringing data from the remote cache takes less CPU cycles than from the main memory. This is the limitation of the moving computation approach.

By analyzing the different critical sections of the Radiosity application, we can see that moving computation approach will also be beneficial when resource guardian will be able to reuse the cached shared data for processing future requests of the UPE instances. Moreover, benefit of the moving computation to the data can be achieved by increasing the L1 cache size of the resource guardian. Because hit ratio of the shared data will also be increased in this way.

Sending and receiving primitive of computation should be implemented in such way that waiting time for the receiving computation can be minimized as much as possible. I have ignored waiting time completely. But, in reality, it is not possible to ignore waiting time completely. Because certain amount of overhead is associated with receiving procedure of a computation which is ignorable. This is why, further investigation is needed for efficient sending and receiving procedure implementation.

References

[1] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. "Why On-Chip Cache Coherence Is Here to Stay". *Communications of the ACM, vol 55, no 7, July 2012.*

[2] BA Hong. ”Hardware-based Synchronization Support for Shared Accesses in Multi-core Architectures”. *ACST '08 Proceedings of the Fourth IASTED International Conference on Advances in Computer Science and Technology.*

[3] Vajda, A. ”Handling of Shared Memory in Many-core systems without Locks and Transactional Memory”. *Accepted at the 3rd Workshop on Programmability Issues for Multi-core Computers (MULTIPROG), with HiPEAC 2010 conference.*

[4] www.Tilera.com. ”Application-Libraries-Reference-Manual-UG227”.

[5] David E. Culler, Anoop Gupta, Jaswinder Pal Singh. *”Parallel Computer Architecture: A Hardware/Software Approach”*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1997, ISBN 1558603433.

[6] James Archibald, Jean Loup Baer. ”Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model”. *ACM Transactions on Computer Systems (TOCS), Volume 4 Issue 4, Nov. 1986.*

[7] Vajda, A. ”The Case for Coherence-less Distributed Cache Architecture ”. *4th Workshop on Chip Multiprocessor Memory Systems and Interconnects, with HPCA 2010 conference.*

[8] D Wentzlaff. ”On-Chip Interconnection Architecture of the Tile Processor”. *IEEE Micro Volume 27, Issue 5, September 2007.*